REVIEW OF C PROGRAMMING

Instructor: Dr. Sukhjit Singh Sehra

Execution of the code

- We will use GNU Compiler Collection (GCC).
- It is a robust, open-source compiler system supporting various programming languages, but it's most notably known for its support for C and C++.
- GCC is widely used for compiling C programs due to its efficiency, portability, and extensive support for various standards and optimizations.

Execution of the code

- In C programming, GCC serves as a critical tool to translate human-readable C code into machine code.
- The compiler takes the source code written in C and converts it into executable binary code that a computer can run.
- This process involves several steps, including preprocessing, compilation, assembly, and linking.

Compiling the Program

```c
#include  <stdio.h>
int main (void)
{
printf ("Hello, world!\n");
return 0;
}
```

- Compile the program: `gcc hello.c -o hello`.
- Here, `hello.c` is the source file, `-o hello` specifies the output executable name.

Running the Compiled Program

- Run the program: `./hello`.
- `./` is used to execute a file in the current directory.

Multistep program execution

Compiling multiple source files:

1. In the following example we will split up the program Hello World into three files: 'main.c', 'hello_fn.c' and the header file 'hello.h'. Here is the main program 'main.c':

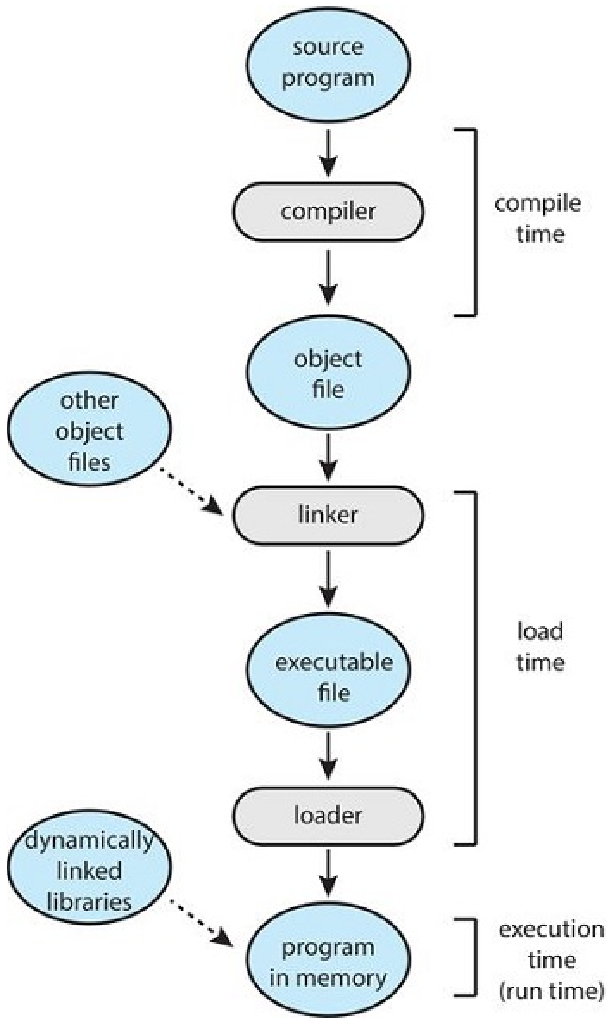

Figure 1: Multistep program execution, adapted from Operating System Concepts by Abraham Silberschatz; Greg Gagne; Peter B

```
#include "hello.h"
int main (void)
{
hello ("world");
return 0;
}
```

- The declaration in ‘hello.h’ is a single line specifying the prototype of the function hello:

```
void hello (const char * name);
```

- The definition of the function hello itself is contained in the file ‘hello_fn.c’:

```
#include <stdio.h>
#include "hello.h"
void hello (const char * name)
{
printf ("Hello, %s!\n", name);
}
```

- Now, run `$ gcc -Wall main.c hello_fn.c -o newhello`

Compiling files independently:

- The command-line option ‘-c’ is used to compile a source file to an object file. For example, the following command will compile the source file ‘main.c’ to an object file:

  `$ gcc -Wall -c main.c`

- This produces the object file ‘hello_fn.o’.

  `$ gcc -Wall -c hello_fn.c`

- Creating executables from object files:

  `$ gcc main.o hello_fn.o -o hello`

Link order of object files

- The object file which contains the definition of a function should appear after any files which call that function. In this case, the file ‘hello_fn.o’ containing the function hello should be specified after ‘main.o’ itself, since main calls hello:

  `$ gcc main.o hello_fn.o -o hello` (correct order)

- With some compilers or linker the opposite ordering would result in an error,

  `$ gcc hello_fn.o main.o -o hello` (in-correct order) main.o: In function ‘main’: main.o(.text+0xf): undefined reference to ‘hello’

Linking with external libraries:

- To avoid the need to specify long paths on the command line, the compiler provides a short-cut option '-l' for linking against libraries. For example, the following command:

  ```
  $ gcc -Wall calc.c -lm -o calc
  ```

- is equivalent to:

  ```
  $ gcc -Wall calc.c /usr/lib/libm.a -o calc
  ```

- Here The library 'libm.a' contains object files for all the mathematical functions, such as sin, cos, exp, log and sqrt. The linker searches through these to find the object file containing the sqrt function.
- In general, the compiler option '-lNAME' will attempt to link object files with a library file 'libNAME.a' in the standard library directories.

Command-Line Arguments in C

- Command-line arguments are values entered by a user when executing a program from a command-line interface.
- These arguments are typed after the program's name in a command prompt within certain execution environments.

Accessing Command-Line Arguments

- The `main()` function can be defined with two parameters: `int argc` and `char* argv[]` to access command-line arguments.
- `argc` (argument count) represents the number of command-line arguments, including the program's name itself.
- `argv` (argument vector) is an array of strings, where each element represents one command-line argument. `argv[0]` is always the program's name.

Example Program

- The program `argtest.c` demonstrates the use of `argc` and `argv`.
- It prints the values of `argc` and each string in `argv[]`.
- The program uses a `for` loop to iterate through the arguments, although understanding `for` loops is not essential for this context.

```
#include "stdio.h"

int main(int argc, char* argv[]) {
   int i;

   // Prints argc and argv values
   printf("argc: %d\n", argc);
   for (i = 0; i < argc; ++i) {
      printf("argv[%d]: %s\n", i, argv[i]);
   }
```

```
    return 0;
}
```

Functionality and Examples

- When `./argtest` is run without additional arguments, `argc` is 1, and `argv[0]` is `./argtest`.
- With additional arguments (e.g., `./argtest Hello`), `argc` increases, and `argv` contains each argument as a separate string.

```
> ./argtest
argc: 1
argv[0]: ./argtest

> ./argtest Hello
argc: 2
argv[0]: ./argtest
argv[1]: Hello

> ./argtest Hey ABC 99 -5
argc: 5
argv[0]: ./argtest
argv[1]: Hey
argv[2]: ABC
argv[3]: 99
argv[4]: -5
```

Example usage

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Usage: program username userage */
int main(int argc, char* argv[]) {
    char nameStr[100];       // User name
    char ageStr[100];        // User age

// Get inputs from command line
    strcpy(nameStr, argv[1]);
    strcpy(ageStr, argv[2]);

    // Output result
    printf("Hello %s. ", nameStr);
    printf("%s is a great age.\n", ageStr);

    return 0;
}
```

Output:

```
> myprog.exe Amy 12
Hello Amy. 12 is a great age.

...

> myprog.exe Jordan 44 HEY
Hello Jordan. 44 is a great age.

...

> myprog.exe Denming
Segmentation fault
```

The last call to the program results in an out-of-range array access.

Checking the number of command-line arguments

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Usage: program username userage */
int main(int argc, char* argv[]) {
   char nameStr[100];       // User name
   char ageStr[100];        // User age

   // Check if correct number of arguments provided
   if (argc != 3) {
      printf("Usage: myprog.exe name age\n");
      return 1; // 1 indicates error
   }

   // Grab inputs from command line
   strcpy(nameStr, argv[1]);
   strcpy(ageStr, argv[2]);

   // Output result
   printf("Hello %s. ", nameStr);
   printf("%s is a great age.\n", ageStr);

   return 0;
}
```

Additional Information

- Command-line arguments in C are C strings.
- The `atoi()` function, included via `#include <stdlib.h>`, converts a C string to an integer.
- Enclosing an argument in quotes allows it to contain spaces.

#define directive

- It is of the form #define MACROIDENTIFIER replacement, instructs the processor to replace any occurrence of MACROIDENTIFIER in the subsequent program code by the replacement text.
- #define is sometimes called a macro. The #define line does not end with a semicolon.

```
#define PI_CONST  3.14159

double CalcCircleArea(double radius) {
   return PI_CONST * radius * radius;
}
```

- Most uses of #define are strongly discouraged by experienced programmers and a better way is defining `const double PI_CONST = 3.14159`

MakeFile and GCC For C programs

- When you are working on C, C++ and others-language and want to compile them from terminal Command.
- Makefile would run and compile your programs more efficiently.
- Makefile helps in reducing the typing again and again, a numbers of source files as well as linker flags, those are required during compilation of the program.

Makefile structure

- Creating a Makefile involves defining rules to automate the build process for C programs. Here's a brief note with an example:
- Makefile Structure: A Makefile typically contains a set of rules. Each rule consists of three parts:
  - A target: The name of the file to be generated.
  - Prerequisites: Files needed to create the target.
  - A recipe: Commands to generate the target.

Example

```
all: program

program: main.o utils.o
    gcc -o program main.o utils.o

main.o: main.c
    gcc -c main.c

utils.o: utils.c
    gcc -c utils.c
```

```
clean:
    rm -f program main.o utils.o
```

Compiling and Usage

- This Makefile compiles a program with `main.c` and `utils.c`.
- The `all` target builds the final executable, `program`.
- The `clean` target removes compiled files.
- Usage:
  - Run `make` to build the executable.
  - Run `make clean` to remove compiled files.

Note: For a detailed guide and more examples, refer to the GNU Make Manual.

C Style Guide

- `Meaningful names for variables, constants and functions. Do not use camelcase;` use underscores for multi-word variables. For example, use hot_water_temperature instead of hotWaterTemperature. Using a variable name with a single character is only appropriate if the variable is the iteration variable of a loop; otherwise using a single-character variable is not correct. For example, if a variable represents a sum, name it `sum` instead of `s`.
- `Good indentation (3 or 4 spaces). Use a proper editor (e.g., emacs, vi) that assists with indentation.`
- `If variables have the same type, declare them on the same line if possible.`
- Leave one blank line between variable declarations and the first line of code in a function.
- Consistent style (e.g., use of curly brackets).
- `Opening brace must be on the same line as conditional or function.`
- `#defined constants must be in uppercase (e.g., #define MAX_LEN 10 and NOT #define max_len 10).`

C Style Guide Contd...

- In your code you should leave one blank space between operators (e.g., x = 5 + 7).
- Leave one space after a comma.
- If the code is too complicated to be read on its own, simplify/split/rename variables.
- `Use braces; avoid loops and conditionals without them.`
- Use parentheses for clarity, especially with bit operations.
- `Avoid global variables where they are unnecessary.`
- `If p is a pointer to a structure, you should access members by using p->member rather than (\*p).member.`

- If p is a pointer to a structure, do not surround the -> operator with spaces (use p->id instead of p -> id).
- You must avoid code duplication.

Code Organization

- Use the universal convention to organize your code:

  1. #include <>
  2. #include “ ”
  3. #defines
  4. Data Types (e.g., structures)
  5. Globals
  6. Prototypes
  7. Code

- The main() function is either first or last.
- #includes and #defines in the middle of code are usually frowned upon.

Functions

- You must avoid code duplication by calling appropriate functions (rather than cutting and pasting code).
- Input parameters must appear before output parameters.
- Annotate helper functions with static.
- Annotate unmodified parameters with const.
- And annotate functions intended to be used from outside with extern.
- Functions should represent a reasonable unit of complexity or be reused frequently.

Comments

- All code must have some comments (there is no such thing as self-documenting code).
- Describe the intent of a block of code.
- A description should appear at the top of each function.