CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 25
December 7, 2016

Function-Like Constructs

Named functions

C and C++ support *named global functions*. Declaration syntax is
`return_type name( parameters...) {function_body}`

C++ also permits *named member functions*. These are declared inside of a class using the same syntax as for named global functions.

The function name is qualified by the class name. Thus, the name of `f` within class `C` is `C::f`

Function calls

The syntax for calling a named global function is
`name( arguments...)`

The function call is an expression whose type is the `return_type` of the function.

The syntax for calling a named member function of class `C` is
`implicit_arg.name( explicit_arguments )` .

The implicit argument must be an instance of `C`. If called from within `C`, the implicit argument defaults to `*this`.

Function types

The type of a named global function is its **signature**, that is, the return type and the list of parameter types.

For example, a function with a double and an integer reference as parameters that returns a long integer would have signature `long int (&) (double, int&)`

To define such a function, we would add names for the function and parameters and a body to perform the computation.

```
long int (myfun) (double d, int& k) {
    return (k += 2.5*d);
}
```

The parens around `myfun` are optional and would normally be omitted.

`typedef` for function types

Function types can be named using `typedef`. The declaration

```
typedef long int (myFunType) (double, int&);
```

defines the type name `myFunType`.

One can then declare a function to be of that type.

```
myFunType myfun;
```

and use it to define an alias `myfun2` for `myfun`:

```
myFunType& myfun2 = myfun;
```

Without the `typedef`, it would look like this:

```
long int (&myfun2) (double, int&) = myfun;
```

(See demo `25a-Functors/funtypes.cpp`.)

A new way to give names to types

C++ now provides another syntax for defining new type names (since c++11). Instead of

```
typedef long int (myFunType) (double, int&);
```

one can write

```
using myFunType = long int (&) (double, int&);
```

(See demo `25a-Functors/using.cpp`.)

Function pointers

One can have pointers to functions. Function points can be passed as the argument to functions that take function parameters such as the standard template function `sort()`. The third argument is the comparison function `comp` to be used when compairing elements. It returns `true` if its first argument is "smaller" than its second.

```
bool descendingOrder( int i, int j)
    { return (j<i); }
using Comp = bool (*) (int, int);
Comp cptr;
cptr = descendingOrder;
sort (myvector.begin(), myvector.end(), cptr);
```

(See demo `25a-Functors/sort-funptr.cpp`.)

Anonymous functions (a.k.a. lambda functions)

C++ now has anonymous functions. The previous example could be rewritten as

```
sort (myvector.begin(), myvector.end(),
     [] (int i, int j) { return (j<i);} );
```

(See demo `25a-Functors/sort-lambda.cpp`.)

Functors

A **functor** an instance of a class that defines `operator()`. It can be called using function syntax. Here's how it can be used as a sort comparator.

```
class ComparAscending {
public:
   bool operator()(int i, int j) { return (i<j); }
};
...
sort( myvector.begin(), myvector.end()-4,
      CompareAscending() );
```

(See demo `25a-Functors/sort-functor.cpp`.)

Closures

Functors and lambda expressions gain their power through closures.

A **closure** is an expression in which some of the variables have been **bound** to values, whereas other remain as function parameters.

A closure can be used like any other function.

A functor with data members acts like a closure. It result may depend on the values of those data members. Different instantiations of the functor class may result in functors with different behaviors since the data members may have different values.

Lambda capture

A lambda expression has the (simplified) syntax:
`[capture_list] (parameters...)  -> result_type {body}`

When evaluated, the result is a closure with the variables on the capture list being "imported" into the body.

In fact, when a lambda expression is evaluated, the result is a functor of a new compiler-generated class.

The `function` template in header file `<functional>` can be used to turn a closure into an object of a function type.

Closure example

Demo `25a-Functor/closure` uses `map`, `function`, and closures with value-capture to define a table of sales tax functions.

The map key is a state name (or abbreviation), stored as a `string`. The map value is a tax computation function that takes a sales amount as its argument and returns the sales tax. It uses the captured variable `rate` to computer the tax.

Each map pair has a tax computation function that was generated at run time to use a particular tax rate in its computation. The rate capture takes place in the line

```
taxFunType taxfun = [rate](double amt) { return amt*rate; };
```

Graphical User Interfaces

User Interfaces

Modern computer systems support two primary general-purpose user interfaces:

Command line: User input is via a command line typed at the keyboard. Output is character-based and goes to a physical or simulated typewriter-like terminal.

Graphical User Interface (GUI): User input is via a pointing device (mouse), button clicks, and keyboard. Output is graphical and goes to a window on the screen.

Interfaces for C++

The C++ standard specifies a command line interface: `iostream` and associated packages. No similar standard exists for GUIs.

De facto GUI standards in the Linux world are `GTK+` (used by the Gnome desktop) and `Qt` (used by the KDE desktop).

`GTK+` is based on C; `Qt` is based on an extension of `C++` and requires a special preprocessor.

`gtkmm` is a `C++` wrapper on top of `GTK+`.

Advantages: Provides type safety, polymorphism, and subclassing.
Provides a native interface to C++ code.

Disadvantages: Components not so well integrated.
Documentation spread between `gtkmm`, `gtk+`, and other components but improving.

Overall Structure of a GUI

A GUI manages one or more *windows*.

Each window displays one or more *widgets*.
These are objects that provide graphical and textual input and output to the program.

A GUI package such as `gtkmm` maintains a *widget tree*.

A widget controls a particular kind of user input or output.
Examples: label, text box, drawing area, button, scroll bar, etc.

Concurrency and Events

The central problem in building a GUI is handling concurrency.

Data arrives from multiple concurrent sources – mouse and keyboard, network, disk, other threads, etc.

We call the arrival of a piece of data an **event**.

- Event arrival times are unpredictable.
- Events should be processed as quickly as possible.

For example, to have a good interactive feel, the GUI should respond to a mouse click event within milliseconds.

Event Loop

An **event loop** allows a single thread to manage a set of events.

At some level, the hardware or software **polls** for events.

When an event is detected, it is **dispatched** to an **event handler**.

The event handler either processes the event itself, queues a task for later processing, or spawns a thread to process it.

While the event thread is processing one event, no other events can be processed, so event handlers should be short.

Problem is to prevent a long-running low-priority event handler from delaying the handling of a high-priority event.

A GUI event structure

A GUI typically translates raw hardware events into semantically-meaningful software events.

For example, a mouse click at particular screen coordinates might turn into a button-pressed event for some widget in the GUI tree.

Several system layers may be involved in this translation, from the kernel processing of hardware interrupts at the bottom level, up through device drivers, windowing systems such as X, and finally a GUI frameworks such as GTK+.

Interface between user and system code

A major software challenge is how to design the interface between the GUI and the user code that ultimately deals with the events.

In a command-line interface, the user code is at the top level. It connects to the lower layers through familiar function calls.

With a GUI, things are turned upside down.

- The top level is the main event loop.
- It connects to the user by calling appropriate user-defined functions.

Binding system calls to user functions

How can one write the GUI to call user functions that did not even exist when the GUI system itself was written?

The basic idea is that of **interface**.

- The interface is a placeholder for the eventual user functions.
- It describes what functions the user will provide and how to call them but not what the functions themselves are.
- The interface is bound to user code either when the user code is compiled or dynamically at runtime.

Polymorphic binding

C++ virtual functions provide an elegant way to bind user code to an interface.

- The GUI can provide a virtual default event handler.
- The user can override the default handler in a derived class.

Of course, the actual binding occurs at run time through the use of type tags and the `vtable` as we have seen before.

Binding through callback registration

The user explicitly **registers** an event handler with the GUI by calling a special registration function.

- The GUI keeps track of the event handler(s) registered for a particular event.
- When the event happens, it calls all registered event handlers.

This is sometimes called a *callback* mechanism since the user asks to be called back when an event occurs.

Callback using function pointers: GUI side

Callbacks can be done directly in C . Here's the GUI code:

1. Define the signature of the handler function:
   ```
   typedef void (*handler_t)(int, int);
   ```
2. Declare a function pointer in which to save the handler:
   ```
   handler_t buttonPressHandlerPtr;
   ```
3. Define a registration function:
   ```
   void systemRegister(int slot, handler_t f) {
     button_press_handler_ptr = f;
   }
   ```
4. Perform the callback:
   ```
   buttonPressHandlerPtr(23, 45);
   ```

Callback using function pointer: User side

Here's how the user attaches a handler to the GUI:

1. Create an event handler:
   ```
   void myHandler(int x, int y) {
     printf("My handler (%i, %i) called\n", x, y);
   }
   ```
2. Register the handler for event 17:
   ```
   systemRegister(17, myHandler);
   ```

Type safety

The above scheme does not generalize well to multiple events with different signatures.

- Registered handlers need to be stored in some kind of container.
- For type safety, each different handler signature requires a different event container and registration function of the corresponding signature.

The alternative is for `systemRegister()` to take a `void*` for its second argument and to cast function pointers before call them.

This is not type safe and can lead to subtle bugs if the wrong type of function is attached to a callback slot.

Signals and slots

Signals and slots is a more abstract way of linking events to handlers and can be implemented in a type safe way.

- A `connect()` template function is used to bind a `signal` to a `slot`.
- An event `emits` a signal.
- A handler is associated with a slot.
- Whenever the event occurs, the functions associated with all connected slots are called.

Several signals can be connected to the same slot, and several slots can be connected to the same signal.

The `gtkmm` Framework

Structure of `gtkmm`

`gtkmm` relies on several libraries and packages:

- `gtkmm-3.0` is the GUI engine.
- `gdkmm` is a device layer used by `gtk`.
- `cairomm` is a vector graphics drawing package.
- `pango` is a library for laying out and rendering of text, with an emphasis on internationalization.
- `sigc++` is a library for connecting events (signals) to event handlers (slots).

Compiling a `gtkmm` program

Many include files and libraries are needed to compile and build a `gtkmm` program.

A utility `pkg-config` is used to generate the necessary command line for the compiler.

> pkg-config gtkmm-3.0 --cflags

```
-pthread
-I/usr/include/gtkmm-3.0 -I/usr/lib64/gtkmm-3.0/include
-I/usr/include/atkmm-1.6 -I/usr/include/gtk-3.0/unix-print
-I/usr/include/gdkmm-3.0 -I/usr/lib64/gdkmm-3.0/include
-I/usr/include/giomm-2.4 -I/usr/lib64/giomm-2.4/include
-I/usr/include/pangomm-1.4 -I/usr/lib64/pangomm-1.4/include
-I/usr/include/glibmm-2.4 -I/usr/lib64/glibmm-2.4/include
-I/usr/include/gtk-3.0 -I/usr/include/at-spi2-atk/2.0
-I/usr/include/at-spi-2.0 -I/usr/include/dbus-1.0
-I/usr/lib64/dbus-1.0/include -I/usr/include/gtk-3.0
-I/usr/include/gio-unix-2.0/ -I/usr/include/cairo
-I/usr/include/pango-1.0 -I/usr/include/harfbuzz
-I/usr/include/pango-1.0 -I/usr/include/atk-1.0
-I/usr/include/cairo -I/usr/include/cairomm-1.0
-I/usr/lib64/cairomm-1.0/include -I/usr/include/cairo
-I/usr/include/pixman-1 -I/usr/include/freetype2
-I/usr/include/libpng16 -I/usr/include/freetype2
-I/usr/include/libdrm -I/usr/include/libpng16
-I/usr/include/sigc++-2.0 -I/usr/lib64/sigc++-2.0/include
-I/usr/include/gdk-pixbuf-2.0 -I/usr/include/libpng16
-I/usr/include/glib-2.0 -I/usr/lib64/glib-2.0/include
```

Linking a gtkmm program

```
> pkg-config gtkmm-3.0 --libs
```

generates the necessary linker flags.

```
-lgtkmm-3.0 -latkmm-1.6 -lgdkmm-3.0 -lgiomm-2.4
-lpangomm-1.4 -lglibmm-2.4 -lgtk-3 -lgdk-3
-lpangocairo-1.0 -lpango-1.0 -latk-1.0 -lcairo-gobject
-lgio-2.0 -lcairomm-1.0 -lcairo -lsigc-2.0
-lgdk_pixbuf-2.0 -lgobject-2.0 -lglib-2.0
```

To use package config, use the backquote operator on the `g++` command line:

Compiling: `g++ -c ‘pkg-config gtkmm-3.0 --cflags‘ ...`
Linking: `g++ ‘pkg-config gtkmm-3.0 --libs‘ ...`
Both: `g++ ‘pkg-config gtkmm-3.0 --cflags --libs‘ ...`

Using a GUI

The following steps are involved in creating a GUI using `gtkmm`:

1. Initialize `gtkmm`.
2. Create a window.
3. Create and lay out widgets within the window.
4. Connect user code to events.
5. Show the widgets.
6. Enter the main event loop.

The GUI then displays the window and waits for events.
When an event occurs, the corresponding user code is run.
When the event handler returns, the GUI waits for the next event.

Example: `clock`

The code example `25-clock` is a significant extension of the clock example in the `gtkmm` tutorial book.

It illustrates many of the features of `gtkmm`.

Main program

```
#include <gtkmm.h>
#include "clockwin.hpp"

int main(int argc, char* argv[])
{
   auto app =
     Gtk::Application::create(argc, argv, "org.gtkmm.examples");
   ClockWin win;
   return app->run(win);
}
```