Modern C++
Object-Oriented Programming

"Combine old and newer features to get the best out of the language"

Margit ANTAL
2020
C++ - Object-Oriented Programming

Course content
- Introduction to C++
- Object-oriented programming
- Generic programming and the STL
- Object-oriented design
C++ - Object-Oriented Programming

References
Module 1
Introduction to C++
Introduction to C++

Content
- History and evolution
- Overview of the key features
  - New built-in types
  - Scope and namespaces
  - Enumerations
  - Dynamic memory: new and delete
  - Smart pointers: unique_ptr, shared_ptr, weak_ptr
  - Error handling with exceptions
  - References
  - The const modifier
Introduction to C++

History and evolution

- Creator: Bjarne Stroustrup 1983

- Standards:
  - The first C++ standard
    - 1998 (C++98, major)
    - 2003 (C++03, minor)
  - The second C++ standard
    - 2011 (C++11, major) – significant improvements in language and library
    - 2014 (C++14, minor)
    - 2017 (C++17, major)
Introduction to C+

History and evolution

source: https://isocpp.org/std/status
Introduction to C++

History and evolution

- source: https://isocpp.org/std/status

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cfront 1.0 / TC++PL 1e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Cfront 2.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- C++98

<table>
<thead>
<tr>
<th>Year</th>
<th>1998</th>
<th>1999</th>
<th>2000</th>
<th>2001</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DRs (bugs)

<table>
<thead>
<tr>
<th>Year</th>
<th>2002</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
<th>2010</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TSes: feature branches for separate release & then merge**

- C++0x/11
  - Library TR1
  - Decimal TR (not merged)
  - Library TR2 (deferred to post-C++0x, then replaced by File System TS)
  - Math Special Functions IS

**IS: trunk**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Aug</td>
<td>Feb</td>
<td>Oct</td>
<td>Apr</td>
<td>May</td>
<td>Jun</td>
<td>Nov</td>
<td>Mar</td>
<td>Jul</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- C++14
- C++17
- C++20

- File System
- Lib Fundamentals 1
- Parallelism 1
- Concepts
- Tk Memory (not to merge)
- Arrays (abandoned)

- Networking
- Lib Fundamentals 2
- Parallelism 2
- Ranges
- Modules
- Coroutines

TS bars start and end where work on detailed specification wording starts (“adopt initial working draft”) and ends (“send to publication”). Future starts/ends are shaded to indicate that dates, and TS branches are approximate and subject to change.
Introduction to C++

Standard library
- C++ standard library = C standard library + STL (Standard Template Library)
- STL – designed by Alexander Stepanov, provides:
  - Containers: list, vector, set, map …
  - Iterators
  - Algorithms: search, sort, …
Introduction to C++

Philosophy
- Statically typed
- General purpose
- Efficient

- Supports multiple programming styles:
  - Procedural programming (*Standalone functions*)
  - Object-oriented programming (*Classes and objects*)
  - Generic programming (*Templates*)
  - Functional programming (*Lambdas*)
Introduction to C++

Portability
- Recompilation without making changes in the source code means portability.
- Hardware specific programs are usually not portable.
Creating a program

- Use a text editor to write a program and save it in a file → source code

- Compile the source code (compiler is a program that translates the source code to machine language) → object code

- Link the object code with additional code (libraries) → executable code
Introduction to C++

Creating a program (using GNU C++ compiler, Unix)

- Source code: `hello.cpp`
- Compile: `g++ -c hello.cpp`
  - Output: `hello.o` (object code)
- Compile + Link: `g++ hello.cpp`
  - Output: `a.out` (executable code)
- C++ 2014: `g++ hello.cpp -std=c++14`
The first C++ program

//hello.cpp
#include <iostream>
using namespace std;

int main(){
    cout << "Hello" << endl;
    return 0;
}

One-line comment

Preprocessor directive

The main function

I/O streams
Introduction to C++

Building a C++ program: 3 steps
- preprocessor (line starting with #)
- compiler
- linker
Introduction to C++

Most common preprocessor directives

- `#include [file]`
  - the specified file is inserted into the code

- `#define [key] [value]`
  - every occurrence of the specified key is replaced with the specified value

- `#ifndef [key] ... #endif`
  - code block is conditionally included
Introduction to C++

Header files
- C++ header
  
  ```cpp
#include <iostream>
  ```

- C header
  
  ```cpp
#include <stdio>
  ```

- User defined header
  
  ```cpp
#include "myheader.h"
  ```
Avoid multiple includes

```cpp
//myheader.h

#ifndef MYHEADER_H
#define MYHEADER_H

// the contents

#endif
```
Introduction to C++

The **main()** function
- `int main(){ ... }`

or
- `int main( int argc, char* argv[] ){ ... }

- Result status
- The number of arguments
- The arguments
I/O Streams

- **cout**: standard output
  
  ```cpp
  cout << "Hello, world!" << endl;
  ```

- **cin**: standard input

  ```cpp
  int i; double d;
  cin >> i >> d;
  ```
Namespaces
- avoid naming conflicts

//my1.h
namespace myspace1{
    void foo();
}

//my1.cpp
#include "my1.h"
namespace myspace1{
    void foo(){
        cout<<"myspace1::foo\n";
    }
}

//my2.h
namespace myspace2{
    void foo();
}

//my2.cpp
#include "my2.h"
namespace myspace2{
    void foo(){
        cout<<"myspace2::foo\n";
    }
}
Introduction to C++

Variables
- can be declared almost anywhere in your code

    double d;       //uninitialized

int i = 10;      //initialized

int j {10};      //initialized, uniform initialization
Introduction to C++

Variable types
- short, int, long – range depends on compiler, but usually 2, 4, 4 bytes
- long long (C++11) – range depends on compiler – usually 8 bytes
- float, double, long double
- bool
- char, char16_t(C++11), char32_t(C++11), wchar_t
- auto (C++11) – the compiler decides the type automatically (auto i=7;)
- decltype(expr) (C++11)

int i=10;
dcltype(i) j = 20; // j will be int
#include <iostream>
using namespace std;

int main(int argc, char** argv) {
    cout<<"short": "<<sizeof( short)"<<" bytes"<<endl;
    cout<<"int": "<<sizeof( int )"<<" bytes"<<endl;
    cout<<"long": "<<sizeof( long)"<<" bytes"<<endl;
    cout<<"long long": "<<sizeof( long long)"<<" bytes"<<endl;
    return 0;
}
Introduction to C++

C enumerations (*not type-safe*)

- always interpreted as integers →
  - you can compare enumeration values from completely different types

```cpp
enum Fruit{ apple, strawberry, melon};
enum Vegetable{ tomato, cucumber, onion};

void foo(){
    if( tomato == apple){
        cout<<"Hurra"
    }
}
```
Introduction to C++

C++ enumerations (*type-safe*)

```cpp
enum class Mark {
    Undefined, Low, Medium, High
};

Mark myMark( int value ){
    switch( value ){
    case 1: case 2: return Mark::Low;
    case 3: case 4: return Mark::Medium;
    case 5: return Mark::High;
    default:
        return Mark::Undefined;
    }
}```
Range-based for loop

```cpp
int elements[]{1, 2, 3, 4, 5};

for( auto& e : elements){
    cout<<e<<endl;
}
```
The std::array
- replacement for the standard C-style array
- cannot grow or shrink at run time

```cpp
#include <iostream>
#include <array>
using namespace std;

int main()
{
    array<int, 5> arr {10, 20, 30, 40, 50};
    cout << "Array size = " << arr.size() << endl;
    for(int i=0; i<arr.size(); ++i){
        cout<<arr[i]<<endl;
    }
}
```
Introduction to C++

Pointers and dynamic memory

- **compile time array**
  
  int ctarray[ 3 ]; //allocated on stack

- **run time array**
  
  int * rtarray = new int[ 3 ]; //allocated on heap
Introduction to C++

Dynamic memory management
- allocation
  
  ```cpp
  int * x = new int;
  int * t = new int [ 3 ];
  ```

- deletion
  
  ```cpp
  delete x;
  delete [] t;
  ```
Introduction to C++

Strings
- **C-style strings:**
  - array of characters
  - '\0' terminated
  - functions provided in `<cstring`

- C++ string
  - described in `<string`

```cpp
string firstName = "John"; string lastName = "Smith";
string name = firstName + " " + lastName; cout<<name<<endl;
```
References
- A reference defines an alternative name (alias) for an object.
- A reference must be initialized.
- Defining a reference = binding a reference to its initializer

```c++
int i = 10;
int &ri = i;  //OK    ri refers to (is another name for) i
int &ri1;    //ERROR: a reference must be initialized
```
Introduction to C++

Operations on references
- the operation is always performed on the referred object

```cpp
int i = 10;
int &ri = i;
++ri;
cout << i << endl;  // outputs 11
++i;
cout << ri << endl; // outputs 12
```
Introduction to C++

References as function parameters
- to permit *pass-by-reference*:
  - allow the function to modify the value of the parameter
  - avoid copies

```cpp
void inc(int &value)
{
    value++;
}

usage:
int x = 10;
inc( x );
```

```cpp
bool isShorter(const string &s1,
               const string &s2)
{
    return s1.size() < s2.size();
}

usage:
string str1 = "apple";
string str2 = "nut";
cout << str1 << "<" << str2 << "": " <<
isShorter(str1, str2);
```
Introduction to C++

Exceptions
- Exception = unexpected situation
- Exception handling = a mechanism for dealing with problems
  - throwing an exception – detecting an unexpected situation
  - catching an exception – taking appropriate action
```cpp
#include <iostream>
#include <stdexcept>
using namespace std;

double divide(double m, double n)
{
    if (n == 0)
    {
        throw exception();
    }
    else
    {
        return m/n;
    }
}

int main()
{
    try
    {
        cout << divide(1,0) << endl;
    }
    catch (const exception& e)
    {
        cout << "Exception was caught!" << endl;
    }
}
```
Exceptions: *domain_error*

```cpp
#include <iostream>
#include <stdexcept>
using namespace std;

double divide( double m, double n){
    if( n == 0 ){
        throw domain_error("Division by zero");
    }else{
        return m/n;
    }
}

int main() {  
    try{ 
        cout<<divide(1,0)<<endl;
    }catch( const exception& e){
        cout<<"Exception: "+e.what()<<endl;
    }
}
```
Introduction to C++

The `const` modifier
- Defining constants

```cpp
const int N = 10;
int t[ N ];
```

- Protecting a parameter

```cpp
void sayHello( const string& who){
    cout<<"Hello, "+who<<endl;
    who = "new name";
}
```

Compiler error
Uniform initialization (C++ 11)

brace-init

```cpp
int n{2};

string s{"alma"};

map<string,string> m {
    {"England","London"},
    {"Hungary","Budapest"},
    {"Romania","Bucharest"}
};

struct Person{
    string name;
    int age;
};

Person p{"John Brown", 42};
```
```cpp
#include <string>
#include <vector>
#include <iostream>
using namespace std;

int main() {
    vector<string> fruits {"apple","melon"};
    fruits.push_back("pear");
    fruits.push_back("nut");
    // Iterate over the elements in the vector and print them
    for (auto it = fruits.cbegin(); it != fruits.cend(); ++it) {
        cout << *it << endl;
    }
    // Print the elements again using C++11 range-based for loop
    for (auto& str : fruits)
        cout << str << endl;
    return 0;
}
```
Introduction to C++

Programming task:
- Write a program that reads one-word strings from the standard input, stores them and finally prints them on the standard output
- Sort the container before printing
  - use the sort algorithm

```cpp
#include <algorithm>
...
vector<string> fruits;
...
sort(fruits.begin(), fruits.end());
```
Module 2
Object-Oriented Programming
Classes and Objects
Object-Oriented Programming (OOP)

Content
- Classes and Objects
- Advanced Class Features
- Operator overloading
- Object Relationships
- Abstraction
OOP: Classes and Objects

Content
- Members of the class. Access levels. Encapsulation.
- Class: **interface** + implementation
- Constructors and **destructors**
- `const` member functions
- Constructor initializer
- Copy constructor
- Object's lifecycle
OOP: Types of Classes

Types of classes:
- **Polymorphic** Classes – *designed for extension*
  - Shape, exception, ...
- **Value** Classes – *designed for storing values*
  - int, complex<double>, ...
- **RAII** (Resource Acquisition Is Initialization) Classes –
  - (encapsulate a resource into a class → resource lifetime object lifetime)
  - thread, unique_ptr, ...

What type of resource?
OOP: Classes and objects

Class = Type ( Data + Operations)

- Members of the class

- Data:
  - data members (properties, attributes)

- Operations:
  - methods (behaviors)

- Each member is associated with an access level:
  - private
  - public
  - protected
OOP: Classes and objects

Object = Instance of a class

- An employee object: `Employee emp;`

  - **Properties** are the characteristics that describe an object.
    - *What makes this object different?*
      - id, firstName, lastName, salary, hired

  - **Behaviors** answer the question:
    - *What can we do to this object?*
      - `hire()`, `fire()`, `display()`, get and set data members
OOP: Classes and objects

Encapsulation
- an object encapsulates *data* and *functionality*.

![Diagram showing data and functionality encapsulation]

### Employee Class
- `mId`: int
- `mFirstName`: string
- `mLastName`: string
- `mSalary`: int
- `bHired`: bool
- `Employee()`: Constructor
- `display()`: void (query)
- `hire()`: void
- `fire()`: void
- `setFirstName(string)`: void
- `setLastName(string)`: void
- `setID(int)`: void
- `setSalary(int)`: void
- `getFirstName()`: string (query)
- `getLastName()`: string (query)
- `getSalary()`: int (query)
- `getIsHired()`: bool (query)
- `getId()`: int (query)
OOP: Classes and objects

Class creation
- class declaration - interface
  - Employee.h
- class definition – implementation
  - Employee.cpp
class Employee{
public:
   Employee();
   void display() const;
   void hire();
   void fire();
   // Getters and setters
   void setFirstName( string inFirstName );
   void setLastName ( string inLastName );
   void setId( int inId );
   void setSalary( int inSalary );
   string getFirstName() const;
   string getLastName() const;
   int getSalary() const;
   bool getIsHired() const;
   int getId() const;
private:
   int mId;
   string mFirstName;
   string mLastName;
   int mSalary;
   bool bHired;
};
OOP: Classes and objects

The Constructor and the object's state

- The **state of an object** is defined by its data members.
- The **constructor** is responsible for the **initial state** of the object

```cpp
Employee :: Employee() : mId(-1),
    mFirstName(""),
    mLastName(""),
    mSalary(0),
    bHired(false)
{
}
```

Members are initialized through the constructor initializer list

Members are assigned

Only constructors can use this **initializer-list** syntax!!!
OOP: Classes and objects

Constructors
- **responsibility**: data members initialization of a class object
- invoked automatically for each object
- have the *same name* as the class
- have *no return type*
- a class can have *multiple constructors* (function **overloading**)
- may not be declared as **const**
  - constructors can write to const objects
OOP: Classes and objects

Member initialization (C++11)

class C{
    string s ("abc");
    double d = 0;
    char * p {nullptr};
    int y[4] {1,2,3,4};
public:
    C(){}
};

class C{
    string s;
    double d;
    char * p;
    int y[5];
public:
    C():s("abc"),
        d(0.0),p(nullptr),
        y{1,2,3,4} {}
};
OOP: Classes and objects

Defining a member function
- Employee.cpp

- A **const** member function cannot change the object's state, can be invoked on const objects

```cpp
void Employee::hire()
{
    bHired = true;
}

string Employee::getFirstName() const
{
    return mFirstName;
}
```
Defining a member function

```cpp
void Employee::display() const {
    cout << "Employee: " << getLastName() << ", "
         << getFirstName() << endl;
    cout << "-------------------------" << endl;
    cout << (bHired ? "Current Employee" : "Former Employee") << endl;
    cout << "Employee ID: " << getId() << endl;
    cout << "Salary: " << getSalary() << endl;
    cout << endl;
}
```
OOP: Classes and objects

TestEmployee.cpp
- Using const member functions

```cpp
void foo(const Employee& e) {
    e.display();  // OK. display() is a const member function
    e.fire();     // ERROR. fire() is not a const member function
}

int main() {
    Employee emp;
    emp.setFirstName("Robert");
    emp.setLastName("Black");
    emp.setId(1);
    emp.setSalary(1000);
    emp.hire();
    emp.display();
    foo( emp );
    return 0;
}
```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
using namespace std;

class Employee {
public:
    Employee();
    //...

protected:
    int mId;
    string mFirstName;
    string mLastName;
    int mSalary;
    bool bHired;
};

#endif

#include "Employee.h"

Employee::Employee() :
    mId(-1),
    mFirstName(""),
    mLastName(""),
    mSalary(0),
    bHired(false){
}

string Employee::getFirstName() const{
    return mFirstName;
}
/ 
/ ...
OOP: Classes and objects

Object life cycles:
- creation
- assignment
- destruction
OOP: Classes and objects

Object creation:

```c
int main() {
    Employee emp;
    emp.display();

    Employee *demp = new Employee();
    demp->display();
    // ..
    delete demp;
    return 0;
}
```

- all its *embedded objects* are also created
OOP: Classes and objects

Object creation – constructors:
- **default constructor** (0-argument constructor)

```cpp
Employee :: Employee() : mId(-1), mFirstName(""), mLastName(""), mSalary(0), bHired(false)
{
}
```

- `Employee employees[10];`
- `vector<Employee> emps(10);`

- memory allocation
- constructor call on each allocated object
OOP: Classes and objects

Object creation – constructors:
- **Compiler-generated default constructor**
  
  - if a class *does not specify* any constructors, the *compiler will generate* one that does not take any arguments

```cpp
class Value{
public:
    void setValue( double inValue);
    double getValue() const;
private:
    double value;
};
```
OOP: Classes and objects

Constructors: **default** and **delete** specifiers (C++ 11)

```cpp
class X{
    int i = 4;
    int j {5};
public:
    X(int a) : i{a} {} // i = a, j = 5
    X() = default;    // i = 4, j = 5
};
```

Explicitly forcing the automatic generation of a **default** constructor by the compiler.
OOP: Classes and objects

Constructors: default and delete specifiers (C++ 11)

class X{
public:
    X( double ){}
};

X x2(3.14); //OK
X x1(10); //OK

int → double conversion

class X{
public:
    X( int )= delete;
    X( double );
};

X x1(10); //ERROR
X x2(3.14); //OK
OOP: Classes and objects

Best practice: always provide default values for members! C++ 11

```cpp
struct Point{
    int x, y;
    Point ( int x = 0, int y = 0 ): x(x), y(y){}
};
class Foo{
    int i {};
    double d {};
    char c {};
    Point p {};
public:
    void print(){
        cout <<"i: "<<i<<endl;
        cout <<"d: "<<d<<endl;
        cout <<"c: "<<c<<endl;
        cout <<"p: "<<p.x<<", "<<p.y<<endl;
    }
};

int main() {
    Foo f;
    f.print();
    return 0;
}
```

OUTPUT:
```
i: 0
d: 0
c:
p: 0, 0
```
Constructor initializer

```cpp
class ConstRef{
public:
    ConstRef( int& );
private:
    int mI;
    const int mCi;
    int& mRi;
};

ConstRef::ConstRef( int& inI ){  
mI = inI;  //OK
mCi = inI;  //ERROR: cannot assign to a const
mRi = inI;  //ERROR: uninitialized reference member
}

ConstRef::ConstRef( int& inI ): mI( inI ), mCi( inI ), mRi( inI ){}
OOP: Classes and objects

Constructor initializer
- data types that must be initialized in a `ctor-initializer`
  - `const` data members
  - reference data members
  - object data members having no default constructor
  - superclasses without default constructor
OOP: Classes and objects

A *non-default* Constructor

```cpp
Employee :: Employee( int inId, string inFirstName,
                      string inLastName,
                      int inSalary, int inHired) :
{  
    mId(inId), mFirstName(inFirstName),
    mLastName(inLastName), mSalary(inSalary),
    bHired(inHired)
}
```
class SomeType{
    int number;

public:
    SomeType(int newNumber) : number(newNumber) {}
    SomeType() : SomeType(42) {}
OOP: Classes and objects

**Copy Constructor**

Employee emp1(1, "Robert", "Black", 4000, true);

- called in one of the following cases:
  
  - Employee emp2( emp1 ); //copy-constructor called
  
  - Employee emp3 = emp2;  //copy-constructor called
  
  - void foo( Employee emp );//copy-constructor called

- if you don't define a copy-constructor explicitly, the compiler creates one for you

  - this performs a **bitwise** copy
```cpp
#ifndef STACK_H
#define STACK_H

class Stack
{
public:
    Stack( int inCapacity );
    void push( double inDouble );
    double top() const;
    void pop();
    bool isFull() const;
    bool isEmpty() const;

private:
    int mCapacity;
    double * mElements;
    double * mTop;
};
#endif /* STACK_H */
```

```cpp
//Stack.cpp
#include "Stack.h"

Stack::Stack( int inCapacity ){
    mCapacity = inCapacity;
    mElements = new double [ mCapacity ];
    mTop = mElements;
}

void Stack::push( double inDouble ){
    if( !isFull() ){
        *mTop = inDouble;
        mTop++;
    }
}
```
```cpp
#include "Stack.h"

int main(){
    Stack s1(3);
    Stack s2 = s1;
    s1.push(1);
    s2.push(2);
    cout<<"s1: "<<s1.top()<<endl;
    cout<<"s2: "<<s2.top()<<endl;
}
```
OOP: Classes and objects

Copy constructor: \( T \ (\ const \ T&) \)

```cpp
// Stack.h

#ifndef STACK_H
#define STACK_H

class Stack{
    public:
        // Copy constructor
        Stack( const Stack& );
    private:
        int mCapacity;
        double * mElements;
        double * mTop;
};
#endif /* STACK_H */

// Stack.cpp

#include "Stack.h"

Stack::Stack( const Stack& s ){
    mCapacity = s.mCapacity;
    mElements = new double[ mCapacity ];
    int nr = s.mTop - s.mElements;
    for( int i=0; i<nr; ++i ){
        mElements[ i ] = s.mElements[ i ];
    }
    mTop = mElements + nr;
}
```
```cpp
// TestStack.cpp
#include "Stack.h"

int main()
{
    Stack s1(3);
    Stack s2 = s1;
    s1.push(1);
    s2.push(2);
    cout<<"s1: "<<s1.top()<<endl;
    cout<<"s2: "<<s2.top()<<endl;
}
```
OOP: Classes and objects

Destructor
- when an object is destroyed:
  - the object's destructor is automatically invoked,
  - the memory used by the object is freed.
- each class has one destructor
- usually place to perform cleanup work for the object
- if you don't declare a destructor → the compiler will generate one, which destroys the object's member
### Destructor

**Syntax:** \( T :: \sim T() \);

```cpp
Stack::~Stack(){
    if( mElements != nullptr ){
        delete[] mElements;
        mElements = nullptr;
    }
}
```

```cpp
{ // block begin
    Stack s(10); // s: constructor
    Stack* s1 = new Stack(5); // s1: constructor
    s.push(3);
    s1->push(10);
    delete s1; // s1: destructor
    s.push(16);
} // block end // s: destructor
```
OOP: Classes and objects

Default parameters
- if the user specifies the arguments → the defaults are ignored
- if the user omits the arguments → the defaults are used
- the default parameters are specified **only** in the *method declaration* (not in the definition)

```c++
//Stack.h
class Stack{
public:
    Stack( int inCapacity = 5 );
    ..
};

//Stack.cpp
Stack::Stack( int inCapacity ){
    mCapacity = inCapacity;
    mElements = new double [ mCapacity ];
    mTop = mElements;
}
```

```c++
//TestStack.cpp
Stack s1(3);    //capacity: 3
Stack s2;       //capacity: 5
Stack s3( 10 ); //capacity: 10
```
The **this** pointer
- every method call passes a pointer to the object for which it is called as *hidden parameter* having the name **this**
- Usage:
  - for disambiguation

```cpp
Stack::Stack( int mCapacity ){
    this → mCapacity = mCapacity;
    //..
}
```
class Queue
{
    enum {Q_SIZE = 10};
private:
    // private representation to be developed later
public:
    Queue(int qs = Q_SIZE); // create queue with a qs limit
    ~Queue();
    bool isempty() const;
    bool isfull() const;
    int queuecount() const;
    bool enqueue(const Item &item); // add item to end
    bool dequeue(Item &item); // remove item from front
};
Programming task [Prata]

class Queue
{
private:
    // class scope definitions

    // Node is a nested structure definition local to this class
    struct Node { Item item; struct Node * next;};
    enum {Q_SIZE = 10};

    // private class members
    Node * front; // pointer to front of Queue
    Node * rear; // pointer to rear of Queue
    int items; // current number of items in Queue
    const int qsize; // maximum number of items in Queue
};
Module 3
Object-Oriented Programming
Advanced Class Features
OOP: Advanced class features

Content
- Inline functions
- Stack vs. Heap
- Array of objects vs. array of pointers
- Passing function arguments
- Static members
- Friend functions, friend classes
- Nested classes
- Move semantics (C++11)
OOP: Advanced class features

Inline functions
- designed to speed up programs (like macros)
- the compiler replaces the function call with the function code (no function call!)
- advantage: speed
- disadvantage: code bloat
  - ex. 10 function calls → 10 * function's size
How to make a function **inline**?
- use the `inline` keyword either in function declaration or in function definition
- both member and standalone functions can be **inline**
- common practice:
  - place the implementation of the `inline` function into the header file
- only small functions are eligible as `inline`
- the compiler may completely ignore your request
inline double square(double a) {
    return a * a;
}

class Value{
    int value;
public:
    inline int getValue() const { return value; }
    inline void setValue(int value) {
        this->value = value;
    }
};
OOP: Advanced class features

- Stack vs. Heap

- Heap – Dynamic allocation

```cpp
void draw()
{
    Point * p = new Point();
    p->move(3,3);
    //...
    delete p;
}
```

- Stack – Automatic allocation

```cpp
void draw()
{
    Point p;
    p.move(6,6);
    //...
}
```
OOP: Advanced class features

Array of objects

```cpp
class Point{
    int x, y;
    public:
        Point( int x=0, int y=0);
        //...
};
```

What is the difference between these two arrays?

Point * t1 = new Point[4];

Point t1[4];

```
<table>
<thead>
<tr>
<th></th>
<th>:Point</th>
<th>:Point</th>
<th>:Point</th>
<th>:Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0</td>
<td>x: 0</td>
<td>x: 0</td>
<td>x: 0</td>
<td></td>
</tr>
<tr>
<td>y: 0</td>
<td>y: 0</td>
<td>y: 0</td>
<td>y: 0</td>
<td></td>
</tr>
</tbody>
</table>
```
OOP: Advanced class features

Array of pointers

```cpp
Point ** t2 = new Point*[ 4 ];
for(int i=0; i<4; ++i ){
    t2[i] = new Point(0,0);
}
for( int i=0; i<4; ++i ){
    cout<<*t2[ i ]<<endl;
}
```
OOP: Advanced class features

Static members:
  • static methods
  • static data

- Functions belonging to a class scope which don't access object's data can be static
- Static methods can't be const methods (they do not access object's state)
- They are not called on specific objects ⇒ they have no this pointer
OOP: Advanced class features

- Static members

//Complex.h

class Complex{
public:
    Complex(int re=0, int im=0);
    static int getNumComplex();
    // ...
private:
    static int num_complex;
    double re, im;
};

//Complex.cpp

int Complex::num_complex = 0;

int Complex::getNumComplex(){
    return num_complex;
}

Complex::Complex(int re, int im){
    this->re = re;
    this->im = im;
    ++num_complex;
}
OOP: Advanced class features

- Static method invocation

```cpp
Complex z1(1,2), z2(2,3), z3;
cout << "Number of complexes: \n" << Complex::getNumComplex() << endl;
cout << "Number of complexes: \n" << z1.getNumComplex() << endl;
```

elegant

non-elegant
OOP: Advanced class features

Complex z1(1,2), z2(2,3), z3;

Each object has its own re and im

Only one copy of the static member

num_complex: 3
OOP: Advanced class features

- Classes vs. Structs

  - default access specifier
    - **class**: `private`
    - **struct**: `public`
  
  - **class**: data + methods, can be used polymorphically
  - **struct**: mostly data + convenience methods
OOP: Advanced class features

- Classes vs. structures

```cpp
class list{
private:
    struct node
    {
        node *next;
        int val;
        node( int val = 0, node * next = nullptr):val(val), next(next){}
    }
    node * mHead;
public:
    list();
    ~list();
    void insert (int a);
    void printAll()const;
};
```
OOP: Advanced class features

- Passing function arguments
  
  - **by value**
    - the function works on a copy of the variable
  
  - **by reference**
    - the function works on the original variable, may modify it
  
  - **by constant reference**
    - the function works on the original variable, may not modify (verified by the compiler)
OOP: Advanced class features

- Passing function arguments

```c
void f1(int x) {x = x + 1;}
void f2(int& x) {x = x + 1;}
void f3(const int& x) {x = x + 1;}// !!!!
void f4(int *x) {*x = *x + 1;}

int main()
{
    int y = 5;
    f1(y);
    f2(y);
    f3(y);
    f4(&y);
    return 0;
}
```

passing primitive values
OOP: Advanced class features

- Passing function arguments

```c
void f1(Point p);
void f2(Point& p);
void f3(const Point& p);
void f4(Point *p);
int main(){
    Point p1(3,3);
f1(p1);
f2(p1);
f3(p1);
f4(&p1);
    return 0;
}
```

- passing objects
- copy constructor will be used on the argument
- only `const` methods of the class can be invoked on this argument
OOP: Advanced class features

- **friend functions**, **friend classes**, **friend member functions**
  - friends are allowed to access private members of a class
  - Use it rarely
  - operator overloading
- friend vs. static functions

```cpp
class Test{
    private:
        int iValue;
        static int sValue;
    public:
        Test( int in ):iValue( in ){}
        void print() const;
        static void print( const Test& what );
        friend void print( const Test& what );
};
```
OOP: Advanced class features

- **friend** vs. **static** functions

```cpp
int Test :: sValue = 0;

void Test :: print() const
    { cout << "Member: " << iValue << endl; }

void Test :: print( const Test & what )
    { cout << "Static: " << what.iValue << endl; }

void print( const Test & what )
    { cout << "Friend: " << what.iValue << endl; }

int main() 
    { Test test( 10 );
      test.print();
      Test::print( test );
      print( test );
    }
```
OOP: Advanced class features

- **friend class vs. friend member function**

```cpp
class List{
private:
    ListElement * head;
public:
    bool find( int key );
    ...;
};

class ListElement{
private:
    int key;
    ListElement * next;
friend class List;
    ...
};

class ListElement{
private:
    int key;
    ListElement * next;
friend class List::find( int key);
    ...
};
```
OOP: Advanced class features

- Returning a reference to a `const` object

C++03

```cpp
// version 1
vector<int> Max(const vector<int> & v1, const vector<int> & v2)
{
    if (v1.size() > v2.size())
        return v1;
    else
        return v2;
}

// version 2
const vector<int> & Max(const vector<int> & v1, const vector<int> & v2)
{
    if (v1.size() > v2.size())
        return v1;
    else
        return v2;
}
```

The reference should be to a non-local object
OOP: Advanced class features

- Returning a reference to a `const` object

```cpp
vector<int> selectOdd( const vector<int>& v){
    vector<int> odds;
    for( int a: v ){
        if (a % 2 == 1){
            odds.push_back( a );
        }
    }
    return odds;
}
//...
vector<int> v(N);
for( int i=0; i<N; ++i){
    v.push_back( rand()% M);
}
vector<int> result = selectOdd( v );
```

EFFICIENT! MOVE constructor invocation
OOP: Advanced class features

- Nested classes
  - the class declared within another class is called a *nested class*
  - usually helper classes are declared as nested

```cpp
// Version 1

class Queue
{
  private:
    // class scope definitions
    // Node is a nested structure definition local to this class
    struct Node { Item item; struct Node * next; };
    ...
};
```
OOP: Advanced class features

- Nested classes [Prata]

```cpp
// Version 2
class Queue
{
    // class scope definitions
    // Node is a nested class definition local to this class
class Node
    {
        public:
            Item item;
            Node * next;
            Node(const Item & i) : item(i), next(0) {}
    };
};
```
OOP: Advanced class features

- Nested classes
  - a nested class B declared in a **private** section of a class A:
    - B is local to class A (only class A can use it)
  - a nested class B declared in a **protected** section of a class A:
    - B can be used both in A and in the derived classes of A
  - a nested class B declared in a **public** section of a class A:
    - B is available to the outside world (A :: B b;)

OOP: Advanced class features

- Features of a *well-behaved* C++ class
  - implicit constructor
    
    \[
    \begin{align*}
    \text{T} &:: \text{T}() \{ \ldots \} \\
    \text{T} &:: \sim\text{T}() \{ \ldots \}
    \end{align*}
    \]
  - destructor
    
    \[
    \begin{align*}
    \text{T} &:: \text{T( const T& )} \{ \ldots \}
    \end{align*}
    \]
  - copy constructor
    
    \[
    \begin{align*}
    \text{T} &:: \text{operator=} \text{( const T& )}\{ \ldots \}
    \end{align*}
    \]
  - assignment operator (*see next module*)
    
    \[
    \begin{align*}
    \text{T} &:: \text{operator=} \text{( const T& )}\{ \ldots \}
    \end{align*}
    \]
OOP: Advanced class features

- Constructor delegation (C++11)

```cpp
// C++03
class A
{
  void init() { std::cout << "init()"; }
  void doSomethingElse() { std::cout << "doSomethingElse()\n"; }
public:
  A() { init(); }
  A(int a) { init(); doSomethingElse(); }
};

// C++11
class A
{
  void doSomethingElse() { std::cout << "doSomethingElse()\n"; }
public:
  A() { ... }
  A(int a) : A() { doSomethingElse(); }
};
```
OOP: Advanced class features

- **Lvalues:**
  - Refer to objects accessible at more than one point in a source code
    - Named objects
    - Objects accessible via pointers/references
  - Lvalues may not be moved from

- **Rvalues:**
  - Refer to objects accessible at exactly one point in source code
    - Temporary objects (e.g. by value function return)
  - Rvalues may be moved from
OOP: Advanced class features

- Lvalue

```
int x;
x = 10;
```

Rvalue
OOP: Advanced class features

- Move Semantics (C++11)

```cpp
class string{
    char* data;
public:
    string( const char* );
    string( const string& );
    ~string();
};

string :: string(const char* p){
    size_t size = strlen(p) + 1;
    data = new char[size];
    memcpy(data, p, size);
}

string :: string(const string& that){
    size_t size = strlen(that.data) + 1;
    data = new char[size];
    memcpy(data, that.data, size);
}

string :: ~string(){
    delete[] data;
}
```
OOP: Advanced class features

- **Move Semantics (C++11)**: lvalue, rvalue

```cpp
string a(x);                      // Line 1
string b(x + y);                 // Line 2
string c(function_returning_a_string()); // Line 3
```

- **lvalue**: real object having an address
  
  - **Line 1**: x

- **rvalue**: temporary object – no name
  
  - **Line 2**: x + y
  
  - **Line 3**: function_returning_a_string()
OOP: Advanced class features

- Move Semantics (**C++11**): rvalue reference, move constructor

```cpp
// string&& is an **rvalue reference** to a string
string :: string(string&& that) {
    data = that.data;
    that.data = nullptr;
}
```

- **Move constructor**
- **Shallow copy** of the argument
- **Ownership transfer** to the new object
OOP: Advanced class features

- **Move** constructor – Stack class

```cpp
Stack::Stack(Stack&& rhs){
    //move rhs to this
    this->mCapacity = rhs.mCapacity;
    this->mTop = rhs.mTop;
    this->mElements = rhs.mElements;

    //leave rhs in valid state
    rhs.mElements = nullptr;
    rhs.mCapacity = 0;
    rhs.mTop = 0;
}
```
OOP: Advanced class features

- Copy constructor vs. move constructor
  - Copy constructor: **deep copy**
  - Move constructor: **shallow copy + ownership transfer**

```c++
// constructor
string s="apple";
// copy constructor: s is an lvalue
string s1 = s;
// move constructor: right side is an rvalue
string s2 = s + s1;
```
OOP: Advanced class features

- Passing large objects

```cpp
// C++98
// avoid expense copying
void makeBigVector(vector<int>& out){
    ...
}
vector<int> v;
makeBigVector( v );
```

```cpp
// C++11
// move semantics
vector<int> makeBigVector()
{
    ...
}  
auto v = makeBigVector();
```

- All STL classes have been extended to support **move semantics**
- The content of the temporary created vector is moved in `v` (not copied)
class A{
    int value {10};
    static A instance;

public:
    static A& getInstance() { return instance; }
    static A getInstanceCopy() { return instance; }
    int getValue() const { return value; }
    void setValue(int value) { this->value = value; }
};

A A::instance;
int main(){
    A& v1 = A::getInstance();
    cout<<"v1: "<<v1.getValue()<<endl;
    v1.setValue(20);
    cout<<"v1: "<<v1.getValue()<<endl;
    A v2 = A::getInstanceCopy();
    cout<<"v2: "<<v2.getValue()<<endl;
    return 0;
}
Module 4
Object-Oriented Programming
Operator overloading
OOP: Operator overloading

Content

● Objectives
● Types of operators
● Operators
  - Arithmetic operators
  - Increment/decrement
  - Inserter/extractor operators
  - Assignment operator (copy and move)
  - Index operator
  - Relational and equality operators
  - Conversion operators
OOP: Operator overloading

Objective
- To make the class usage easier, more intuitive
  - the ability to read an object using the extractor operator (>>)
    - Employee e1; cin >> e;
  - the ability to write an object using the inserter operator (<<)
    - Employee e2; cout<<e<<endl;
  - the ability to compare objects of a given class
    - cout<< ((e1 < e2) ? "less" : "greater");
OOP: Operator overloading

Limitations

- You cannot add new operator symbols. Only the existing operators can be redefined.
- Some operators cannot be overloaded:
  - . (member access in an object)
  - :: (scope resolution operator)
  - sizeof
  - ?:
- You cannot change the **arity** (the number of arguments) of the operator
- You cannot change the **precedence** or **associativity** of the operator
OOP: Operator overloading

How to implement?
- write a function with the name `operator<symbol>`
- alternatives:
  - method of your class
  - global function (usually a friend of the class)

OOP: Operator overloading

- There are 3 types of operators:
  - operators that must be methods (member functions)
    - they don't make sense outside of a class:
      - operator=, operator(), operator[], operator->
  - operators that must be **global functions**
    - the left-hand side of the operator is a variable of different type than your class:
      - `operator<<, operator>>`
      - `cout << emp;`
      - `cout`: ostream
      - `emp`: Employee
  - operators that can be **either** methods or global functions
    - **Gregoire**: “Make every operator a method unless you must make it a global function.”
OOP: Operator overloading

- Choosing argument types:
  - value vs. reference
  - const vs. non const
    - Prefer const unless you modify it.

- Choosing return types
  - you can specify any return type, however
    - follow the built-in types rule:
      - comparison always return \texttt{bool}
      - arithmetic operators return an object representing the result of the arithmetic
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex{
public:
    Complex(double, double);
    void setRe( double );
    void setIm( double im);
    double getRe() const;
    double getIm() const;
    void print() const;
private:
    double re, im;
};
#endif
#include "Complex.h"
#include <iostream>
using namespace std;

Complex::Complex(double re, double im): re(re), im(im) {} 

void Complex::setRe( double re){this->re = re;}

void Complex::setIm( double im){ this->im = im;}

double Complex::getRe() const{ return this->re;}

double Complex::getIm() const{ return this->im;}

void Complex::print() const{ cout<<re<<"+"<<im<<"i";}
OOP: Operator overloading

- Arithmetic operators (**member** or standalone func.)
  - unary minus
  - binary minus

```cpp
Complex Complex::operator-() const
{
    Complex temp(-this->re, -this->im);
    return temp;
}

Complex Complex::operator-( const Complex& z) const
{
    Complex temp(this->re - z.re, this->im - z.im);
    return temp;
}
```
OOP: Operator overloading

- Arithmetic operators (member or standalone func.)
  - unary minus
  - binary minus

```cpp
Complex operator-( const Complex& z ){
    Complex temp(-z.getRe(), -z.getIm());
    return temp;
}

Complex operator-( const Complex& z1, const Complex& z2 ){
    Complex temp(z1.getRe()-z2.getRe(), z1.getIm()-z2.getIm());
    return temp;
}
```
OOP: Operator overloading

- Increment/Decrement operators
  - postincrement:
    - int i = 10; int j = i++; // j → 10
  - preincrement:
    - int i = 10; int j = ++i; // j → 11
- The C++ standard specifies that the prefix increment and decrement return an lvalue (left value).
OOP: Operator overloading

- Increment/Decrement operators (member func.)

```cpp
Complex& Complex::operator++() { //prefix
    (this->re)++;  
    (this->im)++;  
    return *this;
}

Complex  Complex::operator++( int ){ //postfix
    Complex temp(*this);
    (this->re)++;  
    (this->im)++;  
    return temp;
}
```

Which one is more efficient? Why?
OOP: Operator overloading

- Inserter/Extractor operators (*standalone func.*)

```cpp
//complex.h

class Complex {
public:
    friend ostream& operator<<(ostream& os, const Complex& c);
    friend istream& operator>>(istream& is, Complex& c);
    //...
};
```
OOP: Operator overloading

- Inserter/Extractor operators (standalone func.)

```cpp
//complex.cpp

ostream& operator<<( ostream& os, const Complex& c){
    os<<c.re<<"+"<<c.im<<"i";
    return os;
}

istream& operator>>( istream& is, Complex& c){
    is>>c.re>>c.im;
    return is;
}
```
OOP: Operator overloading

- Inserter/Extractor operators

- Syntax:

  \[
  \text{ostream} \& \ operator<< (\ \text{ostream} \& \ os, \ \text{const} \ T \& \ \text{out})
  \]

  \[
  \text{istream} \& \ operator>>(\ \text{istream} \& \ is, \ T \& \ \text{in})
  \]

- Remarks:

  - Streams are always \textit{passed by reference}
  
  - \textbf{Q:} Why should inserter operator return an \texttt{ostream}\&?
  
  - \textbf{Q:} Why should extractor operator return an \texttt{istream}\&?
OOP: Operator overloading

- Inserter/Extractor operators

Usage:

```cpp
Complex z1, z2;
cout<<"Read 2 complex number:";
  //Extractor
cin>>z1>>z2;
  //Inserter
cout<<"z1: "<<z1<<endl;
cout<<"z2: "<<z2<<endl;

cout<<"z1++: "<<(z1++)<<endl;
cout<<"++z2: "<<(++z2)<<endl;
```
OOP: Operator overloading

- **Assignment operator (=)**
  - **Q:** When should be overloaded?
  - **A:** When bitwise copy is not satisfactory (e.g. if you have dynamically allocated memory ⇒ when we should implement the copy constructor and the destructor too).
  - Ex. our Stack class
OOP: Operator overloading

- Assignment operator (member func.)
  - Copy assignment
  - Move assignment (since C++11)
OOP: Operator overloading

- **Copy** assignment operator (**member func.**)
  - **Syntax:** `X& operator=( const X& rhs);`
  - **Q:** Is the return type necessary?
    - Analyze the following example code

```
Complex z1(1,2), z2(2,3), z3(1,1);
z3 = z1;
z2 = z1 = z3;
```
OOP: Operator overloading

- Copy assignment operator example

```cpp
Stack& Stack::operator=(const Stack& rhs) {
    if (this != &rhs) {
        //delete lhs - left hand side
        delete [] this->mElements;
        this->mCapacity = 0;
        this->_melements = nullptr; // in case next line throws
        //copy rhs - right hand side
        this->mCapacity = rhs.mCapacity;
        this->mElements = new double[mCapacity];
        int nr = rhs.mTop - rhs.mElements;
        std::copy(rhs.mElements,rhs.mElements+nr,this->mElements);
        mTop = mElements + nr;
    }
    return *this;
}
```
OOP: Operator overloading
- Copy assignment operator vs Copy constructor

```
Complex z1(1,2), z2(3,4);  // Constructor
Complex z3 = z1;            // Copy constructor
Complex z4(z2);            // Copy constructor
z1 = z2;                    // Copy assignment operator
```
OOP: Operator overloading

- **Move** assignment operator *(member func.)*
  
  - **Syntax:** `X& operator=( X&& rhs);`
  
  - **When it is called?**

```cpp
Complex z1(1,2), z2(3,4);  //Constructor
Complex z4(z2);  //Copy constructor
z1 = z2;        //Copy assignment operator
Complex z3 = z1 + z2;  //Move constructor
z3 = z1 + z1;        //Move assignment
```
OOP: Operator overloading

- **Move** assignment operator example

```cpp
Stack& Stack::operator=(Stack&& rhs){
    //delete lhs - left hand side
    delete [] this->mElements;
    //move rhs to this
    this->mCapacity = rhs.mCapacity;
    this->mTop = rhs.mTop;
    this->mElements = rhs.mElements;
    //leave rhs in valid state
    rhs.mElements = nullptr;
    rhs.mCapacity = 0;
    rhs.mTop = 0;
    //return permits s1 = s2 = create_stack(4);
    return *this;
}
```
OOP: Advanced class features

- Features of a *well-behaved* C++ class (2011)
  - implicit constructor `T :: T();`
  - destructor `T :: ~T();`
  - copy constructor `T :: T( const T& );`
  - move constructor `T :: T( T&& );`
  - copy assignment operator
    - `T& T :: operator=( const T& );`
  - move assignment operator
    - `T& T :: operator=( T&& rhs );`
OOP: Operator overloading

- Subscript operator: needed for arrays (member func.)
- Suppose you want your own dynamically allocated C-style array ⇒ implement your own `CArray`

```cpp
#ifndef CARRAY_H
#define CARRAY_H

class CArray{
public:
    CArray( int size = 10 );
    ~CArray();
    CArray( const CArray&) = delete;
    CArray& operator=( const CArray&) = delete;
    double& operator[]( int index );
    double operator[]( int index ) const;

private:
    double * mElems;
    int mSize;
};
#endif /* ARRAY_H */
```

“If the value type is known to be a built-in type, the const variant should return by value.”

OOP: Operator overloading

- Implementation

```cpp
CArray::CArray( int size ){
    if( size < 0 ){
        this->size = 10;
    }
    this->mSize = size;
    this->mElems = new double[ mSize ];
}
CArray::~CArray(){
    if( mElems != nullptr ){
        delete[] mElems;
        mElems = nullptr;
    }
}
double& CArray::operator[]( int index ){
    if( index <0 || index >= mSize ){
        throw out_of_range(""");
    }
    return mElems[ index ];
}
```

```cpp
#include<stdexcept>
double CArray::operator[]( int index ) const{
    if( index <0 || index >= mSize ){
        throw out_of_range(""");
    }
    return mElems[ index ];
}
```
OOP: Operator overloading

- **const** vs **non-const []** operator

```cpp
void printArray(const CArray& arr, size_t size) {
    for (size_t i = 0; i < size; i++) {
        cout << arr[i] << "" ;
        // Calls the const operator[] because arr is
        // a const object.
    }
    cout << endl;
}

CArray myArray;
for (size_t i = 0; i < 10; i++) {
    myArray[i] = 100;
    // Calls the non-const operator[] because
    // myArray is a non-const object.
}
printArray(myArray, 10);
```
OOP: Operator overloading

- Relational and equality operators
  - used for **search** and **sort**
  - the container must be able to compare the stored objects

```cpp
bool operator ==(const Point& p1, const Point& p2){
    return p1.getX() == p2.getX() && p1.getY() == p2.getY();
}

bool operator <(const Point& p1, const Point& p2){
    return p1.distance(Point(0,0)) < p2.distance(Point(0,0));
}
```

```cpp
set<Point> p;
vector<Point> v;  //...
sort(v.begin(), v.end());
```
OOP: Operator overloading

- The function call operator ()
- Instances of classes overloading this operator behave as functions too (they are function objects = function + object)

```cpp
#include "AddValue.h"
AddValue::AddValue( int inValue ){   
    this->value = inValue;
}
void AddValue::operator()( int& what ){   
    what += this->value;
}
```

```cpp
#ifndef ADDVALUE_H
#define ADDVALUE_H

class AddValue{
    int value;
public:
    AddValue( int inValue = 1);
    void operator()( int& what );
};
#endif /* ADDVALUE_H */
```
OOP: Operator overloading

- The function call operator

```cpp
AddValue func(2);
int array[]={1, 2, 3};
for( int& x : array ){
    func(x);
}
for( int x: array ){
    cout <<x<<endl;
}
```
OOP: Operator overloading

- Function call operator
  - used frequently for defining sorting criterion

```cpp
struct EmployeeCompare{
    bool operator()( const Employee& e1, const Employee& e2){
        if( e1.getLastName() == e2.getLastName())
            return e1.getFirstName() < e2.getFirstName();
        else
            return e1.getLastName() < e2.getLastName();
    }
};
```
OOP: Operator overloading

- Function call operator

  - sorted container

```cpp
set<Employee, EmployeeCompare> s;

Employee e1; e1.setFirstName("Barbara");
e1.setLastName("Liskov");
Employee e2; e2.setFirstName("John");
e2.setLastName("Steinbeck");
Employee e3; e3.setFirstName("Andrew");
e3.setLastName("Foyle");
s.insert(e1); s.insert(e2); s.insert(e3);

for(auto & emp : s){
    emp.display();
}
```
OOP: Operator overloading

- Sorting elements of a given type:
  - A. override operators: <, ==
  - B. define a function object containing the comparison

Which one to use?
- Q: How many sorted criteria can be defined using method A?
- Q: How many sorted criteria can be defined using method B?
OOP: Operator overloading
−

Writing conversion operators

class Complex{
public:
operator string() const;
//
};
Complex::operator string() const{
stringstream ss;
ss<<this->re<<"+"<<this->im<<"i";
return ss.str();
}

//usage

Complex z(1, 2);
string a = z;
cout<<a<<endl;


OOP: Operator overloading

- After templates
  - Overloading operator *
  - Overloading operator →
OOP: Review

- Find all possible errors or shortcomings!

```cpp
(1) class Array {
(2)    public:
(3)        Array (int n) : rep_(new int [n]) { }
(4)        Array (Array& rhs) : rep_(rhs.rep_) { }
(5)        ~Array () { delete rep_; }
(6)        Array& operator = (Array rhs) { rep_ = rhs.rep_;  }
(7)        int& operator [] (int n) { return &rep_[n]; }
(8)    private:
(9)        int * rep_;  
(10) }; // Array
```

Source: http://www.cs.helsinki.fi/u/vihavain/k13/gea/exer/exer_2.html
Solution required!

- It is given the following program!

```cpp
#include <iostream>

int main()
{
    std::cout<<"Hello\n";
    return 0;
}
```

Modify the program *without modifying the main function* so that the output of the program would be:

Start
Hello
Stop
Singleton Design Pattern

```cpp
#include <string>

class Logger{
public:
    static Logger* Instance();
    bool openLogFile(std::string logFile);
    void writeToLogFile();
    bool closeLogFile();

private:
    Logger(); // Private so that it can not be called
    Logger(Logger const&); // copy constructor is private
    Logger& operator=(Logger const&); // assignment operator is private
    static Logger* m_pInstance;
};
```

http://www.yolinux.com/TUTORIALS/C++Singleton.htm
Singleton Design Pattern

- Ensure that **only one instance** of a class is created.
- Provide a **global point of access** to the object.
Module 5
Object-Oriented Programming
Public Inheritance
OOP: Inheritance

- Inheritance
  - is-a relationship - public inheritance
  - protected access
  - virtual member function
  - early (static) binding vs. late (dynamic) binding
  - abstract base classes
  - pure virtual functions
  - virtual destructor
OOP: Inheritance

- **public inheritance**
  - *is-a* relationship
  - **base class**: Employee
  - **derived class**: Manager

- You can do with inheritance
  - **add data**
    - ex. `department`
  - **add functionality**
    - ex. `getDepartment()`, `setDepartment()`
  - **modify methods’ behavior**
    - ex. `print()`
OOP: Inheritance

- **protected access**
  - base class's *private* members can not be accessed in a derived class
  - base class's *protected* members can be accessed in a derived class
  - base class's *public* members can be accessed from anywhere
OOP: Inheritance

− public inheritance

```cpp
class Employee{
public:
    Employee(string firstName = "", string lastName = ", double salary = 0.0) :
        firstName(firstName),
        lastName(lastName),
        salary(salary) {
    }
    //...
};
```

```cpp
class Manager:public Employee{
    string department;
public:
    Manager();
    Manager( string firstName, string lastName, double salary,
            string department );
    //...
};
```
OOP: Inheritance

- Derived class's constructors

Manager::Manager()

Employee's constructor invocation → Default constructor can be invoked implicitly
OOP: Inheritance

- Derived class's constructors

Manager::Manager() {}  

Employee's constructor invocation → Default constructor can be invoked implicitly

Manager::Manager(string firstName, string lastName, double salary, 
string department): Employee(firstName, lastName, salary), 
department(department) {}  

base class's constructor invocation – constructor initializer list
arguments for the base class's constructor are specified in the definition of a derived class's constructor
OOP: Inheritance

- How are derived class's objects constructed?
  
  - *bottom up* order:
    - base class constructor invocation
    - member initialization
    - derived class's constructor block
  
  - destruction
    - in the opposite order
OOP: Inheritance

- Method overriding

```cpp
class Employee
{
public:
    virtual void print(ostream&) const;
};

class Manager: public Employee
{
public:
    virtual void print(ostream&) const;
};
```
OOP: Inheritance

- Method overriding

```cpp
class Employee {
public:
    virtual void print(ostream&) const;
};

void Employee::print(ostream& os) const {
    os << this->firstName << " " << this->lastName << " " << this->salary;
}

class Manager: public Employee {
public:
    virtual void print(ostream&) const;
};

void Manager::print(ostream& os) const {
    Employee::print(os);
    os << " " << department;
}
```
OOP: Inheritance

- Method overriding - virtual functions
  - non virtual functions are bound \textit{statically}
    - compile time
  - virtual functions are bound \textit{dynamically}
    - run time
void printAll( const vector<Employee*>& emps ){
    for( int i=0; i<emps.size(); ++i){
        emps[i]->print(cout);
        cout<<endl;
    }
}

int main(int argc, char** argv) {
    vector<Employee*> v;
    Employee e("John", "Smith", 1000);
    v.push_back(&e);
    Manager m("Sarah", "Parker", 2000, "Sales");
    v.push_back(&m);
    cout<<endl;
    printAll( v );
    return 0;
}

Output:
John Smith 1000
Sarah Parker 2000 Sales
OOP: Inheritance

- Polymorphism
  - a type with virtual functions is called a **polymorphic type**
  - polymorphic behavior **preconditions**:  
    - the member function must be **virtual** 
    - objects must be manipulated through
      - **pointers** or
      - **references**

- `Employee :: print( os )` static binding – no polymorphism
Each class with virtual functions has its own virtual function table (vtbl).
class Base{};
class Derived : public Base{};

Base* basePointer = new Derived();
Derived* derivedPointer = nullptr;

//To find whether basePointer is pointing to Derived type of object

derivedPointer = dynamic_cast<Derived*>(basePointer);
if (derivedPointer != nullptr){
    cout << "basePointer is pointing to a Derived class object";
}else{
    cout << "basePointer is NOT pointing to a Derived class object";
}
RTTI – Run-Time Type Information

dynamic_cast<>(reference)

class Base{};
class Derived : public Base{};

Derived derived;
Base& baseRef = derived;

// If the operand of a dynamic_cast to a reference isn’t of the expected type,
// a bad_cast exception is thrown.

try{
    Derived& derivedRef = dynamic_cast<Derived&>(baseRef);
    // ..
}
OOP: Inheritance

- Abstract classes
  - used for representing abstract concepts
  - used as base class for other classes
  - no instances can be created
OOP: Inheritance

- Abstract classes – **pure virtual functions**

```cpp
class Shape { // abstract class
    public:
        virtual void rotate(int) = 0; // pure virtual function
        virtual void draw() = 0; // pure virtual function
        // ...
};
```

Shape s; //???
Abstract classes – pure virtual functions

```cpp
class Shape { // abstract class
    public:
        virtual void rotate(int) = 0; // pure virtual function
        virtual void draw() = 0; // pure virtual function
        // ...
};

Shape s; // Compiler error
```
OOP: Inheritance

− Abstract class → concrete class

class Point{ /* ... */ };
class Circle : public Shape {
    public:
        void rotate(int);  // override Shape::rotate
        void draw();       // override Shape::draw
    Circle(Point p, int r) ;
    private:
        Point center;
        int radius;
};
OOP: Inheritance

- Abstract class → abstract class

```cpp
class Polygon : public Shape {
public:
    // draw() and rotate() are not overridden
};
```

Polygon p; //Compiler error
OOP: Inheritance

- **Virtual destructor**
  - Every class having at least one virtual function should have virtual destructor. *Why?*

```cpp
class X{
public:
    // ...
    virtual ~X();
};
```
Virtual destructor

```cpp
void deleteAll( Employee ** emps, int size) {
    for( int i=0; i<size; ++i) {
        delete emps[ i ];
    }
    delete [] emps;
}

// main
Employee ** t = new Employee *[ 10 ];
for(int i=0; i<10; ++i) {
    if( i % 2 == 0 )
        t[ i ] = new Employee();
    else
        t[ i ] = new Manager();
} deleteAll( t, 10);
```

Which destructor is invoked?
Module 6
Object-Oriented Programming
Object relationships
OOP: Object relationships

- The **is-a** relationship
  - Private inheritance
  - Multiple inheritance

- The **has-a** relationship
  - Association
  - Composition (strong containment)
  - Aggregation (weak containment)
OOP: Object relationships

- The is-a relationship – *Client's view (1)*
  - works in only one direction:
    - every **Sub** object is also a **Super** one
    - but **Super** object is not a **Sub**

```cpp
void foo1( const Super& s );
void foo2( const Sub& s );
Super super;
Sub sub;

foo1(super);  //OK
foo1(sub);     //OK
foo2(super);   //NOT OK
foo2(sub);     //OK
```
OOP: Object relationships

- The *is-a* relationship – *Client's view (2)*

```cpp
class Super{
public:
    virtual void method1();
};
class Sub : public Super{
public:
    virtual void method2();
};
```
OOP: Object relationships

- The *is-a* relationship – *Sub-class’s view*
  
  1. the Sub class augments the Super class by adding additional methods
  2. the Sub class may override the Super class methods
  3. the subclass can use all the public and protected members of a superclass.
OOP: Object relationships

- The *is-a* relationship: *preventing inheritance* C++11
  
  - *final* classes – cannot be extended

```cpp
class Super final
{
};
```
OOP: Object relationships

- The *is-a* relationship: a *client's view of overridden methods* (1)

  . *polymorphism*

```cpp
class Super {
    public:
        virtual void method1();
};

class Sub : public Super {
    public:
        virtual void method1();
};
```

```
Super super;
super.method1();  //Super::method1()

Sub sub;
sub.method1();    //Sub::method1()

Super& ref = super;
ref.method1();    //Super::method1();

ref = sub;
ref.method1();    //Sub::method1();

Super* ptr = & super;
ptr->method1();   //Super::method1();

ptr = & sub;
ptr->method1();   //Sub::method1();
```
OOP: Object relationships

- The is-a relationship: a client's view of overridden methods

  object slicing

```cpp
class Super{
public:
    virtual void method1();
};
class Sub : public Super{
public:
    virtual void method1();
};
```

```cpp
Sub sub;
Super super = sub;
super.method1(); // Super::method1();
```
OOP: Object relationships

- The *is-a* relationship: *preventing method overriding C++11*

```cpp
class Super{
public:
    virtual void method1() final;
};
class Sub : public Super{
public:
    virtual void method1(); //ERROR
};
```
OOP: Object relationships

- Inheritance for polymorphism
OOP: Object relationships

- The *has-a* relationship
OOP: Object relationships

- Implementing the *has-a* relationship
  
  - An object **A** has an object **B**

```cpp
#include <iostream>

class B;

class A{
private:
    B b;
};

#include <iostream>

class B;

class A{
private:
    B* b;
};

#include <iostream>

class B;

class A{
private:
    B& b;
};
```
OOP: Object relationships

- Implementing the *has-a* relationship
  - An object **A** has an object **B**
  
  *strong containment (composition)*

```cpp
class B;

class A {
  private:
    B b;
};
```

```
A anObject;

anObject: A

b: B
```
OOP: Object relationships

- Implementing the *has-a* relationship
  - An object A has an object B

  **weak containment (aggregation)**

```cpp
class B;

class A{
  private:
    B & b;
  public:
    A( const B & pb):b(pb){}
};
```

```
B bObject;
A aObject1(bObject);
A aObject2(bObject);
```
OOP: Object relationships

- Implementing the *has-a* relationship
  - An object **A** has an object **B**

**weak containment**

```cpp
class B;
class A{
   private:
      B* b;
   public:
      A( B* pb): b( pb ){};
};
```

**strong containment**

```cpp
class B;
class A{
   private:
      B* b;
   public:
      A()
         b = new B();
      ~A()
         delete b;
};
```
OOP: Object relationships

- Implementing the *has-a* relationship
  - An object **A** has an object **B**

**weak containment**

```cpp
class B;
class A{
private:
  B* b;
public:
  A( B* pb): b( pb ){};
};
```

 Usage:
- B bObject;
- A aObject1(&bObject);
- A aObject2(&bObject);

Diagram:
- A has a weak containment with B
- aObject1: A and bObject: B
- aObject2: A
OOP: Object relationships

Implementing the *has-a* relationship

- An object **A** has an object **B**

**strong containment**

```cpp
class B;

class A{
private:
    B* b;
public:
    A(){
        b = new B();
    }
    ~A(){
        delete b;
    }
};
```

Usage:
```
A aObject;
```
OOP: Object relationships

- Combining the *is-a* and the *has-a* relationships
Composite Design Pattern

- Compose objects into tree structures to represent **part-whole hierarchies**.
- Lets clients treat **individual objects** and **composition of objects uniformly**.
Composite Design Pattern

Examples:

- **Menu – MenuItem**: Menus that contain menu items, each of which could be a menu.
- **Container – Element**: Containers that contain Elements, each of which could be a Container.
- **GUI Container – GUI component**: GUI containers that contain GUI components, each of which could be a container

Source: http://www.oodesign.com/composite-pattern.html
Private Inheritance
- another possibility for *has-a* relationship

**Public Inheritance**

- Derived class **inherits** the base class behavior

**Private Inheritance**

- Derived class **hides** the base class behavior
Private Inheritance

template <typename T>
class MyStack : private vector<T> {
public:
    void push(T elem) {
        this->push_back(elem);
    }
    bool isEmpty() {
        return this->empty();
    }
    void pop() {
        if (!this->empty()) this->pop_back();
    }
    T top() {
        if (this->empty()) throw out_of_range("Stack is empty");
        else return this->back();
    }
};
Non-public Inheritance

- it is very rare;
- use it cautiously;
- most programmers are not familiar with it;
What does it print?

```cpp
class Super{
public:
    Super(){}
    virtual void someMethod(double d) const{
        cout<<"Super"<<endl;
    }
};
class Sub : public Super{
public:
    Sub(){}
    virtual void someMethod(double d){
        cout<<"Sub"<<endl;
    }
};

Sub sub; Super super;
Super& ref = sub; ref.someMethod(1);
ref = super; ref.someMethod(1);
```
What does it print?

```cpp
class Super{
public:
    Super(){} 
    virtual void someMethod(double d) const{
        cout<<"Super"<<endl;
    }
};
class Sub : public Super{
public:
    Sub(){}
    virtual void someMethod(double d){
        cout<<"Sub"<<endl;
    }
};

Sub sub; Super super;
Super& ref = sub; ref.someMethod(1);
ref = super; ref.someMethod(1);
```

creates a new method, instead of overriding the method
The *override* keyword C++11

```cpp
class Super{
public:
    Super(){}
    virtual void someMethod(double d) const{
        cout<<"Super"<<endl;
    }
};
class Sub : public Super{
public:
    Sub(){}
    virtual void someMethod(double d) const override{
        cout<<"Sub"<<endl;
    }
};

Sub sub; Super super;
Super& ref = sub; ref.someMethod(1);
ref = super; ref.someMethod(1);
```
Module 7

Generic Programming: Templates
Outline

- Templates
  - Class template
  - Function template
  - Template metaprogramming
Templates
Templates

- Allow generic programming
  
  - template programmer's obligation: specify the requirements of the classes that define these objects

  - template user's obligation: supplying those operators and methods that the template programmer requires
Function Template

- Allows writing function families

```cpp
// Function Template Example

template<typename T>
const T max(const T& x, const T& y) {
    return x < y ? y : x;
}
```

```cpp
// Function Template Example

template<class T>
const T max(const T& x, const T& y) {
    return x < y ? y : x;
}
```

- What are the requirements regarding the type T?
Function Template

```
template<class T>
const T max(const T& x, const T& y) {
    return x < y ? y : x;
}
```

- Requirements regarding the type T:
  - less operator (<)
  - copy constructor
Function Template

template<class T>
const T max(const T& x, const T& y) {
    return x < y ? y : x;
}

- cout<<max(2, 3)<<endl;  // **max**: T → int
- string a(“alma”); string b(“korte”);
- cout<<max(a, b)<<endl;  // **max**: T → string
- Person p1(“John”, “Kennedy”), p2(“Abraham”, “Lincoln”);
- cout<<max(p1, p2)<<endl;  // **max**: T→ Person
Function Template

```cpp
template<class T>
void swap(T& x, T& y) {
    const T tmp = x;
    x = y;
    y = tmp;
}
```

- Requirements regarding the type T:
  - copy constructor
  - assignment operator
Function Template

- Allows writing **function families**
  
  - **polymorphism:** *compile time*

- How the compiler processes templates?
  
  - `cout<<max(2, 3)<<endl; // max: T → int`
  - `cout<<max(2.5, 3.6)<<endl; // max: T → double`

- How many max functions?

  **Warning:** Code bloat!
Function Template

- What does it do? [Gregoire]

```c
static const size_t MAGIC = (size_t)(-1);
template <typename T>
size_t Foo(T& value, T* arr, size_t size)
{
    for (size_t i = 0; i < size; i++) {
        if (arr[i] == value) {
            return i;
        }
    }
    return MAGIC;
}
```
Class Template

- Allow writing class families

```cpp
template<typename T>
class Array {
    T* elements;
    int size;
public:
    explicit Array(const int size);
    ...
};
```
Class Template

- Template class's method definition

```cpp
template<typename T>
class Array {
    T* elements;
    int size;
public:
    explicit Array(const int size);
    ...
};

template<typename T>
Array<T>::Array(const int size):
    size(size),
    elements(new T[size]){}
```
Class Template

- Template parameters
  - type template parameters
  - non-type template parameters

```cpp
template<typename T>
class Array {
    T* elements;
    int size;
public:
    Array(const int size);
    ...;
};

template<class T, int MAX=100>
class Stack{
    T elements[MAX];
public:
    ...
};
```
Class Template

- Distributing Template Code between Files

  ● Normal class:
    - Person.h → interface
    - Person.cpp → implementation

  ● Template class:
    - interface + implementation go in the same file e.g. Array.h
      - it can be a .h file → usage: #include “Array.h”
      - it can be a .cpp file → usage: #include “Array.cpp”
Class Template+ Function Template

```cpp
#include <utility>

template<class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    pair();
    pair(const T1& x, const T2& y);
    ...
};

template<class T1, class T2>
pair<T1, T2> make_pair(const T1& x, const T2& y){
    return pair<T1, T2>(x, y);
}
```
Advanced Template

*template* template parameter

```cpp
template<typename T, typename Container>
class Stack{
    Container elements;
public:
    void push( const T& e ){
        elements.push_back( e );
    }
    ...
};
```

Usage:

```cpp
Stack<int, vector<int>> v1;
Stack<int, deque<int>> v2;
```
Advanced Template

- *template template parameter*

```cpp
template<typename T, typename Container=vector<T> >
class Stack{
    Container elements;

public:
    void push( const T& e ){
        elements.push_back( e );
    }
    ...
};
```
What does it do?

```cpp
template < typename Container >
void foo( const Container& c, const char * str="" )
{
    typename Container::const_iterator it;
    cout<<str;
    for(it = c.begin(); it != c.end(); ++it)
        cout<<*it<<' ';
    cout<<endl;
}
```
Advanced Template

What does it do?

```cpp
template < typename Container >
void foo( const Container& c, const char * str="" )
{
    typename Container::const_iterator it;
    cout<<str;
    for(auto& a: c ){
        cout<< a <<' ';
    }
    cout<<endl;
}
```
Examples

Implement the following template functions!

```
template <typename T>
bool linsearch( T* first, T* last, T what);

template <typename T>
bool binarysearch( T* first, T* last, T what);
```
More Advanced Template

Template Metaprogramming

```cpp
template<unsigned int N> struct Fact{
    static const unsigned long int value = N * Fact<N-1>::value;
};
template<> struct Fact<0>{
    static const unsigned long int value = 1;
};
// Fact<8> is computed at compile time:
const unsigned long int fact_8 = Fact<8>::value;
int main()
{
    cout << fact_8 << endl;
    return 0;
}
```
Module 8

STL – Standard Template Library
Alexander Stepanov

https://www.sgi.com/tech/stl/drdobbs-interview.htm
Outline

- Containers
- Algorithms
- Iterators
STL – General View

- library of *reusable components*
- a support for C++ development
- based on *generic programming*
STL – General View

- **Containers** – Template Class
  - generalized data structures (you can use them for any type)

- **Algorithms** – Template Function
  - generalized algorithms (you can use them for almost any data structure)

- **Iterators** – Glue between Containers and Algorithms
  - specifies a position into a container (generalized pointer)
  - permits traversal of the container
Basic STL Containers

- **Sequence containers**
  - linear arrangement
    - vector, deque, list
    - stack, queue, priority_queue

- **Associative containers**
  - provide fast retrieval of data based on keys
    - set, multiset, map, multimap

Container adapters

- `<vector>` `<deque>` `<list>`
- `<stack>` `<queue>`
- `<set>` `<map>`
Sequence Containers

- vector
- deque
- list
STL Containers C++11

- Sequence containers
  - array (C-style array)
  - forward_list (singly linked list)

- Associative containers
  - unordered_set, unordered_multiset (hash table)
  - unordered_map, unordered_multimap (hash table)
STL Containers

- homogeneous:
  - `vector<Person>, vector<Person*>`

- polymorphism
  - `vector<Person*>`

```cpp
class Person{};
class Employee: public Person{};
class Manager : public Employee{};
```
STL Containers

`vector<Person>`

<table>
<thead>
<tr>
<th>Person</th>
<th>Person</th>
<th>...</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>

homogenous
### STL Containers

#### Homogenous Containers

```
vector<Person>
```

<table>
<thead>
<tr>
<th>Person</th>
<th>Person</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

| vector<Person *>
```

<table>
<thead>
<tr>
<th>Person</th>
<th>Employee</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

#### Heterogenous Containers

- **Person**
- **Employee**
- **Manager**
The vector container - constructors

```cpp
vector<T> v; //empty vector
vector<T> v(n, value); //vector with n copies of value
vector<T> v(n); //vector with n copies of default for T
```
The *vector* container – add new elements

```cpp
vector<int> v;
for( int i=1; i<=5; ++i){
    v.push_back( i );
}
```
The vector container

```cpp
vector<int> v(10);
cout << v.size() << endl; // ???
for (int i = 0; i < v.size(); ++i) {
    cout << v[i] << endl;
}
for (int i = 0; i < 10; ++i) {
    v.push_back(i);
}
cout << v.size() << endl; // ???
for (auto& a : v) {
    cout << a << " ";
}
```
push_back vs. emplace_back

```cpp
vector<Point> v;
for( int i=0; i<10; ++i){
  v.emplace_back(i, i);
  v.emplace_back(Point(i,i));
  v.push_back(Point(i,i));
}
```
The \texttt{vector} container: typical errors

\begin{itemize}
  \item \textbf{Find the error and correct it!}
\end{itemize}

\begin{verbatim}
vector<int> v;
cout<<v.size()<<endl;//???
for( int i=0; i<10; ++i )
  v[ i ] = i;
}

cout<<v.size()<<endl;//???
for( int i=0; i<v.size(); ++i )
  cout<<v[ i ]<<endl;
\end{verbatim}
The vector container: capacity and size

```cpp
vector<int> v;
v.reserve(10);

cout << v.size() << endl;  // ???
cout << v.capacity() << endl;  // ???
```
The **vector** container: **capacity** and **size**

```cpp
vector<int> v;
v.reserve(10);

cout << v.size() << endl;//???
cout << v.capacity() << endl;//???
```

--------------------------------------

```cpp
vector<int> gy(256);
ifstream ifs("szoveg.txt"); int c;
while( (c = ifs.get()) != -1 ){
    gy[c]++;
}
```
The **vector** - indexing

```cpp
int Max = 100;
vector<int> v(Max);
//???
for (int i = 0; i < 2*Max; i++) {
    cout << v[i] << " ";
}
--------------------------------------

int Max = 100;
vector<int> v(Max);
for (int i = 0; i < 2*Max; i++) {
    cout << v.at(i) << " ";
}
```
The vector - indexing

```cpp
int Max = 100;
vector<int> v(Max);
//???...
for (int i = 0; i < 2*Max; i++) {
    cout << v[i] << " ";
}
--------------------------------------

int Max = 100;
vector<int> v(Max);
for (int i = 0; i < 2*Max; i++) {
    cout << v.at(i) << " ";
}
```

Efficient

Safe

out_of_range exception
The list container

- doubly linked list

```
list<int> l;
for (int i=1; i<=5; ++i){
    l.push_back( i );
}
```

```
. 1  . 2  . 3  . 4  . 5
```

l.begin()  l.end()
The **deque** container

- double ended vector

```cpp
deque<int> l;
for( int i=1; i<=5; ++i)
    l.push_front( i );
```
Algorithms - sort

- what to sort: [first, last)
- how to compare the elements:
  . <
  . comp

template <class RandomAccessIterator>
void sort ( RandomAccessIterator first, RandomAccessIterator last );

template <class RandomAccessIterator, class Compare>
void sort ( RandomAccessIterator first, RandomAccessIterator last, Compare comp );
struct Rec {
    string name;
    string addr;
};

vector<Rec> vr;

// …

sort(vr.begin(), vr.end(), Cmp_by_name());

sort(vr.begin(), vr.end(), Cmp_by_addr());
struct Cmp_by_name{
  bool operator()(const Rec& a, const Rec& b) const{
    return a.name < b.name;
  }
};

struct Cmp_by_addr{
  bool operator()(const Rec& a, const Rec& b) const{
    return a.addr < b.addr;
  }
};
Strategy Design Pattern

- Define a **family of algorithms**, encapsulate each one, and make them interchangeable.

- Strategy allows the *algorithm* vary independently from clients that use it.
Strategy Design Pattern

- Define a **family of algorithms**, encapsulate each one, and make them interchangeable.
- Strategy **lets the algorithm vary** independently from clients that use it.
Strategy Design Pattern

- Define a **family of algorithms**, encapsulate each one, and make them interchangeable.

- Strategy **lets the algorithm vary** independently from clients that use it.
Strategy Design Pattern

- **sort**
  - Context
    + ContextInterface()
  - Strategy
    + AlgorithmInterface()
  - ConcreteStrategyA
    + AlgorithmInterface()
  - ConcreteStrategyB
    + AlgorithmInterface()
  - ConcreteStrategyC
    + AlgorithmInterface()

- bool operator()(const T&, const T&)

- Cmp_by_name
- Cmp_by_addr
Iterators

- The *container* manages the contained objects but **does not know** about *algorithms*

- The *algorithm* works on data but **does not know** the internal structure of *containers*

- *Iterators* fit containers to algorithms
**Iterator - the glue**

```cpp
int x[]={1,2,3,4,5}; vector<int>v(x, x+5);
int sum1 = accumulate(v.begin(), v.end(), 0);

list<int> l(x, x+5);
double sum2 = accumulate(l.begin(), l.end(), 0);
```
Iterator - the glue

```cpp
template<class InIt, class T>
T accumulate(InIt first, InIt last, T init)
{
    while (first!=last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```
The `set` container

`set<Key, Comp = less<Key>>` usually implemented as a balanced binary search tree

Find the Error!

Source: http://www.cpp-tutor.de/cpp/le18/images/set.gif
The `set` container - usage

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

set<int> intSet;

set<Person> personSet1;

set<Person, PersonComp> personSet2;
```
The `set` container - usage

```cpp
#include <set>

set<int> intSet;

set<Person> personSet1;

set<Person, PersonComp> personSet2;
```
The \texttt{set} container - usage

```cpp
#include <set>

set<int> intSet;

set<Person> personSet1;

set<Person, PersonComp> personSet2;

bool operator<(const Person&, const Person&);
```
The set container - usage

```cpp
#include <set>

set<int> intSet;

set<Person> personSet1;

struct PersonComp{
    bool operator()(const Person&, const Person&);
};

set<Person, PersonComp> personSet2;
```
#include <set>

set<int> mySet;
while( cin >> nr ){
    mySet.insert( nr );
}

set<int>::iterator iter;
for (iter=mySet.begin(); iter!=mySet.end(); ++iter){
    cout << *iter << endl;
}
The \texttt{set} container - usage

```cpp
set<int>::iterator iter;
for (iter=mySet.begin(); iter!=mySet.end(); ++iter){
    cout << *iter << endl;
}
```

```
for( auto& i: mySet ){
    cout<<i<<endl;
}
```
multiset<int> mySet;
size_t nrElements = mySet.count(12);

multiset<int>::iterator iter;
iter = mySet.find(10);

if (iter == mySet.end()){
    cout<<"The element does not exist"<<endl;
}
The **multiset** container - usage

```cpp
multiset<int> mySet;
auto a = mySet.find(10);

if (a == mySet.end()){
    cout<<"The element does not exist"<<endl;
}
```
The `set` container - usage

class PersonCompare;
class Person {
    friend class PersonCompare;
    string firstName;
    string lastName;
    int yearOfBirth;
public:
    Person(string firstName, string lastName, int yearOfBirth);
    friend ostream& operator<<(ostream& os, const Person& person);
};
The **set** container - usage

class **PersonCompare** {
  public:
    enum Criterion { NAME, BIRTHYEAR};
  private:
    Criterion criterion;
  public:
    **PersonCompare**(Criterion criterion) : criterion(criterion) {}
    bool operator()(const Person& p1, const Person& p2) {
      switch (criterion) {
        case NAME: //
        case BIRTHYEAR: //
      }
    }
};
The `set` container - usage

```cpp
set<Person, PersonCompare> s( PersonCompare::NAME );
s.insert(Person("Biro", "Istvan", 1960));
s.insert(Person("Abos", "Gergely", 1986));
s.insert(Person("Gered","Attila", 1986));
----------------------------------------------------
for( auto& p : s){
    cout << p << endl;
}
```
The `map` container

`map< Key, Value[, Comp = less<Key>]>`

usually implemented as a balanced binary tree

`map`: associative array

`multimap`: allows duplicates

Source: http://www.cpp-tutor.de/cpp/le18/images/map.gif
The map container - usage

```cpp
#include <map>

map<string,int> products;

products.insert(make_pair("tomato",10));
products.insert({"onion",3});

products["cucumber"] = 6;

cout<<products["tomato"]<<endl;
```
The `map` container - usage

```cpp
#include <map>
map<string,int> products;

products.insert(make_pair("tomato",10));
products["cucumber"] = 6;

cout<<products["tomato"]<<endl;
```

Difference between `[]` and `insert`!!!
The `map` container - usage

```cpp
#include <map>
using namespace std;

int main ()
{
    map < string , int > m;
    cout << m.size () << endl; // 0
    if( m["c++"] != 0 ){
        cout << "not 0" << endl;
    }
    cout << m.size () << endl ; // 1
}
```
The \texttt{map} container - usage

typedef map<string, int>::iterator MapIt;
for (MapIt it = products.begin(); it != products.end(); ++it){
    cout << (it->first) << " : " << (it->second) << endl;
}

---

for (auto& i: products ){
    cout << (i.first) << " : " << (i.second) << endl;
}

---

for (auto& [key, value]: products ){
    cout << key << " : " << value << endl;
}
The `multimap` container - usage

```cpp
multimap<string, string> cities;
cities.insert(make_pair("HU", "Budapest"));
cities.insert(make_pair("HU", "Szeged"));
cities.insert(make_pair("RO", "Seklerburg"));
cities.insert(make_pair("RO", "Neumarkt"));
cities.insert(make_pair("RO", "Hermannstadt"));

typedef multimap<string, string>::iterator MIT;
pair/MIT, MIT> ret = cities.equal_range("HU");
for (MIT it = ret.first; it != ret.second; ++it)
    cout << (*it).first <<"\t"<<(*it).second<<endl;
```
The `multimap` container - usage

```cpp
multimap<string, string> cities;
cities.insert(make_pair("HU", "Budapest");
cities.insert(make_pair("HU", "Szeged");
cities.insert(make_pair("RO", "Seklerburg");
cities.insert(make_pair("RO", "Neumarkt");
cities.insert(make_pair("RO", "Hermannstadt");

auto ret = cities.equal_range("HU");
for (auto& [country, city]: cities){
    cout << country << "\t" << city << endl;
}
```
The `multimap` container - usage

```cpp
multimap<string, string> cities;
cities.insert(make_pair("HU", "Budapest"));
cities.insert(make_pair("HU", "Szeged"));
cities.insert(make_pair("RO", "Seklerburg"));
cities.insert(make_pair("RO", "Neumarkt"));
cities.insert(make_pair("RO", "Hermannstadt"));

auto ret = cities.equal_range("HU");
for (auto& [country, city]: cities){
    cout << country << \\
         "\t" << city << endl;
}
```

multimaps do not provide `operator[]`. Why???
The \texttt{set/map} container - removal

\begin{verbatim}
void erase ( iterator position );
size_type erase ( const key_type& x );
void erase ( iterator first, iterator last );
\end{verbatim}
The **set** – pointer key type

Output??

```cpp
set<string *> animals;
animals.insert(new string("monkey"));
animals.insert(new string("lion"));
animals.insert(new string("dog"));
animals.insert(new string("frog"));

for( auto & i: animals ){
    cout<<*i<<endl;
}
```
The *set* – pointer key type

Corrected

```cpp
struct StringComp{
    bool operator()(const string* s1,
                    const string * s2){
        return *s1 < *s2;
    }
};

set<string*, StringComp> animals;
animals.insert(new string("monkey"));
animals.insert(new string("dog"));
animals.insert(new string("frog"));

for( auto& animal: animals ){
    cout<< *animal <<endl;
}
```
Hash Tables

http://web.eecs.utk.edu/~huangj/CS302S04/notes/extendibleHashing.htm
Hash Tables

Collision resolution by chaining

Source: http://integrator-crimea.com/ddu0065.html
Unordered Associative Containers - Hash Tables

- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap
Unordered Associative Containers

− The STL standard does not specify which collision handling algorithm is required
  ● most of the current implementations use linear chaining
  ● a lookup of a key involves:
    − a hash function call $h(key)$ – calculates the index in the hash table
    − compares key with other keys in the linked list
Hash Function

- *perfect hash*: no collisions
- *lookup time*: $O(1)$ - constant
- there is a default hash function for each STL hash container
The **unordered_map** container

template <class **Key**, class **T**,
class **Hash** = hash<Key>,
class **Pred** = std::equal_to<Key>,
class **Alloc**= std::allocator<pair<const Key, T>>>
class unordered_map;

Template parameters:
- **Key** – key type
- **T** – value type
- **Hash** – hash function type
- **Pred** – equality type
The unordered_set container

\texttt{template <class Key,}
\texttt{\quad class Hash = hash<Key>,}
\texttt{\quad class Pred = std::equal_to<Key>,}
\texttt{\quad class Alloc = std::allocator<pair<const Key, T>>>}

\texttt{class unordered_set;}

Template parameters:
- \texttt{Key} – key type
- \texttt{Hash} – hash function type
- \texttt{Pred} – equality type
Problem

- Read a file containing double numbers. Eliminate the duplicates.
- Solutions???
Solutions

- `vector<double> + sort + unique`
- `set<double>`
- `unordered_set<double>`

- Which is the best? Why?
- What are the differences?
auto begin = chrono::high_resolution_clock::now();
// Code to benchmark
auto end = chrono::high_resolution_clock::now();
cout << chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count() << "ns" << endl;
## Ellapsed time

<table>
<thead>
<tr>
<th>Container</th>
<th>Time (mean)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector</td>
<td>1.38 sec</td>
</tr>
<tr>
<td>set</td>
<td>3.04 sec</td>
</tr>
<tr>
<td>unordered_set</td>
<td>1.40 sec</td>
</tr>
</tbody>
</table>
Which container to use?

- implement a PhoneBook, which:
  - stores names associated with their phone numbers;
  - names are unique;
  - one name can have multiple phone numbers associated;
  - provides $O(1)$ time search;
Which container to use?

- Usage:

```cpp
PhoneBook pbook;
pbook.addItem("kata","123456");
pbook.addItem("timi","444456");
pbook.addItem("kata","555456");
pbook.addItem("kata","333456");
pbook.addItem("timi","999456");
pbook.addItem("elod","543456");

cout<<pbook<<endl;
```
unordered_map: example

class PhoneBook {
    unordered_map<string, vector<string>> book;

public:
    void addItem(const string& name, const string& phone);

    bool removeItem(const string& name, const string& phone);

    vector<string> findItem(const string& name);

    friend ostream& operator<<(ostream& os, const PhoneBook& book);
};
void PhoneBook::addItem(const string &name, const string &phone) {
    this->book[name].push_back(phone);
}

bool PhoneBook::removeItem(const string &name, const string &phone) {
    // Locate the name → use map.at(key) + try - catch
    // If the name does not exist
    // → return false
    // Else
    // locate the given phone in the vector associated to the
    // name and delete it
    // In case of empty phone list delete the map entry too
    // → return true
}
<table>
<thead>
<tr>
<th>C++</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objects</strong></td>
<td>X x;</td>
</tr>
<tr>
<td></td>
<td>X * px = new X();</td>
</tr>
<tr>
<td><strong>Parameter passing</strong></td>
<td>void f( X x );</td>
</tr>
<tr>
<td></td>
<td>void f( X * px);</td>
</tr>
<tr>
<td></td>
<td>void f( X&amp; rx);</td>
</tr>
<tr>
<td></td>
<td>void f( const X&amp;rx);</td>
</tr>
<tr>
<td><strong>run-time binding</strong></td>
<td>only for <strong>virtual</strong> functions</td>
</tr>
<tr>
<td><strong>memory management</strong></td>
<td>explicit (2011 - <strong>smart pointers</strong>)</td>
</tr>
<tr>
<td><strong>multiple inheritance</strong></td>
<td>yes</td>
</tr>
<tr>
<td><strong>interface</strong></td>
<td>no (<strong>abstract class with pure virtual functions</strong>)</td>
</tr>
</tbody>
</table>
Algorithms
- OOP **encapsulates** data and functionality
  - data + functionality = object
- The STL separates the *data* (containers) from the *functionality* (algorithms)
  - only partial separation
Algorithms – why separation?

STL principles:
- **algorithms** and **containers** are independent
- (almost) any **algorithm** works with (almost) any **container**
- **iterators** mediate between **algorithms** and **containers**
  - provides a standard interface to traverse the elements of a container in sequence
Algorithms

Which one should be used?

```
set<int> s;
set<int>::iterator it = find(s.begin(), s.end(), 7);
if( it == s.end() ){
    //Unsuccessful
}else{
    //Successful
}
```

```
set<int> s;
set<int>::iterator it = s.find(7);
if( it == s.end() ){
    //Unsuccessful
}else{
    //Successful
}
```
Algorithms

Which one should be used?

```cpp
set<int> s;
set<int>::iterator it = find(s.begin(), s.end(), 7);
if( it == s.end() ){
    //Unsuccessful
}else{
    //Successful
}
```

```cpp
set<int> s;
set<int>::iterator it = s.find(7);
if( it == s.end() ){
    //Unsuccessful
}else{
    //Successful
}
```

$O(n)$

$O(\log n)$
Algorithm categories

- Utility algorithms
- Non-modifying algorithms
  - Search algorithms
  - Numerical Processing algorithms
  - Comparison algorithms
  - Operational algorithms
- Modifying algorithms
  - Sorting algorithms
  - Set algorithms
Utility Algorithms

- min_element()
- max_element()
- minmax_element() C++11
- swap()
Utility Algorithms

```cpp
vector<int> v = {10, 9, 7, 0, -5, 100, 56, 200, -24};

auto result = minmax_element(v.begin(), v.end());

cout << "min: " << *result.first << endl;
cout << "min position: " << (result.first - v.begin()) << endl;

cout << "max: " << *result.second << endl;
cout << "max position: " << (result.second - v.begin()) << endl;
```
Non-modifying algorithms

Search algorithms

- `find()`, `find_if()`, `find_if_not()`, `find_first_of()`
- `binary_search()`
- `lower_bound()`, `upper_bound()`, `equal_range()`
- `all_of()`, `any_of()`, `none_of()`
- ...
Non-modifying algorithms

Search algorithms - Example

```cpp
bool isEven (int i) { return ((i%2)==0); }

typedef vector<int>::iterator VIT;

int main () {
    vector<int> myvector={1,2,3,4,5};
    VIT it= find_if (myvector.begin(), myvector.end(), isEven);
    cout << "The first even value is " << *it << '
';
    return 0;
}
```
Non-modifying algorithms

Numerical Processing algorithms

- count(), count_if()
- accumulate()
- ...

Non-modifying algorithms

Numerical Processing algorithms - Example

```cpp
bool isEven(int i) { return ((i%2)==0); }

int main () {
    vector<int> myvector={1,2,3,4,5};
    int n = count_if (myvector.begin(), myvector.end(), isEven);
    cout << "myvector contains " << n << " even values.\n";
    return 0;
}
```

bool isEven(int i) { return i %2 == 0; }
Non-modifying algorithms

Comparison algorithms

- equal()
- mismatch()
- lexicographical_compare()
Non-modifying algorithms

Problem

It is given **strange alphabet** – the order of characters are unusual.

Example for a strange alphabet: \{b, c, a\}.  
Meaning: 'b'->1, c->'2', 'a' ->3

In this alphabet: "abc" >"bca"

Questions:

- How to represent the alphabet (which container and why)?
- Write a function for string comparison using the strange alphabet.
Non-modifying algorithms
Comparison algorithms - Example

// strange alphabet: 'a' ->3, 'b'->1, c->'2'
map<char, int> order;

// Compares two characters based on the strange order
bool compChar(char c1, char c2){
    return order[c1]<order[c2];
}

// Compares two strings based on the strange order
bool compString(const string& s1, const string& s2){
    return lexicographical_compare(
        s1.begin(), s1.end(), s2.begin(), s2.end(), compChar);
}
Non-modifying algorithms

Comparison algorithms - Example

// strange alphabet: 'a'->3, 'b'->1, c->'2'
map<char, int> order;

// Compares two strings based on the strange order
struct CompStr{
    bool operator()(const string& s1, const string& s2){
        return lexicographical_compare(
            s1.begin(), s1.end(), s2.begin(), s2.end(),
            [](char c1, char c2){return order[c1]<order[c2];});
    }
}

set<string, CompStr> strangeSet;
Non-modifying algorithms

- **for_each()**

```cpp
void doubleValue( int& x)
    x *= 2;
}

vector<int> v ={1,2,3};
for_each(v.begin(), v.end(), doubleValue);
```
Non-modifying algorithms

Operational algorithms

- `for_each()

```cpp
void doubleValue( int& x){
  x *= 2;
}

vector<int> v ={1,2,3};
for_each(v.begin(), v.end(), doubleValue);
```

```cpp
for_each(v.begin(), v.end(), []( int& v){ v *=2;});
```
Modifying algorithms

- copy(), copy_backward()
- move(), move_backward() C++11
- fill(), generate()
- unique(), unique_copy()
- rotate(), rotate_copy()
- next_permutation(), prev_permutation()
- nth_element() - nth smallest element
Modifying algorithms

Permutations

```cpp
void print( const vector<int>& v){
    for(auto& x: v){
        cout<<x<<"\t";
    }
    cout << endl;
}

int main(){
    vector<int> v ={1,2,3};
    print( v );
    while( next_permutation(v.begin(), v.end())){
        print( v );
    }
    return 0;
}
```
double median(vector<double>& v) {
    int n = v.size();
    if( n==0 ) throw domain_error("empty vector");
    int mid = n / 2;
    // size is an odd number
    if( n % 2 == 1 ){
        nth_element(v.begin(), v.begin()+mid, v.end());
        return v[mid];
    } else{
        nth_element(v.begin(), v.begin()+mid-1, v.end());
        double val1 = v[ mid -1 ];
        nth_element(v.begin(), v.begin()+mid, v.end());
        double val2 = v[ mid ];
        return (val1+val2)/2;
    }
}
Iterators
Outline

- Iterator Design Pattern
- Iterator Definition
- Iterator Categories
- Iterator Adapters
Iterator Design Pattern

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- The abstraction provided by the iterator pattern allows you to modify the collection implementation without making any change.
Iterator Design Pattern - Java

java.util.Collection

java.util.LinkedList

java.util.Iterator

java.util.ListIterator

Aggregate
+ CreateIterator()

ConcreteAggregate
+ CreateIterator()
  return new ConcreteIterator(this)

Iterator
+ First()
+ Next()
+ IsDone()
+ CurrentItem()
Iterator Design Pattern - C++

```
list<T> = list<T>::iterator

class Framework{
    class Iterator{
        + First()
        + Next()
        + IsDone()
        + CurrentItem()
    }
    class Aggregate{
        + CreateIterator()
    }
    class ConcreteAggregate{
        + CreateIterator()
        return new ConcreteIterator(this)
    }
```

(iterator) class iterator

list<T>::iterator
Definition

- Each container provides an iterator
- Iterator – **smart pointer** – knows *how to iterate* over the elements of that specific container
- C++ containers provides iterators a common iterator interface
Base class

template <class Category, class T, 
class Distance = ptrdiff_t, 
class Pointer = T*, 
class Reference = T&>

struct iterator {
    typedef T         value_type;
    typedef Distance  difference_type;
    typedef Pointer   pointer;
    typedef Reference reference;
    typedef Category  iterator_category;
};

does not provide any of the functionality an iterator is expected to have.
Iterator Categories

- Input Iterator
- Output Iterator
- Forward Iterator
- Bidirectional Iterator
- Random Access Iterator
Iterator Categories

- **Input Iterator**: read forward, `object=*it; it++;`
- **Output Iterator**: write forward, `*it=object; it++;`
- **Forward Iterator**: read and write forward
- **Bidirectional Iterator**: read/write forward/backward, `it++, it--;`
- **Random Access Iterator**: `it+n; it-n;`
Basic Operations

- *it: element access – get the element pointed to
- it->member: member access
- ++it, it++, --it, it--: advance forward/backward
- ==, !=: equality
Input Iterator

template<class InIt, class T>
InIt find(InIt first, InIt last, T what)
{
    for( ; first != last; ++first )
        if( *first == what ){
            return first;
        }
    return first;
}
template<class InIt, class Func>
Func for_each( InIt first, InIt last, Func f){
  for( ;first != last; ++first){
    f( *first );
  }
  return f;
}
template <class InIt, class OutIt>
OutIt copy( InIt first1, InIt last1,
           OutIt first2) {
    while ( first1 != last1 ) {
        *first2 = *first1;
        first1++;
        first2++;
    }
    return first2;
}
template < class FwdIt, class T >
void replace ( FwdIt first, FwdIt last,
             const T& oldv, const T& newv ){
    for (; first != last; ++first){
        if (*first == oldv){
            *first=newv;
        }
    }
}
template <class BiIt, class OutIt>
OutIt reverse_copy ( BiIt first, BiIt last, OutIt result)
{
    while ( first!=last ){
        --last;
        *result = *last;
        result++;
    }
    return result;
}
template <class T, class It>
It secondOccurrence(It first, It last, const T& what) {
    ???
}
template <class T, class It>
It secondOccurrence(It first, It last, const T& what) {
    while (first != last && *first != what) {
        ++first;
    }
    if (first == last) {
        return last;
    }
    ++first;
    while (first != last && *first != what) {
        ++first;
    }
    return first;
}
Containers & Iterators

- `vector` - Random Access Iterator
- `deque` - Random Access Iterator
- `list` - Bidirectional Iterator
- `set`, `map` - Bidirectional Iterator
- `unordered_set` - Forward Iterator
Iterator adapters

- Reverse iterators
- Insert iterators
- Stream iterators
Reverse iterators

- reverses the direction in which a bidirectional or random-access iterator iterates through a range.

- `++ ← → --`

- `container.rbegin()`

- `container.rend()`
Insert iterators

- special iterators designed to allow algorithms that usually overwrite elements to instead insert new elements at a specific position in the container.
- the container needs to have an insert member function
//Incorrect
int x[] = {1, 2, 3};
vector<int> v;
copy( x, x+3, v.begin());

//Correct
int x[] = {1, 2, 3};
vector<int> v;
copy( x, x+3, back_inserter(v));
Insert iterator - Example

```cpp
#include <iostream>

int main()
{
    int a[] = {1, 2, 3, 4, 5};
    int b[] = {2, 3, 4, 5, 6};
    int result = copy(a, a+5, b);
    std::cout << std::endl;
    for (int i = 0; i < result; i++)
        std::cout << b[i] << ' ';
    std::cout << std::endl;
    return 0;
}

template <class InIt, class OutIt>
OutIt copy( InIt first1, InIt last1,OutIt first2){
    while( first1 != last1){
        *first2 = *first1; // overwrite → insert
        first1++;        
        first2++;        
    }
    return first2;
}
```
## Types of insert iterators

<table>
<thead>
<tr>
<th>Type</th>
<th>Class</th>
<th>Function</th>
<th>Creation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back inserter</td>
<td>back_insert_iterator</td>
<td>push_back(value)</td>
<td>back_inserter(container)</td>
</tr>
<tr>
<td>Front inserter</td>
<td>front_insert_iterator</td>
<td>push_front(value)</td>
<td>front_inserter(container)</td>
</tr>
<tr>
<td>Inserter</td>
<td>insert_iterator</td>
<td>insert(pos, value)</td>
<td>inserter(container, pos)</td>
</tr>
</tbody>
</table>

```cpp
*pos = value;
```
Stream iterators

- Objective: connect algorithms to streams
vector<int> v;
copy(v.begin(), v.end(),
ostream_iterator<int>(cout, ",");

copy(istream_iterator<int>(cin),
    istream_iterator<int>(),
    back_inserter(v));
Problem 1.

- It is given a CArray class

```cpp
string str[] =
    {"apple", "pear", "plum", "peach", "strawberry", "banana"};

CArray<string> a(str, str+6);
```
Problem 1.

- It is given a **Smart API** too

```
Smart<string> smart;
smart.doIt( ? );
```
string str[] = {"apple", "pear", "plum", "peach", "strawberry"};
CArray<string> a (str, str+5);
CArrayIterator<string> cit ( a );
Smart<string> smart;
smart.doIt(cit);
Problem 2.

- It is given a **CArray** class

```cpp
string str[] =
    {"apple", "pear", "plum", "peach", "strawberry", "banana"};

CArray<string> a(str, str+6);
```
Problem 2.

- It is given a **Smarter API**

```cpp
class Smarter{
    public:
        template <class RaIt>
        void doIt( RaIt first, RaIt last ){
            while(  first != last ){
                cout<< *first <<std::endl;
                ++first;
            }
        }
};
```
Problem 2.

- Call the `doIt` function in the given way!

```cpp
CArray<string> a(str, str+6);
//...
Smarter smart;
smart.doIt(a.begin(), a.end());
```
Problem 2. - Solution A.

```cpp
template<class T>
class CArray{
public:
    class iterator{
        T* poz;
    public: ...
    };

    iterator begin(){ return iterator(array); }
    iterator end(){ return iterator(array+size); }

private:
    T * array;
    int size;
};
```
Problem 2. - Solution A.

class CArray{
    T* poz;
    public:
    iterator( T* poz=0 ): poz( poz ){}  
    iterator( const iterator& it ){ poz = it.poz; }
    iterator& operator=( const iterator& it ){  
        if( &it == this ) return *this;
        poz = it.poz; return *this; }
    iterator operator++(){ poz++ ; return *this; }
    iterator operator++( int p ){ 
        iterator temp( *this ); poz++ ; return temp; }
    bool operator == ( const iterator& it )const{
        return poz == it.poz; }
    bool operator != ( const iterator& it )const{
        return poz != it.poz; }
    T& operator*() const { return *poz; }
};

int size;
}
Problem 2. - Solution B.

class CArray{
public:

    typedef T * iterator;

    iterator begin(){ return array; }
    iterator end() { return array+size; }

private:
    T * array;
    int size;
};
template <class T>
class CArray{
    T * data;
    int size;
public:
    ...
    typedef T* iterator;
    typedef T value_type;
    typedef T& reference;
    typedef ptrdiff_t difference_type;
    typedef T * pointer;
};
Module 9
Function Objects & Lambdas
Function object

class FunctionObjectType {
public:
    return_type operator() (parameters) {
        Statements
    }
};
Function pointer vs. function object

- A function object may have a state
- Each function object has its own type, which can be passed to a template (e.g. set, map)
- A function object is usually faster than a function pointer
Function object as a sorting criteria

class PersonSortCriterion {
public:
    bool operator()(const Person& p1, const Person& p2) const {
        if (p1.lastname() != p2.lastname()) {
            return p1.lastname() < p2.lastname();
        } else {
            return p1.firstname() < p2.firstname();
        }
    }
};

// create a set with special sorting criterion
set<Person, PersonSortCriterion> coll;
Function object with internal state

class IntSequence{
    private:
        int value;
    public:
        IntSequence (int initialValue) : value(initialValue) {
        }
        int operator() () {
            return ++value;
        }
    };

list<int> coll;

generate_n (back_inserter(coll), // start
    9, // number of elements
    IntSequence(1)); // generates values, // starting with 1
list<int> coll;

generate_n (back_inserter(coll), // start
          9, // number of elements
         IntSequence(1)); // generates values,
          // starting with 1
class MeanValue {
private:
    long num; // number of elements
    long sum; // sum of all element values
public:
    MeanValue () : num(0), sum(0) {}  
    void operator() (int elem) {
        ++num; // increment count
        sum += elem; // add value
    }
    double value () {
        return static_cast<double>(sum) / num;
    }
};
int main()
{
    vector<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8 };

    MeanValue mv = for_each (coll.begin(), coll.end(), MeanValue());

    cout << "mean value: " << mv.value() << endl;
}
Predicates

- Are function objects that return a boolean value
- A predicate should always be stateless

```
template <typename ForwIter, typename Predicate>
ForwIter std::remove_if(ForwIter beg, ForwIter end,
                        Predicate op)
```
Predefined function objects

**Expression Effect**

- `negate<type>()` - param
- `plus<type>()` - param1 + param2
- `minus<type>()` - param1 - param2
- `multiplies<type>()` - param1 * param2
- `modulus<type>()` - param1 % param2
- `equal_to<type>()` - param1 == param2
- `not_equal_to<type>()` - param1 != param2
- `less<type>()` - param1 < param2
- `greater<type>()` - param1 > param2
- `less_equal<type>()` - param1 <= param2

**Expression Effect**

- `greater_equal<type>()` - param1 >= param2
- `logical_not<type>()` - param
- `logical_and<type>()` - param1 && param2
- `logical_or<type>()` - param1 || param2
- `bit_and<type>()` - param1 & param2
- `bit_or<type>()` - param1 | param2
- `bit_xor<type>()` - param1 ^ param2
Lambdas

- a function that you can write *inline* in your source code

```cpp
#include <iostream>

using namespace std;

int main()
{
    auto func = [] () { cout << "Hello world"; };
    func(); // now call the function
}
```
Lambdas

- **no need to write a separate function or to write a function object**

- `set`

  ```cpp
  auto comp = [](string x, string y) {
    return x > y;
  };
  set<string, decltype(comp)> s(comp);
  //...
  for (auto& x : s) {
    cout << x << endl;
  }
  ```
Lambda syntax

\[
[ \ ] \ ( ) \text{opt} \rightarrow \text{opt} \ \{} \ \{}
\]

[captures]

What outside variables are available, by value or by reference.

( params )

How to invoke it. Optional if empty.

\[-\rightarrow \text{ret} \]

Uses new syntax. Optional if zero or one return statements.

\{ \text{statements;} \}

The body of the lambda

Herb Sutter: [nwcpp.org/may-2011.html](nwcpp.org/may-2011.html)
Examples

- Earlier in scope: Widget w;
- Capture w by value, take no parameters when invoked.

  auto lamb = [w] { for( int i = 0; i < 100; ++i ) f(w); };
lamb();
- Capture w by reference, take a const int& when invoked.

  auto da = [&w] (const int& i) { return f(w, i); };
  int i = 42;
da( i );

Herb Sutter: nwcpp.org/may-2011.html
Lambdas == Functors

class __functor {

private:
CaptureTypes __captures;

public:
__functor( CaptureTypes captures )
: __captures( captures ) {};

auto operator() ( params ) { statements; }

};

Herb Sutter: nwcpp.org/may-2011.html
Capture Example

```cpp
class __functor {
    C1 __c1; C2& __c2;
    __functor( C1 c1, C2& c2 ) : __c1(c1), __c2(c2) {}
    void operator() () → {  f(__c1, __c2); }
};
```

Herb Sutter: [nwcpp.org/may-2011.html](http://nwcpp.org/may-2011.html)
class __functor {

(P1 p1, const P2 & p2) { f(p1, p2); }

public:
    void operator() ( P1 p1, const P2 & p2) {
        f(p1, p2);
    }
};

Herb Sutter: nwcpp.org/may-2011.html
Type of Lambdas

```cpp
auto g = [&]( int x, int y ) { return x > y; };
map<int, int, > m( g );
```

*Syntactic sugar*
Type of Lambdas

```cpp
auto g = [&]( int x, int y ) { return x > y; };
map<int, int, decltype(g)> m( g );
```
Example

```cpp
int x = 5;
int y = 12;
auto pos = find_if (coll.cbegin(), coll.cend(), // range
               [=](int i){return i > x && i < y;}// search criterion
               );
cout << "first elem >5 and <12: " << *pos << endl;
```

= symbols are passed by value
Example

```cpp
vector<int> vec = {1,2,3,4,5,6,7,8,9};
int value = 3;
int cnt = count_if(vec.cbegin(),vec.cend(),
                     [=](int i){return i>value;});
cout << "Found " << cnt << " values > " << value << endl;
```
Module 10
Advanced C++
Outline

- Casting. RTTI
- Handling Errors
- Smart Pointers
- Move Semantics (Move constructor, Move assignment)
- Random Numbers
- Regular Expressions
Casting & RTTI
Casting

- converting an expression of a given type into another type

- traditional type casting:
  . (new_type) expression
  . new_type (expression)

- specific casting operators:
  . `dynamic_cast <new_type> (expression)`
  . `reinterpret_cast <new_type> (expression)`
  . `static_cast <new_type> (expression)`
  . `const_cast <new_type> (expression)`
static_cast<>() vs. C-style cast

- static_cast<>() gives you a compile time checking ability, C-Style cast doesn't.
- You would better avoid casting, except dynamic_cast<>()
Run Time Type Information

- Determining the type of any variable during execution (runtime)
- Available only for **polymorphic classes** (having at least one virtual method)
- RTTI mechanism
  - the `dynamic_cast<>` operator
  - the `typeid` operator
  - the `type_info` struct
Casting Up and Down

class Super{
public:
    virtual void m1();
};
class Sub: public Super{
public:
    virtual void m1();
    void m2();
};

Sub mySub;
//Super mySuper = mySub;  // SLICE
Super& mySuper = mySub;  // No SLICE
mySuper.m1();  // calls Sub::m1() - polymorphism
mySuper.m2();  // ???
class Base{};
class Derived : public Base{};

Base* basePointer = new Derived();
Derived* derivedPointer = nullptr;

//To find whether basePointer is pointing to Derived type of object

derivedPointer = dynamic_cast<Derived*>(basePointer);
if (derivedPointer != nullptr){
    cout << "basePointer is pointing to a Derived class object";
}else{
    cout << "basePointer is NOT pointing to a Derived class object";
}
dynamic_cast<>

class Person{
    public: virtual void print(){cout<<"Person");};
};
class Employee:public Person{
    public: virtual void print(){cout<<"Employee");};
};
class Manager:public Employee{
    public: virtual void print(){cout<<"Manager");};
};

vector<Person*> v;
v.push_back(new Person());
v.push_back(new Employee());
v.push_back( new Manager());
...
class Person{
    public: virtual void print(){cout<<"Person";};
};
class Employee:public Person{
    public: virtual void print(){cout<<"Employee";};
};
class Manager:public Employee{
    public: virtual void print(){cout<<"Manager";};
};

vector<Person*> v;
v.push_back(new Person());
v.push_back(new Employee());
v.push_back( new Manager());

Write a code that counts the number of employees!
```cpp
class Person{
public: virtual void print(){cout<<"Person";};
};
class Employee:public Person{
public: virtual void print(){cout<<"Employee";};
};
class Manager:public Employee{
public: virtual void print(){cout<<"Manager";};
};
vector<Person*> v;
v.push_back(new Person());
v.push_back(new Employee());
v.push_back( new Manager());
...

Write a code that counts the number of employees!

Employee * p = nullptr;
for( Person * sz: v ){
    p = dynamic_cast<Employee *>( sz );
    if( p != nullptr ){
        ++counter;
    }
}
```
Which solution is better? (Solution 1)

```cpp
void speak(const Animal& inAnimal) {
    if (typeid (inAnimal) == typeid (Dog)) {
        cout << "VauVau" << endl;
    } else if (typeid (inAnimal) == typeid (Bird)) {
        cout << "Csirip" << endl;
    }
}

Bird bird; Dog d;
speak(bird); speak(d);
```
Which solution is better? (Solution 2)

class Animal{
    public:
        virtual void speak()=0;
    };

class Dog:public Animal{
    public:
        virtual void speak(){cout<<"VauVau"<<endl;};
    };

class Bird: public Animal{
    public:
        virtual void speak(){cout<<"Csirip"<<endl;};
    };

void speak(const Animal& inAnimal) {
    inAnimal.speak();
}

Bird bird; Dog d;
speak(bird); speak( dog );
class Person{
    public: virtual void print();
};

class Employee: public Person{
    public: virtual void print();
};

class Manager: public Employee{
    public: virtual void print();
};

vector<Person*> v;
v.push_back(new Person());
v.push_back(new Employee());
v.push_back( new Manager());
...

counter = 0;
for(Person * sz: v){
    if( typeid(*sz) == typeid(Employee) ){
        ++counter;
    }
}
```cpp
#include <iostream>
#include <typeinfo>
using namespace std;

int main (){
    int * a;
    int b;
    a=0; b=0;
    if (typeid(a) != typeid(b))
    {
        cout << "a and b are of different types:\n";
        cout << "a is: " << typeid(a).name() << '\n';
        cout << "b is: " << typeid(b).name() << '\n';
    }
    return 0;
}
```

a and b are of different types:

a is: Pi
b is: i
Handling Errors
Handling Errors

- C++ provides **Exceptions** as an *error handling mechanism*

- **Exceptions**: to handle *exceptional* but *not unexpected* situations
Return type vs. Exceptions

Return type:
- caller may ignore
- caller may not propagate upwards
- doesn't contain sufficient information

Exceptions:
- easier
- more consistent
- safer
- cannot be ignored (your program fails to catch an exception → will terminate)
- can skip levels of the call stack
int SafeDivide(int num, int den)
{
    if (den == 0)
        throw invalid_argument("Divide by zero");
    return num / den;
}

int main()
{
    try {
        cout << SafeDivide(5, 2) << endl;
        cout << SafeDivide(10, 0) << endl;
        cout << SafeDivide(3, 3) << endl;
    } catch (const invalid_argument& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0;
}
int SafeDivide(int num, int den)
{
    if (den == 0)
        throw invalid_argument("Divide by zero");
    return num / den;
}
int main()
{
    try {
        cout << SafeDivide(5, 2) << endl;
        cout << SafeDivide(10, 0) << endl;
        cout << SafeDivide(3, 3) << endl;
    } catch (const invalid_argument& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0;
}
try {
    // Code that can throw exceptions
    catch (const invalid_argument& e) {
        // Handle invalid_argument exception
    } catch (const runtime_error& e) {
        // Handle runtime_error exception
    } catch (...) {
        // Handle all other exceptions
    }
}
void func()  throw (extype1, extype2) {
    // statements
}

The throw list is not enforced at compile time!
void func() throw (){
    // statements
}

void func() noexcept{
    // statements
}
The Standard Exceptions

http://cs.stmarys.ca/~porter/csc/ref/cpp_standlib.html
User Defined Exception

- It is recommended to inherit directly or indirectly from the standard exception class

```cpp
<stdexcept>

exception

your_exception
```
User Defined Exception

```cpp
class FileError : public runtime_error
{
public:
    FileError(const string& fileIn):runtime_error (""), mFile(fileIn) {}
    virtual const char* what() const noexcept{
        return mMsg.c_str();
    }
    string getFileName() { return mFile; }
protected:
    string mFile, mMsg;
};
```
Smart Pointers
Outline

- The problem: raw pointers
- The solution: smart pointers
- Examples
- How to implement smart pointers
Why Smart Pointers?

- When to delete an object?
  - No deletion $\rightarrow$ memory leaks
  - Early deletion (others still pointing to) $\rightarrow$ dangling pointers
  - Double-freeing
Smart Pointer Types

- unique_ptr
- shared_ptr
- weak_ptr

It is recommended to use smart pointers!

#include <memory>
Smart Pointers

- Behave like built-in (raw) pointers
- Also manage dynamically created objects
  - Objects get deleted in smart pointer destructor

- Type of ownership:
  - unique
  - shared
The good old pointer

void oldPointer()
{
    Foo * myPtr = new Foo();
    myPtr->method();
}
The good Old pointer

void oldPointer1() {
    Foo * myPtr = new Foo();
    myPtr->method();
}

void oldPointer2() {
    Foo * myPtr = new Foo();
    myPtr->method();
    delete myPtr;
}

Memory leak

Could cause memory leak
When?
The Old and the New

void **oldPointer()**
{  
    Foo * myPtr = new Foo();  
    myPtr->method();  
}

void newPointer()
{  
    shared_ptr<Foo> myPtr (new Foo());  
    myPtr->method();  
}
Creating smart pointers

```cpp
void newPointer()
{
    shared_ptr<Foo> myPtr (new Foo());
    myPtr->method();
}

void newPointer()
{
    auto myPtr = make_shared<Foo>();
    myPtr->method();
}
```
- it will automatically free the resource in case of the unique_ptr goes out of scope.
Each time a `shared_ptr` is assigned
  ● a reference count is incremented (there is one more “owner” of the data)
- When a `shared_ptr` goes out of scope
  ● the reference count is decremented
  ● if `reference_count = 0` the object referenced by the pointer is freed.
Implementing your own smart pointer class

CountedPtr<Person> p(new Person("Para Peti", 1980));
Implementing your own smart pointer class

```cpp
CountedPtr<Person> p1 = p;
CountedPtr<Person> p2 = p;
```
Implementation (1)

```cpp
template < class T>
class CountedPtr{
  T * ptr;
  long * count;
public:
  ...
};
```
CountedPtr( T * p = 0 ):ptr( p ),
   count( new long(1))
{
}

CountedPtr( const CountedPtr<T>& p ): ptr( p.ptr),
   count(p.count){
   ++(*count);
}

~CountedPtr(){
   --(*count);
   if( *count == 0 ){
      delete count; delete ptr;
   }
}"
CountedPtr<T>& operator=( const CountedPtr<T>& p ){
    if( this != &p ){
        --(*count);
        if( *count == 0 ){ delete count; delete ptr; }
        this->ptr = p.ptr;
        this->count = p.count;
        ++$count;
    }
    return *this;
}

T& operator*() const{ return *ptr;}
T* operator->() const{ return ptr;}
Shared ownership with `shared_ptr`

Using C++11's Smart Pointers
Problem with `shared_ptr`

Container of smart pointers

Objects pointing to another object with a smart pointer

Using C++11's Smart Pointers
Solution: weak_ptr

Using C++11's Smart Pointers
weak_ptr

- Observe an object, but does not influence its lifetime
- Like raw pointers - the weak pointers do not keep the pointed object alive
- Unlike raw pointers – the weak pointers know about the existence of pointed-to object
How smart pointers work

Using C++11's Smart Pointers
Restrictions in using smart pointers

- Can be used to refer to objects allocated with `new` (can be deleted with `delete`).
- Avoid using raw pointer to the object referred by a smart pointer.
Inheritance and `shared_ptr`

```cpp
void greeting(shared_ptr<Person>& ptr) {
    cout << "Hello " << (ptr.get())->getFname() << " " << (ptr.get())->getLname() << endl;
}

int main(int argc, char** argv) {
    shared_ptr<Person> ptr_person(new Person("John", "Smith"));
    cout << *ptr_person << endl;
    greeting(ptr_person);

    shared_ptr<Manager> ptr_manager(new Manager("Black", "Smith", "IT"));
    cout << *ptr_manager << endl;
    ptr_person = ptr_manager;
    cout << *ptr_person << endl;
    return 0;
}
```
// p owns the Person
unique_ptr<Person> uptr(new Person("Mary", "Brown"));

unique_ptr<Person> uptr1( uptr ); //ERROR – Compile time

unique_ptr<Person> uptr2;            //OK. Empty unique_ptr

uptr2 = uptr1;                      //ERROR – Compile time
uptr2 = move( uptr );               //OK. uptr2 is the owner

cout<<"uptr2: "<<*uptr2<<endl;      //OK
cout<<"uptr: "<<*uptr<<endl;        //ERROR – Run time

unique_ptr<Person> uptr3 = make_unique<Person>("John","Dee");
cout<<*uptr3<<endl;

Static Factory Method
unique_ptr usage (2)

unique_ptr<Person> uptr1 = make_unique<Person>("Mary","Black");
unique_ptr<Person> uptr2 = make_unique<Person>("John","Dee");

vec.push_back( uptr1  );
vec.push_back( uptr2  );

foreach( auto e: vec ){
    cout<<*e<<" ";
}
cout<<"\n";
unique_ptr<Person> uptr1 = make_unique<Person>("Mary","Black");
unique_ptr<Person> uptr2 = make_unique<Person>("John","Dee");
cout<<*uptr2<<endl;

vector<unique_ptr<Person>> vec;
vec.push_back(move(uptr1));
vec.push_back(move(uptr2));
cout<<"Vec [";
for(auto& e: vec){
    cout<<*e<<" ";
}
cout<<"]"<<endl;
Module 11
I/O Streams
Outline

- Using Streams
- String Streams
- File Streams
- Bidirectional I/O
Using Streams

- file
- keypad
- program

Input Stream

Program

Output Stream

- file
- screen
- program

Stream:
- is data flow
- direction
- associated source and destination
Using Streams

**cin**  An input stream, reads data from the “input console.”

**cout** A *buffered* output stream, writes data to the output console.

**cerr** An *unbuffered* output stream, writes data to the “error console”

**clog** A buffered version of cerr.
Using Streams

- Stream:
  
  ● includes **data**
  
  ● has a **current position**
    
    - *next read or next write*
Using Streams

ios_base

basic_ios<>
ios, wios

basic_istream<>
istream, wistream

basic_ostream<>
ostream, wostream

basic_streambuf<>
streambuf, wstreambuf
Using Streams

- Output stream:
  - inserter operator `<<`
  - raw output methods (binary):
    - put(), write()

```cpp
void rawWrite(const char* data, int dataSize){
    cout.write(data, dataSize);
}

void rawPutChar(const char* data, int charIndex){
    cout.put(data[charIndex]);
}
```
Using Streams

- Output stream:
  - most output streams buffer data (accumulate)
  - the stream will *flush* (write out the accumulated data) when:
    - an endline marker is reached (\n, endl)
    - the stream is destroyed (e.g. goes out of scope)
    - the stream buffer is full
    - explicitly called `flush()`
Using Streams

- Manipulators:
  - objects that modify the behavior of the stream
    - setw, setprecision
    - hex, oct, dec
    - C++11: put_money, put_time

```c
int i = 123;
printf("This should be ' 123': %6d\n", i);
cout <<"This should be ' 123': " << setw(6) << i << endl;
```
Using Streams

- Input stream:
  - extractor operator `>>`
    - will tokenize values according to white spaces
  - raw input methods (binary):
    - `get()`: avoids tokenization

```cpp
string readName(istream& inStream){
    string name;
    char next;
    while (inStream.get(next)) {
        name += next;
    }
    return name;
}
```
Using Streams

- Input stream:
  - `getline()`: reads until end of line

```cpp
string myString;
getline(cin, myString);
```
Using Streams

- Input stream:
  - `getline()`: reads until end of line

```cpp
string myString;
getline(cin, myString);
```

reads an input having more than one word

Reads up to new line character
- Unix line ending: `\n`
- Windows line ending: `\r` `\n`

The problem is that `getline` leaves the `\r` on the end of the string.
Using Streams

- Stream's state:
  - every stream is an object → has a state
  - stream's states:
    - good: OK
    - eof: End of File
    - fail: Error, last I/O failed
    - bad: Fatal Error
Using Streams

- Find the error!

```
list<int> a;
int x;
while( !cin.eof() ) {
    cin>>x;
    a.push_back( x );
}
```

Input:
1
2
3
(empty line)

a: 1, 2, 3, 3
Using Streams

- Handling Input Errors:

  - while( cin )
  - while( cin >> ch )

```c
int number, sum = 0;
while ( true ) {
    cin >> number;
    if (cin.good()){
        sum += number;
    } else{
        break;
    }
}
```
String Streams

- `<sstream>`
  - `ostringstream`
  - `istringstream`
  - `stringstream`

```cpp
string s = "12.34";
stringstream ss(s);
double d;
ss >> d;
```

```cpp
double d = 12.34;
stringstream ss;
ss << d;
string s = "szam:" + ss.str();
```
File Streams

```cpp
{  
    ifstream ifs("in.txt");  // Constructor
    if( !ifs ){
        // File open error
    }
    // Destructor call will close the stream
}

{  
    ifstream ifs;
    ifs.open("in.txt");
    // ...
    ifs.close();
    // ...
}
File Streams

- Byte I/O

```cpp
ifstream ifs("dictionary.txt");
// ios::trunc means that the output file will be
// overwritten if exists
ofstream ofs("dict.copy", ios::trunc);

char c;
while( ifs.get( c ) ){
    ofs.put( c );
}
```
File Streams

- Byte I/O
- Using rdbuf() - quicker

```cpp
#include <fstream>

ifstream ifs("dictionary.txt");
// ios::trunc means that the output file will be
// overwritten if exists
ofstream ofs("dict.copy", ios::trunc);

if (ifs && ofs) {
    ofs << ifs.rdbuf();
}
```
Object I/O

- Operator overloading

```cpp
istream& operator>>( istream& is, T& v )
{
    //read v
    return is;
}

ostream& operator<<(ostream& is, const T& v )
{
    //write v
    return os;
}
```
Module 12
Concurrency
Outline

- High-level interface: `async()` and `future`
- Low-level interface: `thread`, `promise`
- Synchronizing threads
- Mutexes and locks: `mutex`, `lock_guard`, `unique_lock`
- Atomics
Find all words matching a pattern in a dictionary!

**Pattern:** a..l.

**Word:** apple, apply, ...
Single-threaded Solution (1)

```cpp
string pattern = "a..l."
;
// Load the words into the deque
ifstream f("dobbsdict.txt");
if ( !f ) {
    cerr << "Cannot open dobbsdict.txt in the current directory\n";
    return 1;
}
string word;
deque<string> backlog;
while ( f >> word ){
    backlog.push_back( word );
}

// Now process the words and print the results
vector<string> words = find_matches(pattern, backlog);
cerr << "Found " << words.size()<< " matches for " << pattern<< endl;
for ( auto s : words ){
    cout << s << "\n";
}
```
Single-threaded Solution (2)

```cpp
inline bool match( const string &pattern, string word )
{
    if ( pattern.size() != word.size() )
        return false;
    for ( size_t i = 0 ; i < pattern.size() ; i++ )
        if ( pattern[ i ] != '.' && pattern[ i ] != word[ i ] )
            return false;
    return true;
}

vector<string> find_matches( string pattern, deque<string> &backlog )
{
    vector<string> results;
    for ( ; ; )
    {
        if ( backlog.size() == 0 ) { return results; }
        string word = backlog.front();
        backlog.pop_front();
        if ( match( pattern, word ) ) { results.push_back( word );}
    }
    return results;
}
```
Multi-threaded Solution (1)

```cpp
string pattern = "a..l.";
// Load the words into the deque
ifstream f( "dobbsdict.txt" );
if ( !f ) {
    cerr << "Cannot open sowpods.txt in the current directory\n";
    return 1;
}
string word;
deque<string> backlog;
while ( f >> word ){ backlog.push_back( word );}
// Now process the words and print the results
auto f1 = async( launch::async, find_matches, pattern, ref(backlog) );
auto f2 = async( launch::async, find_matches, pattern, ref(backlog) );
auto f3 = async( launch::async, find_matches, pattern, ref(backlog) );
print_results( f1, pattern, 1 );
print_results( f2, pattern, 2 );
print_results( f3, pattern, 3 );
```

Worker thread
Returns a `std::future` object
Multi-threaded Solution (1)

```cpp
string pattern = "a..l.";
// Load the words into the deque
ifstream f("dobbsdct.txt");
if (!f) {
    cerr << "Cannot open sowpods.txt in the current directory\n";
    return 1;
}
string word;
dequ<string> backlog;
while (f >> word) { backlog.push_back(word); }
// Now process the words and print the results
auto f1 = async(launch::async, find_matches, pattern, ref(backlog));
auto f2 = async(launch::async, find_matches, pattern, ref(backlog));
auto f3 = async(launch::async, find_matches, pattern, ref(backlog));

print_results(f1, pattern, 1);
print_results(f2, pattern, 2);
print_results(f3, pattern, 3);
```

parameter as a reference
Multi-threaded Solution (2)

template<class ASYNC>
void print_results( ASYNC &f, string &pattern, int threadno )
{
    vector<string> words = f.get();
    cerr << "Found " << words.size()<< " matches for " << pattern
         << " in thread " << threadno<< endl;
    for ( auto s : words ){ cout << s << "\n";}
}

std::future<>::get()
-returns the return value of the async function
-blocks until the thread is complete
Multi-threaded Solution (3)

```cpp
std::mutex m;

vector<string> find_matches( string pattern, deque<string> &backlog )
{
    vector<string> results;
    for ( ; ; ) {
        m.lock();
        if ( backlog.size() == 0 ) {
            m.unlock();
            return results;
        }
        string word = backlog.front();
        backlog.pop_front();
        m.unlock();
        if ( match( pattern, word ) )
            results.push_back( word );
    }
}
```
Performance

*Multi-threaded* vs. *Single-threaded* solution!!!
Futures

Objectives
− makes easy to get the computed *result back* from a thread,
− able to transport an *uncaught exception* to another thread.

2. When a function has calculated the return value
3. Put the value in a **promise** object
4. The value can be retrieved through a **future**
Futures

- if the other thread has not yet finished the call to get() will block
- avoid blocking:

```cpp
future<T> fut = ...// launch a thread or async
T result = fut.get();
if( fut.wait_for( 0 ) ){
    T result = fut.get();
} else{
    ...
}
```
```c
int val;
mutex valMutex;
valMutex.lock();
if (val >= 0) {
    f(val);
} else {
    f(-val);
} valMutex.unlock();
```

**mutex** = mutual exclusion

Helps to control the concurrent access of a resource
What happens in case of an exception?
mutex vs. lock.guard<mutex>

```cpp
int val;
mutex valMutex;
valMutex.lock();
if (val >= 0) {
    f(val);
}
else {
    f(-val);
}
valMutex.unlock();
```

```cpp
int val;
mutex valMutex;
lock.guard<mutex> lg(valMutex);
if (val >= 0) {
    f(val);
}
else {
    f(-val);
}
```

RAII principle (*Resource Acquisition Is Initialization*)
lock_guard<mutex>

```cpp
int val;
mutex valMutex;
{
    lock_guard<mutex> lg(valMutex);
    if (val >= 0) {
        f(val);
    }
    else {
        f(-val);
    }
}
```

RAII principle (Resource Acquisition Is Initialization)

**Constructor:** acquires the resource

**Destructor:** releases the resource

Destructor is always called even in case of an exception!!!
unique_lock<mutex>

unique_lock = lock_guard + lock() & unlock()
Multithreaded Logger [Gregoire]

class Logger {
    public:
        Logger();
        void log(const string& entry);
    
    protected:
        void processEntries();
        mutex mMutex;
        condition_variable mCondVar;
        queue<string> mQueue;
        thread mThread;  // The background thread.
    
    private:
        // Prevent copy construction and assignment.
        Logger(const Logger& src);
        Logger& operator=(const Logger& rhs);
};
Multithreaded Logger [Gregoire]

Logger::Logger() {
    // Start background thread.
    mThread = thread{&Logger::processEntries, this};
}

void Logger::log(const std::string& entry) {
    // Lock mutex and add entry to the queue.
    unique_lock<mutex> lock(mMutex);
    mQueue.push(entry);
    // Notify condition variable to wake up thread.
    mCondVar.notify_all();
}
Multithreaded Logger

```cpp
void Logger::processEntries()
{
    ofstream ofs("log.txt");
    if (ofs.fail()) { ... return; }
    unique_lock<mutex> lock(mMutex);
    while (true) {
        // Wait for a notification.
        mCondVar.wait(lock);
        // Condition variable is notified → something is in the queue.
        lock.unlock();
        while (true) {
            lock.lock();
            if (mQueue.empty()) {
                break;
            } else {
                ofs << mQueue.front() << endl;
                mQueue.pop();
            }
            lock.unlock();
        }
    }
}
```
void logSomeMessages(int id, Logger& logger)
{
    for (int i = 0; i < 10; ++i) {
        stringstream ss;
        ss << "Log entry " << i << " from thread " << id;
        logger.log(ss.str());
    }
}

int main()
{
    Logger logger;
    vector<thread> threads;
    // Create a few threads all working with the same Logger instance.
    for (int i = 0; i < 10; ++i) {
        threads.push_back(thread(logSomeMessages, i, ref(logger)));
    }
    // Wait for all threads to finish.
    for (auto& t : threads) {
        t.join();
    }
    return 0;
}
Problem: Multithreaded Logger [Gregoire]

end of main() → terminate abruptly **Logger** thread
Solution: Multithreaded Logger [Gregoire]

class Logger
{
public:
    Logger();
    ~Logger();
    // Gracefully shut down background thread.
    void log(const std::string& entry);

protected:
    void processEntries();
    bool mExit;
    ...
};
Solution: Multithreaded Logger [Gregoire]

```cpp
void Logger::processEntries()
{
    ...  
    while (true) {
        // Wait for a notification.
        mCondVar.wait(lock);
        // Condition variable is notified, so something is in the queue
        // and/or we need to shut down this thread.
        lock.unlock();
        while (true) {
            lock.lock();
            if (!mQueue.empty()) {
                break;
            } else {
                ofs << mQueue.front() << endl;
                mQueue.pop();
            }
            lock.unlock();
        }
        if (mExit) break;
    }
    
} 
```
Solution: Multithreaded Logger

```cpp
Logger::Logger() : mExit(false)
{
    // Start background thread.
    mThread = thread{&Logger::processEntries, this};
}

Logger::~Logger()
{
    // Gracefully shut down the thread by setting mExit to true and notifying the thread.
    mExit = true;
    // Notify condition variable to wake up thread.
    mCondVar.notify_all();
    // Wait until thread is shut down.
    mThread.join();
}
```
Solution: Multithreaded Logger [Gregoire]

```cpp
Logger::Logger() : mExit(false)
{
    // Start background thread.
    mThread = thread{&Logger::processEntries, this};
}
Logger::~Logger()
{
    // Gracefully shut down the thread by setting mExit to true and notifying the thread.
    mExit = true;
    // Notify condition variable to wake up thread.
    mCondVar.notify_all();
    // Wait until thread is shut down.
    mThread.join();
}
```
Solution: Multithreaded Logger [Gregoire]

Deadlock
It can happen that this remaining code from the main() function, including the Logger destructor, is executed before the Logger background thread has started its processing loop. When that happens, the Logger destructor will already have called notify_all() before the background thread is waiting for the notification, and thus the background thread will miss this notification from the destructor.
Object Pool

Thread Pool

ObjectPool
- resources: Collection
- maxResources: int
- rFactory: Factory
- aquireObject()
- releaseObject()

Factory
createResource()

Resource

1:1
1:*
Object Pool

C++ implementation [Gregoire]

template <typename T>
class ObjectPool{
public:
  ObjectPool(size_t chunkSize = kDefaultChunkSize)
    throw(std::invalid_argument, std::bad_alloc);
  
  shared_ptr<T> acquireObject();
  void releaseObject(shared_ptr<T> obj);

protected:
  queue<shared_ptr<T>> mFreeList;
  size_t mChunkSize;
  static const size_t kDefaultChunkSize = 10;
  void allocateChunk();

private:
  // Prevent assignment and pass-by-value
  ObjectPool(const ObjectPool<T>& src);
  ObjectPool<T>& operator=(const ObjectPool<T>& rhs);
};
Object Pool

C++ implementation [Gregoire]

```cpp
template <typename T>
ObjectPool<T>::ObjectPool(size_t chunkSize)
    throw(std::invalid_argument,
         std::bad_alloc)
{
    if (chunkSize == 0) {
        throw std::invalid_argument("chunk size must be positive");
    }
    mChunkSize = chunkSize;
    allocateChunk();
}
```
Object Pool

C++ implementation [Gregoire]

```cpp
template <typename T>
void ObjectPool<T>::allocateChunk()
{
    for (size_t i = 0; i < mChunkSize; ++i) {
        mFreeList.push(std::make_shared<T>());
    }
}
```
template <typename T>  
shared_ptr<T> ObjectPool<T>::acquireObject()  
{  
    if (mFreeList.empty()) {  
        allocateChunk();  
    }  
    auto obj = mFreeList.front();  
    mFreeList.pop();  
    return obj;  
}
Object Pool

C++ implementation [Gregoire]

```cpp
template <typename T>
void ObjectPool<T>::releaseObject(shared_ptr<T> obj)
{
    mFreeList.push(obj);
}
```