# C++

## Functors

# Overview

Introduction

Function pointers

Functors:

- Introduction / motivation
- Functors and maps
- Functional library

# Motivation

Container: object holding other objects.

Example:
std::vector<int> vi;
std::vector<float> vf;
class Student;
std::list<Student> ls;
std::map<std::string, int> phone_book;

Containers provide limited set of methods for its manipulation.
Example: verify if empty (empty()), get number of elements (size()), get iterators (begin(), end()), etc

# Motivation

For containers to be useful we need more functionality such as: copy, sort, search for elements, transform each element in a container (map), etc.

These are provided by the algorithms in the standard library. They are in the header: <algorithm> in the namespace std.

Glue between algorithms and containers is provided by the iterators (see lecture 12).

Example: sorting the elements in a container is done by calling std::sort on the appropriate iterators.

# Motivation

Example: sorting the elements in a container is done by calling std::sort on the appropriate iterators. The example below sorts the characters of a string:

```
int main(void) {
 std::string s = “hurry-up”;

 // sort the container in place
 std::sort(s.begin(), s.end());

 std::cout << s << std::endl;
 return 0;
}
```

# Motivation

Customization of the behavior of these algorithms is done through *function objects* (also called *functors*).

Examples of customization: specifying the function to be mapped to a container, specifying the function to be used when sorting a container, etc

# Function pointer

This is a technique inherited from the C language.

Any function in C++ (and in C) has a an address in memory (where the function definition is stored).
A *function pointer* is a variable (a pointer) storing the address of a function.

The notation looks a bit heavy, but a function pointer works like a regular pointer.

# Examples

```
// In all the definitions below, f is a pointer - i.e. a variable
//
// A pointer to a function taking no argument and returning
// no argument
void (*f) (void);

// A pointer to a function taking two ints and returning an int
int (*f) (int a, int b);

// A pointer to a function taking two void pointer and returning
// an int
int (*f) (const void* a, const void* b);
```

# Examples

```
// define the function min
int min(int a, int b) {
 return a < b ? a : b;
}

// define a function pointer f
int (*f) (int a, int b);

int main(void) {
 f = min; // or: f = &min;
 std::cout << f(1, 2) << std::endl;
 return 0;
}
```

# Why?

Why do we need function pointers?

Function pointers are used as *callback.*

Callbacks are registered to the underlying system, i.e. we tell precisely to the system which function needs to be called when a given event is received.

Examples of use are in graphical user interface (GUI) where we want to attach a functionality to a given event.
The following example in OpenGL/glut specifies the function to be called on a display event.

# Callback example

```
void Display (void) {
 glClearColor(1.0,0.0,0.0,1.0);
 glClear(GL_COLOR_BUFFER_BIT);
}

int main(int argc, char** argv) {
 glutInit(&argc, argv);
 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
 glutInitWindowSize(500,500);
 glutCreateWindow(“test”);
 // Register the function Display() to be called on display event
 glutDisplayFunc(Display);
 glutMainLoop();
 return 0;
}
```

# Why?

Why do we need function pointers?

Function pointers are used for customizing algorithms.
Example: the function qsort in the C standard library allows to sort an array using a comparison function (passed by pointer)

```
void qsort(void* ptr, size_t count, size_t size,
  int (*comp) (const void* f, const void* s));
```

# Example: sorting

```
#include <stdlib.h>

static int greater(const void* first, const void* second) {
 int f = *((const int*)first);
 int s = *((const int*)second);
 if (f < s) return 1;
 if (f > s) return -1;
 return 0;
}

int main(void) {
 // assume int* array is filled with array_size ints
 // sort in place:
 qsort(array, array_size, sizeof(int), greater);
 return 0;
}
```

# Example: find an element

find() and find_if() are two functions in the C++ standard library that searches for an element within a range provided by two iterators.

They are defined in the header <algorithm> in the namespace std.

find() searches for the first element with a given value and returns an iterator corresponding to its location.

find_if() searches for the first element that satisfies a given criteria and returns an iterator corresponding to its location.

# Example: find an element

```
bool LessThanFive(int a) { return a < 5; }

int main(void) {
 // std::vector<int> v containing {1, 4, 5, 11, 12, 15, 9, 12}
 std::vector<int>::iterator it;
 it = find_if(v.begin(), v.end(), LessThanFive);
 if (it == v.end()) {
   // no element less than five was found
 } else {
  // it is an iterator to the first element less than five in v
 }
 return 0;
}
```

# Function pointers and STL algorithm

As seen in the previous example, function pointers can be used to customize the behavior of STL algorithms.

What is the problem with function pointers then?
Or said differently: why do we need something else?

Function pointers lack flexibility (because they are stateless).
In the previous example, the criteria (less than five) is hardcoded in the function and can not be customized.

# Function object

A *function object* is a generalization of the notion of function pointer.
Function objects are also often called *functors* (both names refer to the same thing).

A function object is an object that behaves *like a function*. This means that it can be “called” with a syntax identical to a function call.

A function object, however, is not a regular function, but an object: i.e an instance of a class.
Function objects behave like function because they overload the call function operator (i.e. they overload *operator()*).

# Example

```
class StringLengthFunctor {
 public:
  size_t operator() (const std::string& str) {
    return str.size();
  }
};

int main(void) {
 std::string s = “astring”;
 StringLengthFunctor f;
 std::cout << f(s) << std::endl;
 return 0;
}
```

# Function object

A function object is an object that can be called like a function.
But what kind of advantages does it give us?

An important difference between a regular function and a function object is that since a function object is an instance method, it can access the object data members.
In contrary to a regular function, a function object has a state.

If a function object needs data beyond what is provided by its parameters, then we can store that information as a data member inside the object.

# Example

```
class StringAppender {
public:
 StringAppender(const std::string& prefix) : prefix(prefix) {}
 std::string operator() (const string& str) {
  return prefix + “ “ + str;
 }
private:
 std::string prefix;
};

int main(void) {
 StringAppender appender(“programming”);
 std::cout << appender(“in c++”) << std::endl;
 std::cout << appender(“in c”) << std::endl;
}
```

# Example: find an element

Let us generalize and rewrite the function LessThanFive used in the precedent “find_if” example:

```
template<class T>
class LessThan {
public:
 LessThan(const T& bound) : bound(bound) {}
 bool operator()(const T& arg) { return arg < bound; }
private:
 T bound;
};
```

# Example: find an element

The line:
it = find_if(v.begin(), v.end(), LessThanFive);
is changed to:
LessThan<int> LessThanFive(5);
it = find_if(v.begin(), v.end(), LessThanFive);

Or we could combine both lines in one:
it = find_if(v.begin(), v.end(), LessThan<int>(5));

Here *LessThan<int>(5)* constructs a temporary *LessThan* object functor with parameter *5*, then passes it to the *find_if* algorithm.

# Storing objects in STL maps

Objects stored in STL maps (and derivatives) need to be comparable. That is: they need to overload *operator*<.
However, what if we want to store objects in a map but not using the default comparison operator.

Example: Consider a set of c-string: std::set<char *>. The default comparison operator compare two *char** by seeing which one has the lowest address in memory.
Suppose that we want to find if a string is an element of a set, then *find()* would work by returning whether a given pointer is in the set.

# Storing objects in STL maps

We can resolve this issue by providing a functor to compare elements of the set.

```
class CStringCompare {
public:
 bool operator() (const char* l, const char* r) {
   return strcmp(l, r) < 0;
 }
};
```

Then the set of c-string is created as:

```
std::set<char*, CStringCompare> s;
```

The same thing would apply for a map:

```
std::map<char*, int, CStringCompare> m;
```

# Function object bases

The standard library provide base classes to help writing function objects and to support higher order programming: *unary_function* and *binary_function*.

They are defined in header <functional> and are in the namespace std.

The goal of these functions is to provide standard names for the argument and return types.

For *unary_function*: *argument_type* and *result_type*
For *binary_function*: *first_argument_type*, *second_argument_type* and *result_type*.

# Function object bases

Unlike other classes *unary_function* and *binary_function* contain no data or methods; instead all they do is export information from the template types.
This information is needed to make our functors working with the rest of the functional programming library of the STL.

The type information from *unary_function* and *binary_function* is imported into a user defined functor with inheritance.
Example:

```
class MyFunctor: public unary_function {
public:
 bool operator() (const std::string& s) {
  // do something with s
 }
};
```

# Function objects in the standard library

In addition to the base classes, the standard library provides some function objects.

They are defined in header <functional> (namespace std).

They can be classified in:

- predicate: function object that returns a bool (equal_to, not_equal_to, ...)
- arithmetic: function object providing numeric operation (plus, minus, ...)
- binder, adapter and negater: for composing function objects

# Predicate

| Name | Arity | Description |
|---|---|---|
| | | |
| equal_to | binary | arg1 == arg2 |
| not_equal_to | binary | arg1 != arg2 |
| less | binary | arg1 < arg2 |
| greater | binary | arg1 > arg2 |
| less_equal | binary | arg1 <= arg2 |
| greater_equal | binary | arg1 >= arg2 |
| logical_and | binary | arg1 && arg2 |
| logical_or | binary | arg1 \|\| arg2 |
| logical_not | unary | !arg |

# Example: predicate

By default *sort()* sorts elements from the lowest to the highest:

```
// assume std::vector<int> v
// sort elements of v in place from lowest to highest
std::sort(v.begin(), v.end());
```

Sorting the elements in the opposite order can be done using the *greater* predicate from the STL:

```
// sort elements of v in place from highest to lowest
std::sort(v.begin(), v.end(), std::greater<int>());
```

# Arithmetic

| Name | Arity | Description |
|---|---|---|
| | | |
| plus | binary | arg1 + arg2 |
| minus | binary | arg1 - arg2 |
| multiplies | binary | arg1 * arg2 |
| divides | binary | arg1 / arg2 |
| modulus | binary | arg1 % arg2 |
| negate | unary | -arg |

# Binder, adapter, negater

*Binder* allows to use a two-argument function as a one-argument function by binding one argument to a value: *bind1st*, *bind2nd*

*Adapter*

Member function adapter allows a member function of a user defined class to be used as an argument to algorithms: *mem_fun*

Pointer function adapter allows a pointer function to be used as an argument to algorithms: *ptr_fun*

*negater* negates a predicate: *not1* (unary)*, not2* (binary)

# Example: adapter and binder

bool LessThan(int arg, int thresh) { return arg < thresh; }

Using an adapter, we can transform this regular function in a functor:
std::ptr_fun(LessThan)
we can then use a binder to bind the second parameter to a given value:
std::bind2nd(std::ptr_fun(LessThan), 5);

Let us rewrite the previous example with find_if:
it = find_if(v.begin(), v.end(), std::bind2nd(std::ptr_fun (LessThan), 5));

# Example: negater

Let us use a negater to find the first element greater than or equal to 5:
it = find_if(v.begin(), v.end(), std::bind2nd(std::not2(std::
ptr_fun(LessThan)), 5));

Since std::not2 returns a functor, we can pass the result to std::bind2nd to generate a unary function that returns true whether the argument is at least a value of 5

# Limitations of the functional library

The STL functional library is unfortunately limited:
it provides only support for adaptable unary and binary functions.

The new version of the standard (C++ - 11) provides improvement with the possibility to bind or create functors from regular functions with any arity (not only one or two).

# Summary

STL algorithms can be customized with the help of function pointers or functors.

A functor or function object is a generalization of a function pointer.

A function object is an object that can be called “like” a function.

Unlike a function pointer, it is a class instance and has a state.

A function object is obtained by overloading *operator()*

# Summary

The standard library provides base classes and helpers.

Base classes export template types, which are imported by user-defined classes with inheritance.

Helpers allow to create a functor from a regular function or from an instance method; to bind a parameter to a value; etc