DEVELOPING C MODULES WITH GUIS

Using FLTK and GLUI Libraries

José M. Garrido
Department of Computer Science

September 2014

College of Science and Mathematics
Kennesaw State University

© 2014, J. M. Garrido
1 Introduction

A program is usually partitioned into several modules. This is known as modular design. A function is the most basic decomposition unit in a C program and it carries out a specific task in a program. In object oriented programming languages, such as C++ and Java, the fundamental module is a class. A module in C is a related collection of C functions. These functions will normally be stored in a single C source file. Every module has a specific purpose and may be implemented in a specific programming language.

This report provides details on implementing C and C++ modules and the appropriate linkage that results in a single executable file or program. One of the higher–level modules implements a graphical user interface (GUI).

2 Modular Design

A problem is often too large and complex to deal with as a single unit. In problem solving and algorithmic design, the problem is partitioned into smaller problems that are easier to solve. The final software solution consists of an assembly of these smaller solutions. The partitioning of a problem into smaller parts is known as decomposition. These small parts are known as modules, which are much easier to develop and manage.

System design usually emphasizes modular structuring, also known modular design, and modular decomposition. With this approach, the solution to a problem consists of several smaller solutions corresponding to each of the subproblems. A problem is divided into smaller problems (or subproblems), and a solution is designed for each subproblem. These modules are considered building blocks for constructing larger and more complex algorithms.

Modularity and abstraction improve the overall design because a module is much simpler (and smaller) than the overall algorithm; the implementation is also simpler and less prone to errors. The modules have the potential of being reused in other programs or applications.

Additional advantage of modularity is: easy to modify and maintain, because the modules are simpler than the overall algorithm, they are more clearly understood, the implementation and testing is easier to manage.

Modules are implemented in various forms depending on programming language used. Some of these forms are: files, libraries, packages, classes, procedures, subroutines, functions, and macros. A computational model implemented in the C
programming language is decomposed into program files and functions.

Modular design facilitates algorithmic and computational model development and is particularly important in large and complex problems. Abstraction is an important aspect of modularity because each module represents a separate and high-level functionality in the overall design of the algorithm. Each module is defined on a single concept (a single functional aspect) and is independent as possible from other modules.

3 Fundamental Principles of Modular Design

- **Separation of Concerns.** Each module should address a separate concern (one aspect of functionality).

- **Coupling** — a high-level description of the dependency of a module on other modules. This description the interfaces. A module should exhibit loose coupling.

- **Cohesion** — the level of dependency within a module. A module should exhibit strong cohesion.

4 Modular Design with Computational Models

Design and implementation of a large and complex problem can be broken down into several levels of modules. A top-down approach or hierarchical approach is used, by first defining the upper-level modules, then the modules in the next level down, and so on. Each module is defined by two or more submodules at the next level down. Another way to state this principle is by saying that each module is refined in terms of two or more submodules; this principle is known as stepwise refinement.

A program that implements a computational model is partitioned into several modules at various levels of detail. At the highest level of abstraction, only the top modules are defined. This level is adequate when a program includes a GUI module for user interaction (front–end module), the main logic, and the module for the data visualization of the output data produced by the main logic module. This modular structure is shown in Figure 1.

At the next level of abstraction, more detailed decomposition is applied, as shown in Figure 2. The main module has four subordinate modules: 2a, 2b, 2c, and 2d. A module will call or invoke a subordinate module and optionally pass data to it. The main module invokes its subordinate modules in some predefined order. Module 2a
has two subordinate modules: 3a and 3b. Some modules may not have subordinate modules.

Figure 2: Modular structure of a computational model.

Modules communicate data via their interfaces. Every module has its own interface that specifies the type and amount of data that can be transferred to and from the module and how the various functions can be invoked by other modules. With the C programming language, the interface of a module is typically specified in one or more header files.

5 Toolkits for Building GUIs

There are several good GUI toolkits or packages and most are implemented in C or C++. Not all of these packages are based on the OPENGL standard. Some of the most widely-used toolkits for GUI implementation are:

- GLUI – A very small and simple package that is implemented on top of
GLUT/OPENGL.

- GLUI2 – A recent re-development of GLUI
- FLTK – A slightly larger package that is also based on OPENGL.
- Other similar packages are GLFW, GLOW, FOX, and GLV.
- More complete, larger, and more sophisticated packages are: WxWidgets (Wxwindows), GTK+, and Qt.
- Other packages are more geared toward game development, such as SDL.

6 Implementing Graphical User Interfaces (GUIs)

In this section, two relatively small packages, GLUI and FLTK, are used to build GUIs and add them to main logic modules.

This section shows the techniques that can be used to implement the main logic module with the C programming language and the GUI module with the C++ programming language. The reason for this is that the GLUI and FLTK packages each consist of a collection of C++ classes.

This is basically an example of calling C++ functions called by C functions and some C functions called by C++ functions. Because the main module is implemented in C, it must include the declarations of the external functions it calls that are located in the GUI module.

6.1 Using GLUI

The following declarations appear in the main module.

```c
// These functions are defined in the GUI module
// functions to get input data from GUI module
extern int get_numpts();
extern double get_height();
extern double get_tlow();
extern double get_thigh();

// functions to send output data to GUI module
extern void set_completed_jobs(int comp_jobs);
extern void set_rejected(int rejected);
extern void set_avgwait(double avwait);
extern void set_cpuutil(double cutilp);
extern void set_diskutil(double dutil);
```
// initial setup of GUI and get the input data
extern void setupg(const char *compname);
// display the computed output data
extern void compmainb();

void compmain(); // called by GUI module

The GUI module is implemented in C++ and must include the prototypes for
the functions listed previously with the extern "C" phrase. This tells the linker
that these the C calling convention must be used for these functions.

A simplified example of complete C code of the main module and C++ of the
GUI module is stored in files genmod.c and modgui.cpp. These two files are com-
plied separately with the C compiler and with the C++ compiler respectively. The
linkage must be done with the C++ compiler.

1. Execution of the program starts in function main, which is located in the main
module, genmod.

2. Function setupg is called in module genmod. This function is located in the
GUI module, genmodgui. This function sets up the GUI and displays the first
window on the screen. The user may start entering appropriate values into
the input fields.

3. When the user clicks the Compute button, the GUI module calls function
compmain, which is located in the main module (genmod). This function per-
forms the following sequence of tasks:

   (a) Gets the values of the input fields that were entered by the user.
   (b) Computes the results.
   (c) Sends the appropriate output values (if any) to the GUI module, by
calling function compmainb.
   (d) Signals the GUI module to display the second window with the output
   values.

4. When the user clicks the Quit button, the program terminates.

When the program starts execution, it displays the first window on the screen,
which is shown in Figure 3(a). After the user enters the input values, he or she
clicks the Compute button and then the second window is displayed on the screen.
This second window is shown in Figure 3(b).

The commands for compiling and linking the two modules are:
C Modules GUI

(a) Window for input data and control

(b) Window for output data

Figure 3: Windows using GLUI

\$ gcc -c -Wall genmod.c
\$ g++ -c -Wall genmodgui.cpp -I /usr/include/GL
\$ g++ genmod.o genmodgui.o -lglui -lglut -lm

The default name of the executable file is `a.out`, and program starts execution after typing the command:

\$ ./a.out

6.2 Using FLTK

The same general procedure used previously with the GLUI package, applies when implementing GUIs using the FLTK package. In the example described in this section, the main difference is the list of names of the functions to get the input values from the GUI module and the names of the functions to set the output values of the results.

The real difference appears in the GUI module. The two windows displayed are shown in Figure 4(a) and Figure 4(b).

The commands for compiling and linking the two modules using FLTK are:

\$ gcc -c -Wall fgenmod.c
\$ g++ -c -Wall fmodgui.cpp
\$ g++ fgenmod.o fmodgui.o -lfltk -lm
The default name of the executable file is *a.out*, and program starts execution after typing the command:

```
$ ./a.out
```

Instead of using the commands mentioned previously for compiling and linking the files, the recommended procedure is to use a *makefile*. This is explained after the next section.

### 7 Using the `tar` Command to Process Archives

Several files can be stored in a single archive file, known in Unix/Linux as a *tar* file. An additional advantage is that the archive file can be compressed using several techniques to reduce the overall size of the archive.

The compressed archive, `ffallobj3.tar.gz`, includes the following files: `ffallobj3g.c`, `compgui.cpp`, `ffallobj3g.mak`, `ffallobj3g.o`, `compgui.o`, and `ffallobj3g`. These files make up the project of the Free-falling object computational model, which uses the GLUI package.

To extract the files from the archive `ffallobj3.tar.gz`, type the following command on an appropriate directory:

```
$ tar -zxvf ffallobj3.tar.gz
```
The options used with the tar command are: decompress (z), extract (x), verbose mode (v), the following specified file (f).

In a similar manner, the archive genmod.tar.gz includes the following files: genmod.c, modgui.cpp, genmos.mak, genmod.o, modgui.o, and genmod. These files make up the project of the generic computational model, which uses the GLUI package.

The archive fgenmod.tar.gz includes the following files: fgenmod.c, fmodgui.cpp, fgenmod.mak, fgenmod.o, fmodgui.o, and fgenmod. These files make up the project of the generic computational model, which uses the FLTK package.

8 Using make Command to Build Projects

Developing programs that consist of several source files can involve significantly a large amount of work from the point of view of just compiling and linking with external libraries. The files of a program are organized as a project and specified in a makefile, which is used by the make command to build the application (or project).

The makefile includes a set of rules of how to compile the source files and the dependencies of all the source and object files. Every rule specifies the target, a dependency list and a sequence of commands.

These rules indicate which source files need to be compiled or recompiled, and when is linking necessary to build the application. The following is a makefile of the generic computational models that uses the GLUI package. This file is used to build the genmod computational model, into the executable file genmod.

```bash
# make file for a generic computational model that
# links with a GUI module using GLUI
# J. M. Garrido (September, 2013)

BLDIR=/comp_models/basic
OBJs=genmod.o modgui.o
LIBS=-lbasic_lib -lglui -lglut -lGLU -lGL -lm

genmod: $(OBJs)
g++ -o genmod $(OBJs) -L $(BLDIR) $(LIBS)
genmod.o: genmod.c
gcc -c -Wall genmod.c -I $(BLDIR)

modgui.o: modgui.cpp
g++ -c -Wall modgui.cpp -I /usr/include/GL
```

To process this file, start the make command. The first time this command
executes, it will compile the source files (with gcc and g++) and link them with the specified libraries.

The *makefile* has the rules necessary to build this application only. A different project will have its own *makefile*, which might be assigned a different name, and/or be located in a different directory. The following command starts the *make* command using a makefile with the name *mymakef*.

```
$ make -f mymakef
```

Most of the integrated developments environments (IDEs) use projects files to organize the various source and binary files in a specific application or program. Depending on the IDE, the project file may be compatible or can be converted to a standard make file.

The files `ffallobj3.tar.gz`, `genmod.tar.gz`, and `fgenmod.tar.gz` are stored on the web site:  
[science.kennesaw.edu/~jgarrido/comp_models](http://science.kennesaw.edu/~jgarrido/comp_models)

These compressed archives include the source, binary, executable, and make files for the three computational models mentioned previously.