המחלקה למדעי המחשב

89-210 תכנות מתקדם 1

מרצה: ד"ר אליהו חלסצ'י

6.1 Template Functions

DR. ELIAHU KHALASTCHI

What functions do we need to make it work?

```
int max(int x, int y){
    return (x >= y) ? x : y;
}

char max(char x, char y){
    return (x >= y) ? x : y;
}

double max(double x, double y){
    return (x >= y) ? x : y;
}

int main() {
    cout << "max(1,2)=="      << max(1, 2)       << endl;
    cout << "max(2,1)=="      << max(2, 1)       << endl;
    cout << "max('a','z')=="  << max('a', 'z')   << endl;
    cout << "max(3.14,2.73)=="<< max(3.14, 2.73) << endl;
    return 0;
}
```

The same template of code, different types…

Do we really need to repeat the same code??

We can use templates!

Source code

```
template <class T>
const T& max(const T& x, const T& y) {

    return (x>=y) ? x : y;
}
int main() {
    cout << "max(1,2)=="   << max(1, 2)  << endl;
    cout << "max(2,1)=="   << max(2, 1)  << endl;
    cout << "max('a','z')=="   << max('a', 'z')   << endl;
    cout << "max(3.14,2.73)==" << max(3.14, 2.73) << endl;
    return 0;
}
```

We can use templates!

Source code

```
template <class T>
const T& max(const T& x, const T& y) {

    return (x>=y) ? x : y;
}
int main() {
    cout << "max(1,2)=="    << max(1, 2)  << endl;
    cout << "max(2,1)=="    << max(2, 1)  << endl;
    cout << "max('a','z')=="    << max('a', 'z')   << endl;
    cout << "max(3.14,2.73)=="  << max(3.14, 2.73) << endl;
    return 0;
}
```

T should be
double

Compiled code

```
const int& max(const int& x,const int& y){
   return (x >= y) ? x : y;
}

const char& max(const char& x, const char& y){
   return (x >= y) ? x : y;
}

const double& max(const double& x, const double& y){
   return (x >= y) ? x : y;
}

int main() {
      ...
}
```

In compilation time…

- The compiler must be able to **deduce** the type of T

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

```
int main() {
    max(5, 3);
}
```

T should be int

```
template <class T>
void func(int i){
    T a, b; //...
}
```

```
int main() {
    func(10);
}
```

error C2783: 'void func(int)' : could not deduce template argument for 'T'

- Otherwise, we should tell it!

```
int main() {
 func<float>(10);
 func<int>(10);
 func<char>(10);
}
```

Specialization…

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

const char* max(const char* a,const char* b) {
    if (strcmp(a, b) > 0)
        return a;
    else
        return b;
}

int main() {
    cout << max(1, 2)  << endl;
    cout << max("hello", "world") << endl;
    return 0;
}
```

Restrictions…

```
template <class C> void printIfEqual(const C& a, const C& b){
   if (a == b){
      cout << a << " and " << b << " are equal" << endl;
   }
}
```

What do we expect of type C?

Restrictions…

```
template <class C> void printIfEqual(const C& a, const C& b){
   if (a == b){
      cout << a << " and " << b << " are equal" << endl;
   }
}
```

What do we expect of type C?

```
class Student{
   char* name;
   int age;
   public:
   bool operator==(const Student& s)const{
      return (age == s.age && strcmp(name, s.name) == 0);
   }
   friend ostream& operator<<(ostream& out, const Student& s);
};
ostream& operator<<(ostream& out, const Student& s){
   out << "name: " << s.name << endl;
   out << "age: " << s.age << endl;
   return out;
}
```

```
int main(){
   Student a, b;
   //...
   printIfEqual(a, b);
}
```

המחלקה למדעי המחשב

89-210 תכנות מתקדם 1

מרצה: ד"ר אליהו חלסצ'י

6.2 Generic Algorithm

TEMPLATES VS. POLYMORPHIC (VIRTUAL) METHODS

DR. ELIAHU KHALASTCHI

Polymorphic Generic Algorithm

```
Worker_Age_Sorter s1;
s1.sort(workers); // sort by age

Worker_Salary_Sorter s2;
s2.sort(workers); // sort by salary
```

Class Worker_Sorter

```
void sort(Worker** workers){
  …
  if(compare(workers[i],workers[i+1])<0)
  …
}
virtual int compare(Wroker* w1, Worker* w2)=0;
```

This is the generic algorithm.
It appears only once in the code

Class Worker_Age_Sorter

```
virtual int compare(Wroker* w1, Worker* w2){
  return w1->getAge() – w2->getAge();
}
```

Class Worker_Salary_Sorter

```
virtual int compare(Wroker* w1, Worker* w2){
  return w1->getSalary() – w2->getSalary();
}
```

But, it has 2 different implementations

Template (Generic) Algorithm

```
// the comparable needs to implement the "<=" operator
template <class Comparable>
void sort(Comparable** comparables){
   //...
   if (*comparables[i+1] <= *comparables[i]){
    Comparable* tmp = comparables[i];
    comparables[i] = comparables[i + 1];
    comparables[i + 1] = tmp;
   }
   //...
}
```

```
// the complied version...
void sort(Student** comparables){
   //...
   if (*comparables[i+1] <= *comparables[i]){
     Student* tmp = comparables[i];
     comparables[i] = comparables[i + 1];
     comparables[i + 1] = tmp;
   }
   //...
}
```

```
class Student{
   char* name;
   int age;
   public:
   bool operator<=(const Student& s)const{
      return (age <= s.age &&
                strcmp(name, s.name) <= 0);
   }
};
```

```
Student** students = new Student*[n];
//...
sort(students);
```

Template alg’ vs. polymorphic alg’

- If the algorithm applies to **several families** of classes – use a **template**
- If it applies only to **one family** of classes – use **polymorphism**
- **Code size:**
  - Polymorphism – code exists only in the base class. Overhead of some classes. Not inflating the compiled code
  - Template – code exists only once. Inflating the compiled code
- **Runtime:**
  - Polymorphism – slower (dynamic function binding – 2 memory calls)
  - Template – faster (static function binding)

המחלקה למדעי המחשב

89-210 תכנות מתקדם 1

מרצה: ד"ר אליהו חלסצ'י

6.3 Template Classes

DR. ELIAHU KHALASTCHI

Template Classes

```
class A{
   int x;
   public:
   A(int ax){ x = ax; }
};
```

```
template <class T> class A{
   T x;
   public:
   A(const T& ax){ x = ax; }
};
```

```
void main() {
    A<int> a(0);
    A<double> b(0.5);
    A<Student> c(Student("Eli"));
}
```

Array Class Example

```
template<class T>
class Array {
   T* m_arr;
   int m_size;

   public:
   Array(int size) : m_size(size) { m_arr=new T[size]; }
   Array(const Array& a);
   ~Array() { delete[] m_arr; }
   const Array& operator=(const Array& a);

   T& operator[](int index) { return m_arr[index]; }
   void Print() const;
};
```

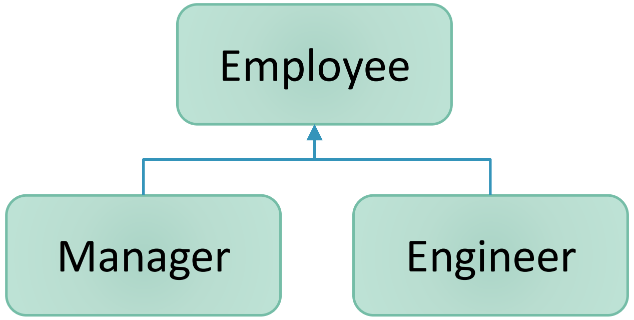
```
template<class T> void Array<T>::Print() const
{
   for (int i = 0; i < m_size; i++)
      cout << "Array[" << i << "]:" << m_arr[i] << endl;
}
```

```
void main() {

   Array<int> iArray(10);
   Array<double>  dArray(10);

   for (int i = 0; i < 10; i++) {
      iArray[i] = i;
      dArray[i] = i / 3.0;
   }

   iArray.Print();
   dArray.Print();
}
```

General Container

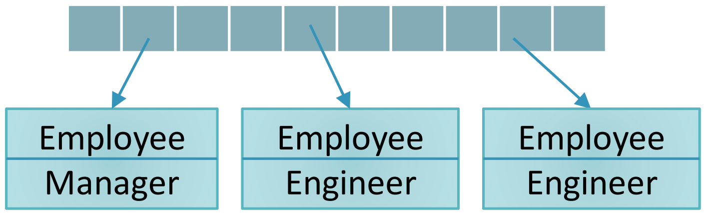
Template classes vs. Polymorphism

```
template<class T>
class Array {
   T* m_arr;
   int m_size;
   //...
};

Array<Student> sArray(10);
Array<double>  dArray(10);
Array<string>  strArray(10);
```



General Container:

```
Employee** employees;
```



We can combine! `Array<Employee*> eArray(10);`

Inheriting Template Classes

- By defining the type
- By being an undefined template class

```
template <class T> class A{
    T x;
    public:
    A(const T& ax){ x = ax; }
};
```

```
class B : A<int>{
    public:
    B(int x) :A(x){}
};
```

```
B b(0);
```

```
template <class T> class A{
    T x;
    public:
    A(const T& ax){ x = ax; }
};
```

```
template <class T> class B : A<T>{
public:
    B(const T& x) :A(x){}
};
```

```
B<int> b(0);
```

המחלקה למדעי המחשב

89-210 תכנות מתקדם 1

מרצה: ד"ר אליהו חלסצ'י

6.4 Object Functions

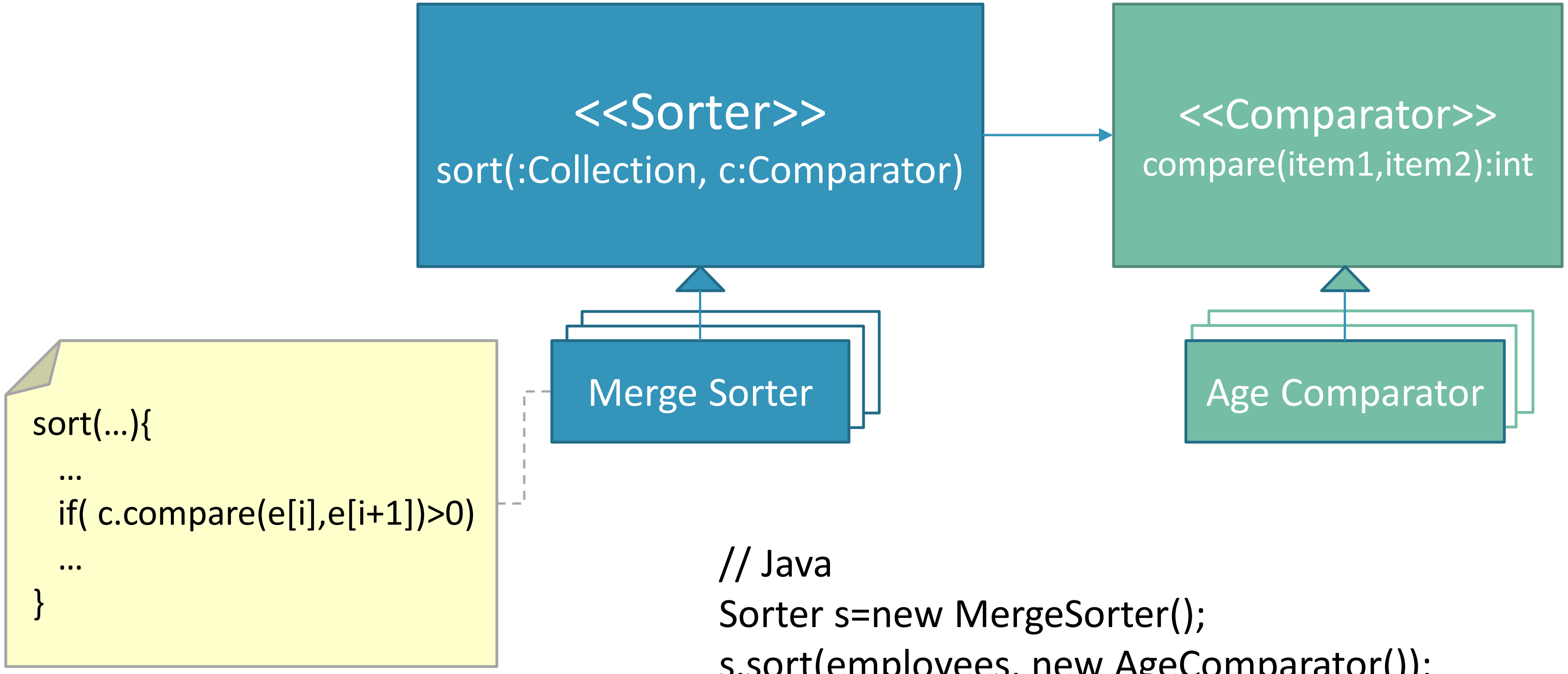
A.K.A FUNCTION OBJECTS, FUNCTORS

DR. ELIAHU KHALASTCHI

Object Functions

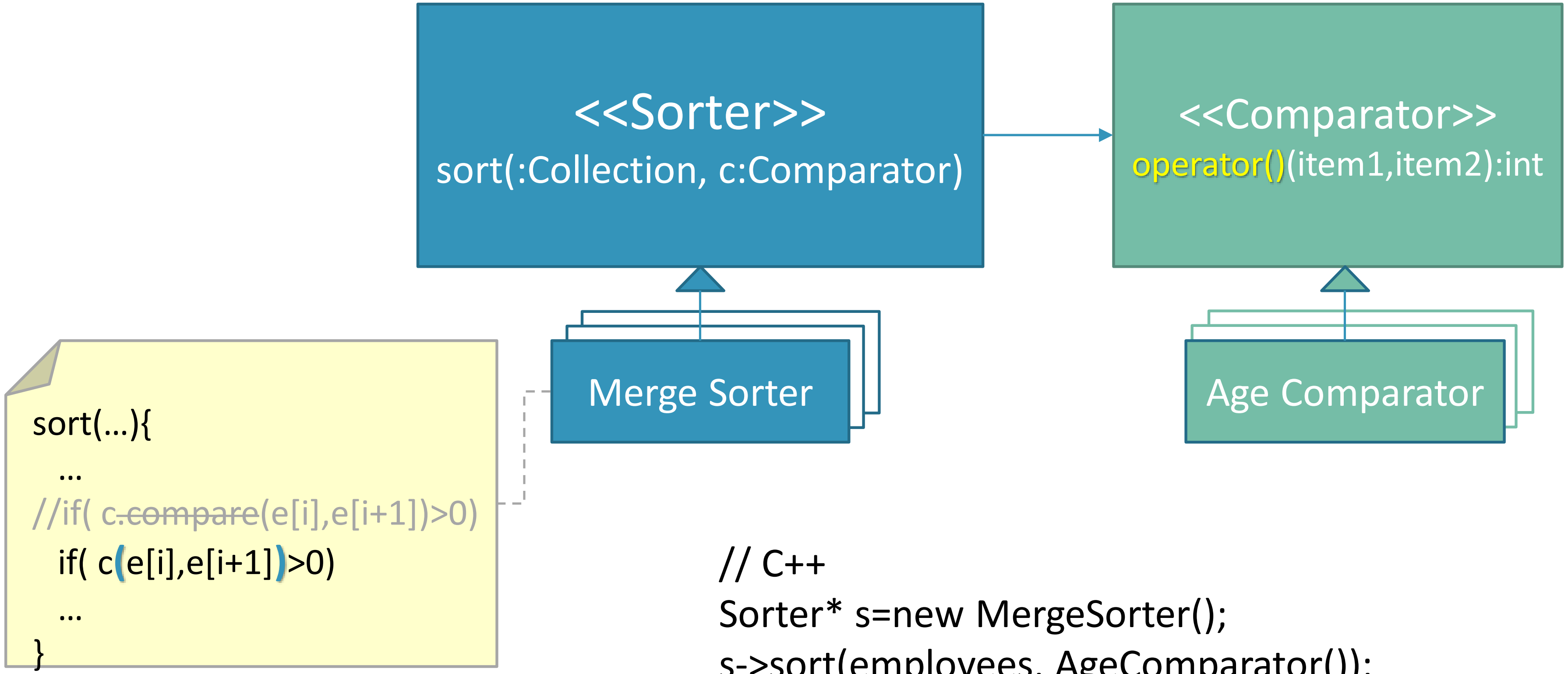
- We want to pass a *function* as a *parameter* to another function
- In the OOP way… (so we need to pass an object)
- The answer: Object Functions!
  - A struct
  - With a template method – operator()

From Strategy Pattern to Object Function



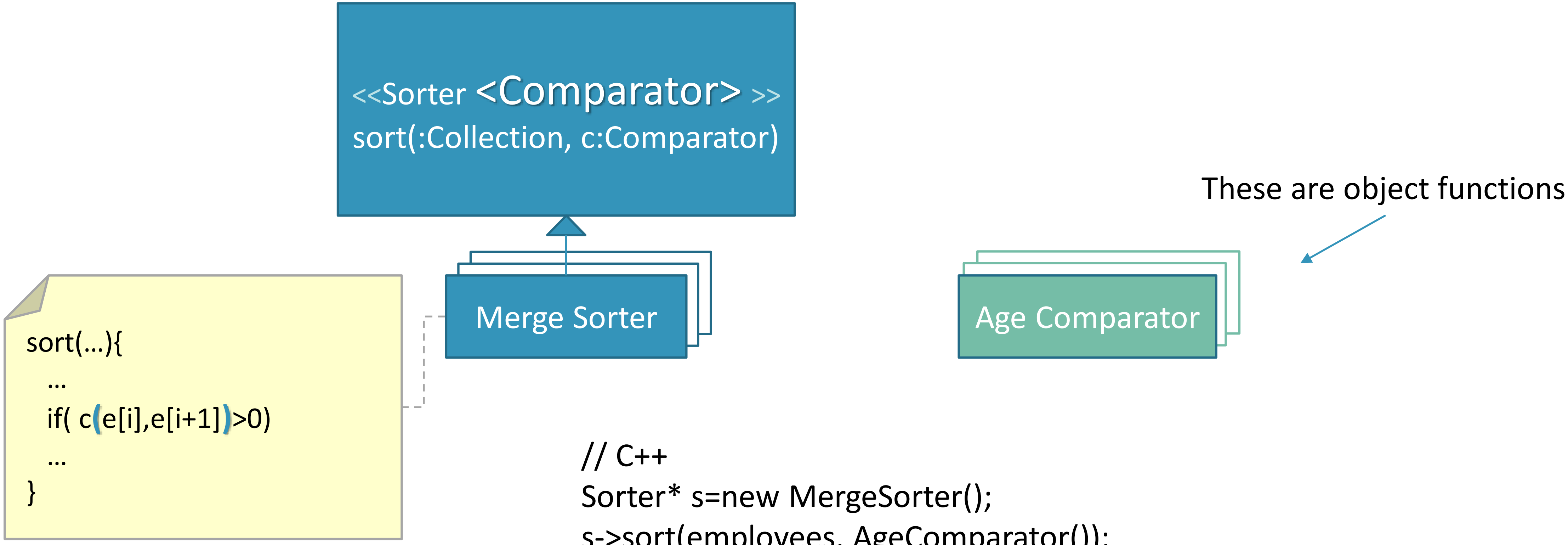

```
// Java
Sorter s=new MergeSorter();
s.sort(employees, new AgeComparator());
```

From Strategy Pattern to Object Function




```
// C++
Sorter* s=new MergeSorter();
s->sort(employees, AgeComparator());
```

From Strategy Pattern to Object Function




```
// C++
Sorter* s=new MergeSorter();
s->sort(employees, AgeComparator());
```

objects vs. object functions vs. functions

```
class Student{
  float grade;
  public:
  void setGrade();
  float getGrade();
};
```

```
struct Sqr{ // this is an object function
   template<class T>
   void operator()(T& number) const {
      number = number * number;
   }
};
```

```
bool func(int x){
  double y;
   …
   return true;
}
```

- Stateful nature
- Methods use data members

- Wraps a function (in *operator()*)
- Stateless nature – preferable
  - But not a must…
- Can be passed as a parameter
  - Since it’s an object

- Stateless nature
- Gets parameters
- Has local variables
- Returns a value

Examples of Object Functions

```
struct Sqr{
   template<class T>
   void operator()(T& number) const {
      number = number * number;
   }
};
```

```
struct Print {
   template<class T>
   void operator()(T& printable) const{
      cout << printable << endl;
   }
};
```

A general function:

```
template <class T, class func>
void applyOnArray(T* array, int size, const func& f){
   for (int i = 0; i < size; i++)
      f(array[i]);
}
```

We assume f has the () operator,
which can be applied to every type T

```
int array[] = { 3, 2, 5, 7, 2, 8, 11 };
Sqr s;
applyOnArray(array, 7, s);
applyOnArray(array, 7, Sqr());
applyOnArray(array, 7, Print());
```

<u>Exercise:</u> build a general function
that can work on other data structures

A truly generic function

```
template <class Iterator, class func>
void apply(Iterator begin, Iterator end, const func& f){
    for (; begin != end; begin++)
        f(*begin);
}
```

It is not dependent on
- the data structure
- the function it applies

A generic container

A generic algorithm

```
LinkedList<int> iList;
for (int i = 1; i < 10; i++)
    iList.insert(iList.end(), i);

apply(iList.begin(), iList.end(), Print());
```

The data-structure's iterators

An object function

אוניברסיטת בר-אילן

המחלקה למדעי המחשב

89-210 תכנות מתקדם 1

מרצה: ד"ר אליהו חלסצ'י

6.5 Sorting example

TEMPLATE VS. TEMPLATE AND OBJECT FUNCTION

DR. ELIAHU KHALASTCHI

Template (Generic) Algorithm

```
// the comparable needs to implement the "<=" operator
template <class Comparable>
void sort(Comparable** comparables){
   //...
   if (*comparables[i+1] < *comparables[i]){
    Comparable* tmp = comparables[i];
    comparables[i] = comparables[i + 1];
    comparables[i + 1] = tmp;
   }
   //...
}
```

```
class Student{
   char* name;
   int age;
   public:
   bool operator<(const Student& s)const{
      return (age < s.age);
   }
};
```

```
Student** students = new Student*[n];
//...
sort(students);
```

We can only implement the operator< once!
How can we sort with by different fields of Student?
(without changing the code of the class)

Template Algorithm + Object Function

```
template <class T,class Comparator>
void sort(T** array, const Comparator& comp){
    //...
    if (comp(*array[i], *array[i + 1]) < 0){//...}
    //...
}
```

```
struct NameComparator{
public:
  int operator()(const Student& s1, const Student& s2){
     return strcmp(s1.getName(), s2.getName());
  }
};
```

```
struct AgeComparator{
public:
  int operator()(const Student& s1, const Student& s2){
     return s1.getAge()-s2.getAge();
  }
};
```

```
class Student{
   char* name;
   int age;
public:
   int getAge()const{ return age; }
   const char* getName()const { return name; }
};
```

```
Student** students = new Student*[n];
//...
// sort by name
sort(students, NameComparator() );
// sort by age
sort(students, AgeComparator() );
```

sort() is much more generic now!

המחלקה למדעי המחשב

89-210 תכנות מתקדם 1

מרצה: ד"ר אליהו חלסצ'י

6.6 STL – Standard Template Library

THE STL IS FILLED WITH GENERIC CONTAINERS, GENERIC FUNCTIONS, AND OBJECT FUNCTIONS…

DR. ELIAHU KHALASTCHI

STL - introduction

- STL contains generic data structures like the LinkedList we have built
- Now, you are allowed to use the things STL really includes
- In general STL includes the following components:
  - **Data Structures** – (vector, list, set, …)
  - **Generic Algorithms** – (for each, find, sort, …)
  - **Object Functions**

STL – Data Structures

- array - New in C++11. A fixed sized array
- vector - A dynamic array (supports resizing)
- bitset - New in C++11. A bit array
- deque - A double-ended queue
- forward_list - New in C++11. A singly linked list
- list - A doubly linked list
- map - Hash table of key-value pairs
  - multimap - Hash table, supports numerous values stored with each key
- queue - a single-ended queue
- priority_queue - a priority queue

set - a set of values, based on a hash table, each value appears once

multiset – each value can appear several times

stack - a stack

unordered_map / unordered_multimap - New in C++11. unordered hash tables of key-value pairs

unordered_set - / unordered_multiset - New in C++11. unordered hash tables of values

(Generic) Algorithms Library

- All are included in <algorithm>
- Includes a variety of general algorithms
  - for_each (similar to our apply)
  - count
    - count_if
  - find
    - find_if
  - sort
  - Etc.

Object Functions Libraries
- Commonly used with STL
- Numerics library
  - Common mathematical functions
  - Complex numbers
  - Pseudo-random number generation

The Student Class

```
class Student{
    string name;
    int age;
  public:
    Student():name(""),age(0){}
    Student(string name,int age){
        this->name = name;
        this->age = age;
    }
    int getAge() const { return age; }
    string getName() const { return name; }

    friend ostream& operator<<(ostream& out, const Student& s){
        out << "name: " << s.name << endl;
        out << "age: " << s.age << endl;
        return out;
    }
};
```

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
```

A Vector of Students

```
vector<Student> students(5);
students[0] = Student("Moshe",18);
students[1] = Student("Avi",23);
students[2] = Student("David",17);
students[3] = Student("Yosi",17);
students[4] = Student("Jacob",30);

students.push_back(Student("Haim",32));

vector<Student>::iterator it;
for (it = students.begin(); it != students.end(); it++){ cout << *it << endl;}
```

Output:

name: Moshe
age: 18

name: Avi
age: 23

name: David
age: 17

name: Yosi
age: 17

name: Jacob
age: 30

name: Haim
age: 32

A Vector of Students

```
vector<Student> students(5);
students[0] = Student("Moshe",18);
students[1] = Student("Avi",23);
students[2] = Student("David",17);
students[3] = Student("Yosi",17);
students[4] = Student("Jacob",30);

students.push_back(Student("Haim",32));

vector<Student>::iterator it;
for (it = students.begin(); it != students.end(); it++){ cout << *it << endl;}

int underAgedCount = count_if(students.begin(), students.end(), AgeSelector());
cout << underAgedCount << endl; // output: 2

vector<Student>::iterator newEnd;
newEnd = remove_if(students.begin(), students.end(), AgeSelector());
```

```
struct AgeSelector{
 public:
    template<class T>
    bool operator()(const T& hasAge){
       return hasAge.getAge() < 18;
    }
};
```

Exercise: implement count_if

A Vector of Students

```
vector<Student> students(5);
students[0] = Student("Moshe",18);
students[1] = Student("Avi",23);
students[2] = Student("David",17);
students[3] = Student("Yosi",17);
students[4] = Student("Jacob",30);

students.push_back(Student("Haim",32));

vector<Student>::iterator it;
for (it = students.begin(); it != students.end(); it++){ cout << *it << endl;}

int underAgedCount = count_if(students.begin(), students.end(), AgeSelector());
cout << underAgedCount << endl; // output: 2

vector<Student>::iterator newEnd;
newEnd = remove_if(students.begin(), students.end(), AgeSelector());

sort(students.begin(), newEnd, NameComparator());

for (it = students.begin(); it != newEnd; it++){ cout << *it << endl;}
```

```
struct NameComparator{
 public:
  bool operator()(const Student& s1, const Student& s2){
     return s1.getName()< s2.getName();
  }
};
```

Output:

name: Avi
age: 23

name: Haim
age: 32

name: Jacob
age: 30

name: Moshe
age: 18

List Example

```
list<Student> studentsList;
studentsList.push_front(Student("Sara", 25));
studentsList.push_front(Student("Neomi", 25));
studentsList.insert(studentsList.end(), Student("Rachel", 22));
studentsList.push_back(Student("Lea", 26));
studentsList.erase(studentsList.begin());

list<Student> queue;
queue.insert(queue.end(), studentsList.begin(), studentsList.end());// the list
queue.insert(queue.end(), students.begin(), students.end());// the vector


while (!queue.empty()){
    cout << *queue.begin() << endl;
    queue.pop_front();
}
```

אוניברסיטת
בר-אילן

המחלקה למדעי המחשב

89-210 תכנות מתקדם 1

מרצה: ד"ר אליהו חלסצ'י

6.7
Object Functions to Lambda Expressions

DR. ELIAHU KHALASTCHI

Lambda Expressions

- Instead of creating structs for object functions,
- We can directly insert a *Lambda Expression*
  - It is an anonymous function
  - Behind the scenes:
    - An Object-Function class is created
    - An instance of the class is injected
  - The syntax:
    - [](parameters…) { implementation}

Lambda Expressions

name: Moshe
age: 18

name: Avi
age: 23

name: Jacob
age: 30

name: Haim
age: 32

```
struct AgeComparator{
 public:
  bool operator()(const Student& s1, const Student& s2){
     return s1.getAge()< s2.getAge();
  }
};
//...
sort(students.begin(), newEnd, AgeComparator());
```

```
// sort using a lambda expression
sort(students.begin(), newEnd, [](const Student& s1, const Student& s2){
    return s1.getAge() < s2.getAge();
});
```

Lambda Expressions – capture outer variables

```
template<class iterator, class Predicate>
void print_if(iterator begin, iterator end, Predicate p){
    while (begin != end){
        if (p(*begin))
            cout << *begin << ",";
        begin++;
    }
};

int main(){
    list<int> myList;
    for (int i = 0; i<10; i++)
        myList.push_back(i);

    print_if(myList.begin(), myList.end(), [](int x){return x>5; });

    return 0;
}
```

Lambda Expressions – capture outer variables

```
template<class iterator, class Predicate>
void print_if(iterator begin, iterator end, Predicate p){
    while (begin != end){
        if (p(*begin))
            cout << *begin << ",";
        begin++;
    }
};

int main(){
    list<int> myList;
    for (int i = 0; i<10; i++)
        myList.push_back(i);

    int y = 5;
    print_if(myList.begin(), myList.end(), [](int x){return x>y; });

    return 0;
```

error: 'y' is not captured

Lambda Expressions – capture outer variables

```
template<class iterator, class Predicate>
void print_if(iterator begin, iterator end, Predicate p){
    while (begin != end){
        if (p(*begin))
            cout << *begin << ",";
        begin++;
    }
};

int main(){
    list<int> myList;
    for (int i = 0; i<10; i++)
        myList.push_back(i);

    int y = 5;
    print_if(myList.begin(), myList.end(), [&y](int x){return x>y; });

    return 0;
```

Bar-Ilan University

המחלקה למדעי המחשב

89-210 תכנות מתקדם 1

מרצה: ד"ר אליהו חלסצ'י

6.8 Java <Generics>

AND THE “TYPE ENSURE” TECHNIQUE

DR. ELIAHU KHALASTCHI

C++ recap on templates

Source code:

```
template<class T>
class Holder{
  T* t;
  public:
  void set(T* t){ this->t = t; }
  T* get(){ return t; }
};
```

```
void main(){
  Holder<Student> hs;
  Holder<Employee> he;
  Holder<int> hi;
  cout << (typeid(hs)==typeid(he)) <<endl;
}
```

false

Complied code:

```
class Holder{
  Student* t;
  ...
};
```

```
class Holder{
  Employee* t;
  ...
};
```

```
class Holder{
  int* t;
  public:
  void set(int * t){ this->t = t; }
  int* get(){ return t; }
};
```

Java - before 1.5

```
public class Holder {
  Object t;
  public void set(Object t){ this.t=t;}
  public Object get(){return t;}
}
```

```
public static void main(String[] args) {
  Holder h=new Holder();

  h.set(new Student());
  ((Student)h.get()).study();

  h.set(new Employee());
  ((Employee)h.get()).work();
}
```

```
h.set(new Employee());
((Employee)h.get()).work();

//...

((Student)h.get()).study();
```

Exception! (at runtime! ☹)

Java - before 1.5

```
public class Holder {
  Object t;
  public void set(Object t){ this.t=t;}
  public Object get(){return t;}
}
```

```
public static void main(String[] args) {
  Holder h=new Holder();

  h.set(new Student());
  ((Student)h.get()).study();

  h.set(new Employee());
  ((Employee)h.get()).work();
}
```

Since 1.5 – generics!

```
public class Holder<T> {
  T t;
  public void set(T t){ this.t=t;}
  public T get(){return t;}
}
```

```
public static void main(String[] args) {
  Holder<Student> hs=new Holder<Student>();
  hs.set(new Student());
  hs.get().study();

  Holder<Employee> he=new Holder<Employee>();
  he.set(new Employee());
  he.get().work();
}
```

Ensured type safety

```
public class Holder<T> {
  T t;
  public void set(T t){ this.t=t;}
  public T get(){return t;}
}
```

```
public static void main(String[] args) {
  Holder<Student> hs=new Holder<Student>();
  hs.set(new Student());
  hs.get().study();

  Holder<Employee> he=new Holder<Employee>();
  he.set(new Employee());
  he.get().work();
}
```

Ensured type safety

```java
public class Holder<T> {
  T t;
  public void set(T t){ this.t=t;}
  public T get(){return t;}
}
```

```java
public static void main(String[] args) {
  Holder<Student> hs=new Holder<Student>();
  hs.set(new Student());
  hs.get().study();

  Holder<Employee> he=new Holder<Employee>();
  he.set(new Employee());
  he.get().work();
}
```

```java
Holder<Student> hs=new Holder<Student>();
hs.set(new Student());
hs.get().study();

//...

hs.set(new Employee());
```

Compilation Error ☺

“type ensure” - used by Java

```
public class Holder<T> {
  T t;
  public void set(T t){ this.t=t;}
  public T get(){return t;}
}
```

Syntax sugar

Complied code:

```
public class Holder {
  Object t;
  public void set(Object t){ this.t=t;}
  public Object get(){return t;}
}
```

```
public static void main(String[] args) {
  Holder<Student> hs=new Holder<Student>();
  hs.set(new Student());
  hs.get().study();

  Holder<Employee> he=new Holder<Employee>();
  he.set(new Employee());
  he.get().work();
}
```

Complied code:

```
public static void main(String[] args) {
  Holder hs=new Holder();
  hs.set(new Student());
  ((Student)h.get()).study();

  Holder he=new Holder();
  he.set(new Employee());
  ((Employee)he.get()).work();
}
```

“type ensure” - used by Java

Complied code:

```java
public class Holder<T> {
  T t;
  public void set(T t){ this.t=t;}
  public T get(){return t;}
}
```

```java
public class Holder {
  Object t;
  public void set(Object t){ this.t=t;}
  public Object get(){return t;}
}
```

Implication: We **can’t** write generic code that requires **runtime information**

- T t = **new** T();
- T[] array = **new** T[10];
- t.doSomething();

Compilation Error ☹

(ok in C++)

In addition:

(again, ok in C++)

```java
// Holder<int> hi;  - compilation error
Holder<Student> hs=new Holder<Student>();
Holder<Employee> he=new Holder<Employee>();
System.out.println((he.getClass()==hs.getClass()));
```

true

המחלקה למדעי המחשב

89-210 תכנות מתקדם 1

מרצה: ד"ר אליהו חלסצ'י