PG - DESD
Batch – Sept 2021
Module – Embedded C Programming

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Mobile No - 9890662093

Function Calling Conventions

- How functions are called on particular CPU architecture?
  - How arguments are pushed on the stack? (left to right or right to left).
  - Who pop arguments from the stack? (calling function or called function)
- How assembly code is generated by the compiler to call a function?
- Calling conventions depends on CPU architecture.
  - **x86 architecture**
    - pascal
    - cdecl
    - stdcall
  - **ARM architecture**
    - AAPCS or ATPCS

- **Pascal Calling Convention**
  - Outdated. Was supported in Turbo C compiler.
  - arg push order: left to right
  - stack cleanup: called function
- **cdecl calling convention**
  - C Declarator. Default calling convention for C programs.
  - arg push order: right to left
  - stack cleanup: calling function
- **stdcall calling convention**
  - Standard Call. Used in some technologies like COM.
  - arg push order: right to left
  - stack cleanup: called function
  - Generated assembly code is more compact (when same function is called multiple times).

Preprocessor Directives

- Preprocessor is part of C programming toolchain/SDK.
  - Removes comments from the source code.
  - Expand source code by processing all statements starting with #.
  - Executed before compiler
- All statements starting with # are called as preprocessor directives.
  - Header file include
    - #include
  - Symbolic constants & Macros
    - #define
  - Conditional compilation
    - #if, #else, #elif, #endif
    - #ifdef #ifndef
  - Miscellaneous
    - #pragma, #error

#include

- #include includes header files (.h) in the source code (.c).
- #include <file.h>
  - Find file in standard include directory.
  - If not found, raise error.
- #include "file.h"
  - File file in current source directory.
  - If not found, find file in standard include directory.
  - If not found, raise error.

#define (Symbolic constants)

- Used to define symbolic constants.
  - #define PI 3.142
  - #define SIZE 10
- Predefined constants
  - __LINE__
  - __FILE__
  - __DATE__
  - __TIME__
- Symbolic constants and macros are available from there declaration till the end of file. Their scope is not limited to the function.

#define (Macro)

- Used to define macros (with or without arguments)
  - #define ADD(a, b) (a + b)
  - #define SQUARE(x) ((x) * (x))
  - #define SWAP(a,b,type) { type t = a; a = b; b = t; }
- Macros are replaced with macro expansion by preprocessor directly.
  - May raise logical/compiler errors if not used parenthesis properly.
- Stringizing operator (#)
  - Converts given argument into string.
  - #define PRINT(var) printf(#var " = %d", var)
- Token pasting operator (##)
  - Combines argument(s) of macro with some symbol.
  - #define VAR(a,b) a##b

#define

- Functions
  - Function have declaration, definition and call.
  - Functions are called at runtime by creating FAR on stack.
  - Functions are type-safe.
  - Functions may be recursive.
  - Functions called multiple times doesn't increase code size.
  - Functions execute slower.
  - For bigger reusable code snippets, functions are preferred.

- Macros
  - Macro definition contain macro arguments and expansion.
  - Macros are replaced blindly by the processor before compilation
  - Macros are not type-safe.
  - Macros cannot be recursive.
  - Macros (multi-line) called multiple times increase code size.
  - Macros execute faster.
  - For smaller code snippets/formulas, macros are preferred.

Conditional compilation

- As preprocessing is done before compilation, it can be used to control the source code to be made available for compilation process.
- The condition should be evaluated at preprocessing time (constant values).
- Conditional compilation directives
    - #if, #elif, #else, #endif
    - #ifdef, #ifndef
    - #undef

```
#define VER 1
int main() {
    #ifndef  VER
        #error "VER not defined"
    #endif
    #if  VER == 1
         printf("This is Version 1.\n");
    #elif  VER == 2
         printf("This is Version 2.\n");
    #else
        printf("This is 3+ Version.\n");
     #endif
     return 0;
}
```

GCC - GNU C Compiler

- Set of tools/programs used to compile C program.
  - These tools are used to develop C programs and SDK (Software Development Kit).
  - Many of these tools are used in sequence and also called as tool-chain.

- Tools
  - Pre-processor (cpp)
  - Compiler (cc1)
  - Assembler (as)
  - Linker (ld)
  - Debugger (gdb)
  - Objdump (objdump)
  - etc.

- "gcc" is front-end for compilation & linking tools.

- gcc internally invokes Pre-processor, Compiler, Assembler and Linker.
  - gcc -E --> Pre-procssor
  - gcc -c --> Compiler
  - gcc -S --> Assembler
  - gcc --> Linker

"gcc" options

- -o output_file --> give output file name.
- -E --> show Pre-procssor output
- -c --> Compile only (.o)
- -S --> Create assembly output file (.s)
- -std --> specify C standard
  - -std=c89 --> ANSI standard
  - -std=c99 --> ANSI standard
  - -std=gnu89 --> C89 with GNU extensions
  - -std=gnu99 --> C99 with GNU extensions (used in Linux device driver development)
- -g --> Debugger level (Higher level --> Higher debug info --> Higher .out file size)
  - -ggdb1
  - -ggdb2 (default)
  - -ggdb3
- -Wxxx --> Warning flags
  - -Wall --> show all warnings
  - -Werror --> treat warning as error (do not create .out file)
- -Ox --> Optimization
  - -O0 --> No optimization
  - -O1
  - -O2
  - -O3 --> Highest optimization
  - -Os --> Optimization for size (compact low level code generated)
- -D --> define symbol, symbolic constant or macro
  - -DPI=3.142
  - -D'BV(n)=(1<<(n))'
- -I --> Include standard dir path.
  - -I/usr/include --> find standard header files into "/usr/include“
  - -I. --> find standard header files into current directory
  - #include <file.h> --> will be searched in standard directory (or -I dirpath)
- -L --> Library standard dir path
  - Standard library: libc.so (by default linked) --> -lc
  - Math library: libm.so (need to link separately) --> -lm
  - Standard libraries are available in /usr/lib (depends on Linux).
  - -L/usr/lib --> file .so/.a files into "/usr/lib".

Debugging

- Debugging is process of finding bugs (logical errors) in the programs.

- It also helps understanding flow of execution of the program.

- Debugger needs symbol & source code info to be present in executable file.
  - Need to compile program so that debugging can be done.
  - -g --> enable debugging (add symbol & source code info in executable file).

- Debugger enable executing the program step by step and monitor values of each variable.

- Debugger in GCC tool-chain is "gdb".

Debugging Steps

- step 1: Compile program to enable debugging.
  - gcc -g
  - Makefile: CFLGAS = -g
- step 2: Start debugger.
  - gdb main.out
- step 3: Give gdb commands to debug step by step.
  - Set a breakpoint (point from which you want to debug step by step).
    - break file.c line_number
    - break function_name
  - Start debugging process (it will auto stop on first breakpoints)
    - run
- Execute step by step
  - next - execute the function but do not show fn code line by line (Step Over)
  - step - execute the function line by line (Step Into)
  - cont - execute directly till next breakpoint
- Monitor variables
  - display varname - print var contents after each step
  - print varname - print var content once
  - Backtrace
- Source code
  - list - show 10 lines of code
- Stop debugging
  - quit

Static and Dynamic Linking of Libraries

- process of collecting and combining multiple object files to create a final executable.
- Linking can be performed at
  - **Compile time** – when machine code is generated from source code **(Static Linking)**
  - **Runtime time** – when program is loaded into memory **(Dynamic Linking)**

- **Static linking**
  - all library modules used in the program are copied into final executable file.
  - Performed by linker and is last step of compilation.
  - Executable size is larger comparatively.

- **Dynamic linking**
  - Linking of all library modules is performed on the fly as program starts running on the system.
  - Name of the shared library is placed in the final executable file.
  - Actual linking takes place at run time (both executable file and library are loaded in the memory).
  - Executable size is reduced.

Steps to create static libraries

1. Create header files and source files for your library. (create mymath.h, add.c, sub.c)
2. Compile all source files.
   - gcc -c sub.c
   - gcc -c add.c
3. Create a static library by combining all object files.
   - ar rs mymath.a add.o sub.o
4. Write a program which will use functions of your library. (create demo.c)
5. Compile your program
   - gcc -I . -c demo.c
6. Link your program with static library to get final executable file.
   - gcc -o demo demo.o libmymath.a
   - gcc –o demo -L . demo.o –lmymath
7. Run your program

Steps to create shared libraries

shared library (on Linux) or a dynamic link library (dll on Windows)

1. Create header files and source files for your library. (create mymath.h, add.c, sub.c)
2. Compile all source files.
   - gcc -fPIC -c sub.c
   - gcc -fPIC -c add.c
3. Create a shared library by combining all object files.
   - gcc -shared -o libmymath.so add.o sub.o
4. Install shared library
   - Add your library in standard directory and run command ldconfig
5. Write a program which will use functions of your library. (create demo.c)
6. Compile your program
   - gcc -c demo.c
7. Link your program with static library to get final executable file.
   - gcc -o demo demo.o libmymath.so
   - gcc -o demo demo.o –lmymath
8. Run your program

Thank you!

Devendra Dhande <devendra.dhande@sunbeaminfo.com>