Using C++ Code in a C Project

Last update: July 26, 2018

Table of Content

Writing a C++ Module Ready for C Use 1
Compile and Linking Modules in Makefile 2
Attachment: The Complete Makefile 4

During the development of a C project, one may encounter a situation when a C++ library is needed for the project or writing a C++ module in the project, but it cannot be achieved by default without specific modifications. This guide will walk you through the steps needed to invoke c++ functions in a C project.

Writing a C++ Module Ready for C Use

The following is an example module whose implementation is in C++ using STL libraries but ported for C use.

util.h

```c
#ifndef UTIL_H
#define UTIL_H

#ifdef __cplusplus
extern "C" {
#endif

long long Util_getCurrentTime();
void Util_InvokeWithDelay(long delayTime, void (*callback)(void));

#ifdef __cplusplus
}
#endif
#endif
```

```c
```
util.cpp
#include <chrono>
#include <thread>
#include "util.h"

extern "C"
long long Util_getCurrentTime() {
    using namespace std::chrono;
    using cast = duration<long long>;
    return duration_cast<cast>(system_clock::now().time_since_epoch()).count();
}

extern "C"
void Util_InvokeWithDelay(long delayTime, void (*callback)(void)) {
    // Create a new thread for delayed queueing wavedata
    std::thread([=]{
        using namespace std;
        this_thread::sleep_for(chrono::milliseconds(delayTime));
        callback();
    }).detach();
}

The change one may need to make to the C++ module is highlighted in yellow. Here is the explanation:

1. **What to do:** We need to wrap function declarations with extern "C" with brackets or in front of each function declaration in both cpp and header files.
   
   **Reason:** Since C++ has function overloading, without specifying extern "C", the compiler will apply some small changes to function declarations and it will not be possible for C code to link the C++ functions any more. Adding extern "C" will force the C++ compiler not to mangle the function name so that the compiled C++ code can be linked with C code.

2. **What to do:** We also need to use #ifdef __cplusplus and #endif to wrap extern "C".
   
   **Reason:** This step is optional if you decide to port your module purely for C usage. However, without the conditional check, the module written in C++ cannot be used by other C++ libraries since it is expected for the compiler to mangle function declarations for C++ linkage. We would want the same module to be shared by C and C++ modules, and hence, adding the conditional check to apply extern “C” depending on whether the module is a C one or a C++ one.

**Compile and Linking Modules in Makefile**

C++ source code is supposed to be compiled by the program g++ while C source code is handled by gcc. In a project using mixed and match C/C++ modules, compiling and linking with different flags can be complicated.
1. We need to download g++ compiler first depending on whether you have it already installed or you want the cross tool version for ARM processors.

$ sudo apt install g++-arm-linux-gnueabihf (Cross tool version)
$ sudo apt install g++ (Host version)

2. Inside a Makefile, using wildcard command to filter out all files with extension .c and also create a similar variable to filter out .cpp files.

```
C_FILES = $(wildcard *.c)
CPP_FILES = $(wildcard *.cpp)
```

3. Create pattern matching rules that will be applied to .c and .cpp files separately. The official documentation explaining pattern rules is here.

$< here is an automatic variable provided by GNU Make referring to the dependencies on the right hand side of the colon, and $@ is another automatic variable referring to the target of the rule, left hand side of the colon.

```
# Pattern to match cpp files and use g++ to compile
$(OBJDIR)/%.o: %.cpp
   $(CC_CPP) $(CPPFLAGS) -c $< -o $@

# Pattern to match c files and use gcc to compile
$(OBJDIR)/%.o: %.c
   $(CC_C) $(CFLAGS) -c $< -o $@
```

4. Finally, you can make a rule which depends on all the object files, which will force GNU Make to go through the previous 2 pattern rules first to generate object files.

```
C_OBJS = $(addprefix $(OBJDIR)/, $(C_FILES:.c=.o))
CPP_OBJS = $(addprefix $(OBJDIR)/, $(CPP_FILES:.cpp=.o))
compile: $(C_OBJS) $(CPP_OBJS)
   $(LD) ^ -o $(OUTDIR)/$(TARGET) $(LFLAGS)
```

C_OBJS and CPP_OBJS are constructed using C_FILES and CPP_FILES defined above by replacing the extensions .c and .cpp to .o and also adding object folder path in the front. The .o object files generated will now go to obj/ folder. The compile rule will use linker (in this case, g++) to link all object files to generate an executable. The automatic variable $^ means taking all dependencies on the right hand side of the colon.
# Makefile for building embedded application.

# Edit this file to compile extra C files into their own programs.
TARGET = miku

OBJDIR = obj
LIBDIR = libs
OUTDIR = $(HOME)/cmpt433/public/myApps

C_FILES = $(wildcard *.c)
C_OBJS = $(addprefix $(OBJDIR)/, $(C_FILES:.c=.o))

CPP_FILES = $(wildcard *.cpp)
CPP_OBJS = $(addprefix $(OBJDIR)/, $(CPP_FILES:.cpp=.o))

CROSS_TOOL = arm-linux-gnueabihf-

# Compiler
CC_CPP = $(CROSS_TOOL)g++
CC_C = $(CROSS_TOOL)gcc

# Linker
LD = $(CROSS_TOOL)g++

CFLAGS = -Wall -g -std=c99 -D _POSIX_C_SOURCE=200809L -Werror
CPPFLAGS = -Wall -g -std=c++11 -Werror

all: compile

compile: $(C_OBJS) $(CPP_OBJS)

$(LD) $(^) -o $(OUTDIR)/$(TARGET)

# Pattern to match cpp files and use g++ to compile
$(OBJDIR)/%.o: %.cpp

 $(CC_CPP) $(CPPFLAGS) -c $< -o @

# Pattern to match c files and use gcc to compile
$(OBJDIR)/%.o: %.c

 $(CC_C) $(CFLAGS) -c $< -o @

clean:
rm -rf $(OBJDIR)
rm -f $(OUTDIR)/$(TARGET)