Callables

Michael Hanke

School of Engineering Sciences

Program construction in C++ for Scientific Computing

Outline

Functions as parameters

Functions are special kinds of objects: They can be invoked and provide a result.

Example: Homework1, Task 2

```
typedef double (*ASI_fkt)(double);
double ASI(ASI_fkt f, double a, double b, double tol);
```

We can use this function as follows:

```
double ptfun(double x) {
    return 1.0+std::sin(std::exp(3.0*x));
}
double I = ASI(ptfun,0.0,1.0,1e-10);
```

Classes with operator()

- Consider

```
class clsfun {
public:
    double operator()(double x) {
    return 1.0+std::sin(std::exp(3.0*x));
  }
};
clsfun cf;
double I = ASI(cf,0.0,1.0,1e-10);
```

- This will not work: `cf` cannot be converted (type casted) to type `ASI_fkt`.
- Remark: `cf` is a so-called *functor*, i.e. an object that acts like a function.

We will need a new data type!

Callables

- Every callable object has a *call signature* consisting of: result type and sequence of argument types
  - Repeat: Signature of a function (used for binding functions [dynamic or static])
- New template class: `function<`*`call_signature`*`>` (defined in library header file `functional`) where *`call_signature`* is *`return_type`*`(`*`arg_list_types`*`)`

Example:

```
#include <functional>
typedef std::function<double(double)> ASI_fkt;
```

New Version of ASI

```
#include <functional>
typedef std::function<double(double)> ASI_fkt;
double ptfun(double x);
clsfun cf;
double Ip = ASI(ptfun,0.0,1.0,1e-10);   // Works!
double Ic = ASI(cf,0.0,1.0,1e-10);      // Works!
```

Why does it work?

There is a type cast from both `ptfun` and `cf` to `ASI_fkt`!

- This is the C++ way of handling function-like objects (callables).

The Problem

- Assume the following parametrized function be given:

```
double ptfunpar(double x, double par) {
    return 1.0+std::sin(std::exp(par*x));
}
```

- What do we want:

```
double par = 3.0;
Ipp = ASI(ptfunpar( ,par), 0.0,1.0,1e-10);  // Error!
```

- The first argument to `ASI` must be a callable taking one double as an argument and returning a double. However, `ptfunpar` takes two doubles as argument.

- *Wanted:* A mapping of `ptfunpar( ,par)` to a function of type `ASI_fkt`.

How to fix this classical problem?

A classical solution: Global parameters

```
static double par;
static double wrapper(double x) {
  return ptfunpar(x,par);
}
int main() {
    par = 3.0;
    Ipp = ASI(wrapper,0.0,1.0,1e-10);  // Works!
}
```

The use of global parameters is a source of bugs (and often a security risk!).

The C++ solution: `bind`

- The general form of a call to bind is

```
auto
newCallable = std::bind(callable,arg_list);
```

- For an existing callable, this makes a new callable *`newCallable`*.
- When invoking *`newCallable`*, the arguments from *`arg_list`* are passed.
- *`arg_list`* may contain placeholders of the form `_n` (*`n`* being a positive integer).
  - Semantics: `_1` ist the first argument of *`newCallable`*, `_2` is the second etc.
  - Note: The placeholders `_n` are defined in the namespace `std::placeholders`.

Example

```
#include <functional>
using namespace std::placeholders;
auto bindfun = std::bind(ptfunpar,_1,3.0);
```

- both `bind` and `placeholders` are defined in the header `functional`.
- As a side effect, this mechanism can be used to reorder arguments:

```
auto reordfun = std::bind(ptfunpar,_2,_1);
```

bind and reference parameters

So far, only arguments passed by value can be provided for non-placeholder arguments.

- For arguments passed by reference, the mapping function `ref` must be used.

Example:

```
#include <functional>
double ptfunrefpar(double x, double & par) {
    return 1.0+std::sin(std::exp(par*x));
}
double p = 3.0;
using std::placeholders;
auto bindfun = bind(ptfunrefpar,_1,ref(p));
```

- For reference parameters to `const` objects, `cref` must be used instead.

Lambda expressions

- *A lambda expression is a special kind of callable*: Comparable to an unnamed, inline function
- A lambda expression (short: lambda) has a return type, a parameter list, and a function body.
- A lambda can be defined everywhere, even inside other functions. So it is local to where it is defined.
- Structure:

```
[capture_list](parameter_list) -> return_type
    { function_body }
```

- `(parameter_list)` can be omitted if it is empty.
- `-> return_type` can be omitted if `function_body` consists only of a `return` statement.
- *An empty capture list is denoted by `[]` and cannot be omitted!*

Capture list

- Indicates which variables of its environment shall be accessible inside the function body.
- The variables can be captured both *by value* (indicated by providing the name) or *by reference* (indicated by a preceding &).
- Special forms of the capture list:
  - `[=]` All used variables from the surrounding unit are captured by value.
  - `[&]` All used variables from the surrounding unit are captured by reference.
  - More general forms: `[=,`*`reference_list`*`]` and `[&,`*`identifier_list`*`]`.

Example

Remember:

```
double ptfunpar(double x, double par) {
    return 1.0+std::sin(std::exp(par*x));
}
double ASI(ASI_fkt,a,b,tol);
```

Then we can use lambdas:

```
int main() {
  // something
  double par = 3.0;
  auto lamfun = [par](double x) -> double {
      return ptfunpar(x,par);
  };
  // return type can be omitted here:
  // auto lamfun = [par](double x)
  //       { return ptfunpar(x,par); };
  double I1 = ASI(lamfun,0.0,1.0,1e-10);
}
```

Capture list: Good to know

- Variables captured by value cannot be changed inside the function body of a lambda unless the keyword `mutable` is used:

```
double par = 2.0;
auto lamfun = [par](double x) -> double {
    par += 1.0;        //  Error!
    return ptfunpar(x,par);
};
```

- But this one will work:

```
double par = 2.0;
auto lamfun = [par](double x) mutable -> double {
    par += 1.0;        //  OK!
    return ptfunpar(x,par);
};
```

par will retain its changed value between function calls!
Guess why? (*Hence, this version is meaningless to use in ASI*)

- However, that value outside of the lambda will not change!

```
double Il = ASI(lamfun,0.0,1.0,1e-10);
cout << par << endl;   // result is 2.0, not 3.0!
```

Capture list: Good to know (cont)

- *Variables captured by value will be evaluated when the lambda is defined*, not when it is used:

```
double par = 3.0;
auto lamfun = [par](double x) {
    return ptfunpar(x,par);
};
par += 1.0;
// par in lamfun has the value 3.0, not 2.0
double Il = ASI(lamfun,0.0,1.0,1e-10);
```

Putting everything together: A comprehensive example

It is demo time!

Summary

- Callable objects
- The library template class std:: function<...>
- bind
- Lambda expressions

- What comes next
  - Even more stuff.