Annex B: C++ Function Objects and Lambda Expressions

In C++, a *callable* object is an object or expression to which the call operator () can be applied. Functions and—as we will see below—function pointers are callable objects inherited from C. C++ adds two other callable types of objects, *function objects* and *lambda expressions*, which are the subject of this annex.

B.1 FUNCTION OBJECTS

Function objects are heavily used by STL algorithms to incorporate user-defined behavior in the library. They generalize the concept of function pointers, enabling some programming capabilities that are not easily implemented with function pointers alone. We have seen that function objects are used by TBB in the STL-like algorithms implementing parallel patterns, as well as in the `task_group` programming interface, to specify the user-defined task to be executed by a worker thread.

B.2 FUNCTION OBJECT SYNTAX

A *function object*, also called a *functor*, is a C++ object, instance of an ordinary C++ class that has just one particular quality: it defines the function `operator()` as a member function of the class (possibly together with other more traditional member functions). This particular member function has in turn a specific quality: it is a *no-name* function called with a syntax that mimics an ordinary function call, *using the arbitrary object name as if it was a function name*. Here is the precise syntax:

```
// The function object class:
// – – – – – – – – – – – – – – –

class MyFunctor
   {
   private:
   // private data and member functions

   public:
   retval operator()(arg1, arg2, ...);
   ...
   // other public data and member functions
   };

// The client code:
// – – – – – – – – –
```

Continued

```
MyFunctor fobj;                    // declare object fobj
retval k = fobj(arg1, arg2, ...); // function call syntax
```

LISTING B.1

Function object syntax

All that happens here is just using the operator overloading capability of C++ to overload the function call operator(). It is important to keep in mind that:

- A functor class is an ordinary class, and a functor object is an ordinary object. It is possible to encapsulate as much data and function members as needed to provide the planned service, and they can all be accessed with the usual C++ syntax for objects: *object name* followed by *member function name*. The only new feature is the presence of the operator() member function, *which is called with the object name only*.
- The signature of the operator() member function is arbitrary: the return value retval can be any type, and there can be as many arguments as needed. Moreover, this member function can be overloaded: it is possible to define *several* operator() functions with different signatures inside the same function object class. The compiler will either select the one that matches the function call, or complain if there is none.

A special operator is then defined as a member function, implementing a function call semantics. We can wonder about the reason for doing this. After all, when services are required from a C++ class, it is always possible to add an ordinary member function, give it a name, and call it in the usual way. However, the fundamental observation here is that function objects implement the same syntax as *function pointers*, which we show in the listing below.

```
retval (*fptr)(arg1, arg2, ...);    // declaration of fptr

// Initialize the pointer: assume that my_function() has the
// correct signature. The function name is also its address.
// We can, but you don't need,  use &my_function to initialize
// the pointer:

fptr = my_function;

// Now, call the function my_function() through the pointer.
// Again, we can use (*fptr) but (fptr) works too: the function
// pointer is also "callable", and dereferencing is not needed.

retval k = fptr(arg1, arg2, ...);
```

LISTING B.2

Function pointer syntax

Function objects and pointers have therefore the same syntax. Function objects can be used at the places where function pointers are expected, that is, at places where a callable object is required. Note that the function pointer fptr is, as the basic types (int, double, etc.), an object not associated with a C++ class. We can therefore say that function objects generalize function pointers in the same way

that C++ classes generalize the basic types that are not associated with a class, with the extra bonus that they can solve problems that ordinary function pointers cannot cope with (an example follows later on).

B.2.1 IMPLEMENTING CALLBACKS

As we know, the main utility of function pointers is to implement *callbacks* that customize the behavior of a library. Library functions that implement a service often require from the user a pointer to a function that sharpens its behavior. The classical example is sorting: a function that sorts an array is called, but this function in turn expects that caller to provide—through a function pointer—the function used to compare two array elements. As emphasized before, function objects can be used at every place a function pointer is expected. The STL uses them heavily to customize its behavior.

Many C++ libraries use function objects to implement callbacks. These libraries give the following instructions to the user:

- Define a class `XXX` (the name of the class is arbitrary) that defines an `operator()` with such and such signature, which performs such and such service.
- Then, declare an object instance of this class and pass me its name (again, the name is arbitrary).
- Or, if it suits you, just pass me the class name. Since the object name is irrelevant, I can construct it myself (we give an example later on).

In this way, users end up passing behavior to the library *without naming conventions*. Only the function object `type` is relevant, as well as the implicit existence of an adequate `operator()` member function in the class.

B.2.2 ABILITIES OF FUNCTION OBJECTS

Let us now examine how function objects enlarge the abilities of function pointers.

Function objects have better runtime performance

It is argued that function objects have better performance than function pointers, because the function call is resolved at *compile time* rather than at *execution time* (as is the case for function pointers). This argument is compelling only if functions are called millions of times.

Easier handling of functions with internal states

Function objects are objects, and they can therefore maintain an internal state between calls that may decide the outcome of the next call (remember the random number generators, etc). Functions can also maintain an internal state through `static` variables that are preserved between calls, but we know that we cannot have more than one function per process, unless we are prepared to explicitly code several functions with different names, all doing the same thing. With function objects, instead, we can declare as many of them as we want or need. Each one will have its own encapsulated internal state totally independent of the others.

Extending the capabilities of generic programming

C++ templates were initially invented in order to parameterize data types, and write generic code that applies as well to `floats` or `doubles`, for example. Since a function object is also a data type, it

can also be parameterized by a template parameter. However, in this case, we are doing much more than text processing, because the specialization to a specific class specializes the functions that can be called with the (arbitrary) object name. Therefore, the template parameter is now used to parameterize *behavior*.

This is what the STL does with the algorithms acting on containers. Often, these algorithms admit two template parameters: one for the type of the element stored in the container, and another one for the function object class that completes the specification of the action of the algorithm on the container.

B.2.3 EXAMPLE

We present next a simple example that explicitly shows the utility of disposing of an internal state inside function objects to cope with contexts that cannot easily be solved with function pointers. Given an integer array of size N, we want to deal with the following two problems:

- To find the first negative element.
- To find the first element larger than a given value val, not known at compilation time.

We will look at this problem by using the find_if STL algorithm:

pointer pos = find_if(*pointer* beg, *pointer* end, *predicate* cond)

where

- beg is a pointer to the first element of the array.
- end is a pointer past the end of the array.
- cond is a pointer to a function that receives an integer and returns true if the argument verifies the search criterion. In the STL, functions that return a bool are called *predicates*.
- The return value pos is a pointer to the position of the found element. If pos==end, we know the search failed.

In the first problem, the condition is known at compilation time, and it can be managed with a function pointer: it is easy to write a function that decides if its argument is positive or negative. However, the second problem *cannot* be managed as a function pointer, because the function needs to compare the integer elements of the array to a value not known at compilation time, so two arguments must be passed to the predicate function. But this is not possible in this case, because the find_if function expects a predicate signature with only *one* argument.

Here, function objects come to our rescue. A function object can be used, with an internal state that stores val passed as argument when the object is constructed. Then the operator()(int n) function compares n to val.

Example: fctobj_example.C

This file contains the complete code for this example. We give below the main elements of the example.

```
// Function that solves the first problem:
// – – – – – – – – – – – – – – – – – – – –
bool IsNegative(int& n)
   {
   if n<0 return true;
   else return false;
   }

// Function object class for second problem
// – – – – – – – – – – – – – – – – – – – – –
class IsBigger
   {
   private:
   int value;

   public:
   IsBigger(int n) : value(n) {}  // constructor

   bool operator()(int& n)        // the predicate
      {
      if n>value return true;
      else return false;
      }
   };

// client code:
// – – – – – – –

int array[N];

// Solve first problem:
// – – – – – – – – – – –
int *pos = find_if(array, array+N, IsNegative);

// Solve second problem:
// – – – – – – – – – – –
IsBigger fctobj = IsBigger(3)   // call constructor, value=3
int *pos = find_if(array, array+N, fctobj);

// but, the name of the object is totally arbitrary, so one does not
// really need to pass it. The following code, that passes directly the
// object constructor, also works:

int *pos = find_if(array, array+N, IsBigger(3));
```

LISTING B.3

Using function objects

B.2.4 FUNCTION OBJECTS IN TBB

In the SPool class, tasks were defined by a pointer to a task function. We have seen that the TBB parallelization algorithms like parallel_for or parallel_reduce or the taslgroup construct are functions or class templates that ask the client code to provide, instead, a function object to specify a task to be performed by a thread, through the overloading of the operator() member function.

Clearly, function objects are used to specified behavior, but there is more than that. The basic task scheduling model is a fork-join of children tasks. Let us assume that a thread executing a task divides the target data set into two halves, and forks a child task, executed by a new thread, to perform half of the initial job. This new thread needs to know what specific task to perform on the reduced data set, so it could naturally get a pointer to the same task function the parent thread is executing. However, what happens if we are running a task function *with a persistent but evolving internal state* needed to implement the algorithm requirements? We end up with a nonthread-safe context, with several threads accessing the same nonthread-safe function.

If the task is defined with a function object, we are much better off, *because function objects, being objects, can be copied*. Therefore, the new thread *does not* get the same function object, *it gets instead a brand new function object* totally independent of the initial one, which may be a full or a partial copy of the parent task function object, according to algorithm requirements. C++ is very flexible in allowing the programmer to define constructors that copy objects. A complete discussion is provided in the TBB chapter, where the following issues were discussed in detail:

- The parallel_for algorithm is well adapted to embarrassingly parallel contexts, where there are no partial results coming from different threads to be collected. The function object class required by this algorithm may have a private internal state if that helps to implement the task, but this internal state must be immutable: the operator() function cannot change it. In this case, TBB only requires the standard, implicit, copy constructor.
- The parallel_reduce algorithm applies when partial results must be combined in some way. The function object class *must* have a public internal state used to accumulate or store partial results. In this case, TBB asks you to provide another constructor, called the *split constructor*, which copies the function object but in addition re-initializes the public internal state to the initial default values. In this way, a new thread that is activated to participate in the parallel execution gets a function object in which the history of previous computations by another thread is forgotten.

B.3 LAMBDA EXPRESSIONS

We have seen in discussing function objects that a new type—a C++ class—must always be defined, but the names of the objects instances of this class—in all cases arbitrary—can even be anonymous, as in the example above in which we passed directly the class constructor rather than the function object name.

Lambda expressions—introduced in the C++11 standard—push this idea much further. They can be seen as an *inlined, "on the fly" definition of an unnamed function object*. When the compiler meets a lambda expression, it certainly creates a new type, but the class name—as well as the function object it instantiates—do not need to be known by the client code. This avoids the necessity of explicitly defining the function object class, making the code much less verbose. The other essential property of lambda

expressions is the "on the fly" definition, which means that a lambda expression can be defined *at the place it is needed, inside a enclosing function*, which is not the case for traditional inlined functions. We have used lambda expressions in Chapter 11 to define "on the fly" task functions passed to a thread manager utility.

The lambda expression syntax provides all the information the compiler needs to define the overloaded operator() in the implicit function object. The most basic syntax is [ ]{...}, where the initial brackets identify a lambda expression, defining a callable unit of code between the following braces. The simplest lambda expression, defining a function that takes no arguments and returns void, is:

```
// A simple lambda expression taking no arguments, with
// no return value
// – – – – – – – – – – – – – – – – – – – – – –

[] { // function body; }
```

LISTING B.4

Very simple lambda expression

The function so defined, like an ordinary function, can reference the global variables in the application, and call global functions. If return type and parameter lists are required, the following general syntax is used:

[*capture list*] (*parameter list*) –>*return type* { *function body* }

Let us examine the different items in the expression above:

- *Return type*:
  - If the function body contains only a return statement, the return type does not need to be specified. It is taken directly from the type returned in the function body.
  - If, instead, the function body contains more than a single return statement, the return type must be explicitly specified. Otherwise, the lambda expression ignores the return statement and returns void.
- *Parameter list*: This is the set of arguments to be passed to the operator() function
- *Capture list*: Defines the way local variables in the enclosing environment are accessed (captured).
  - A lambda expression can use a local variable from its surrounding function *only* if it captures that variable in its capture list.
  - A totally empty capture list means that no local variables from the surrounding environment are used.
  - Local variables captured by the lambda expression become data members in the implicit function object class.
- *Capture codes*: Local variables can be captured by value (their values are copied) or by reference (a reference to the local variable is defined in the implicit class). The capture codes are:
  - [ ]: This lambda does not use any local variable from the surrounding function.
  - [=] Any surrounding local variable is captured by copy.
  - [&] Any surrounding local variable is captured by reference.

- [=, &a, &b] Default is copy but a and b are captured by reference.
- [&, =a, =b] Default is reference but a and b are captured by value.

Using these conventions, it is very easy to rewrite the complete Listing B.3 as, follows, avoiding the explicit definition of predicate functions and function objects.

```
// Solve first problem, n>0:
// – – – – – – – – – – – – – –
int *pos = find_if(array, array+N, [](int& n)–>bool {if n>0 return true;
                                                     else return false; });

// Solve second problem, n>value:
// – – – – – – – – – – – – – – – –
int value;
int *pos = find_if(array, array+N, [=](int& n)–>bool
                                      {if n>value return true;
                                       else return false });
```

LISTING B.5

Listing 3, using lambda expressions

In the first case, we are dealing with a simple function with no internal state, and no local values are captured. Since the body function is not just a return statement, the bool return value must be specified. In the second case, a function object with internal state is needed, and the integer value to be compared to n needs to be captured.

A another example of lamba expressions, we can reexamine the listing 6.11 of the TimedWait_S.C example in chapter 6, where a timed wait in the C++11 environment was discussed. Here is a new version of this example, using a lamba expression instead of a function pointer.

```
using namespace std;

condition_variable CV;
mutex my_mutex;
bool predicate = false;

void worker_thread()
   {  // same as in the original code }

int main(int argc, char **argv)
   {
   bool retval;
   chrono::duration<int, milli> delay(1000);
   thread T(worker_thread);
   T.detach();
   do
      {
      unique_lock<std::mutex> my_lock(my_mutex);
      retval = CV.wait_for<int, milli>(my_lock, delay,
                                       []{return predicate;} );
```

```
    if(!retval) cout << "\n Timed out after 1 second" << endl;
    }while(!retval);
cout << "\n Wait terminated " << endl;
}
```

LISTING B.6

TimedWait_S.C, new version

In the original code, an auxiliary function bool Pred() was defined that returned the predicate. Now this is done on the fly with a simple lambda expression. No local variables are captured no arguments are passed and, since the function body is just a return statement, the bool return value does not need to be specified.

More examples of the usage of lambda expressions can be found in Chapter 11, dealing with the TBB taskgroup utility.