CS429: Computer Organization and Architecture

Intro to C

Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: January 27, 2020 at 13:38

Topics

- Simple C programs: basic structure, functions, separate files
- Compilation: phases, options
- Assembler: GNU style, byte ordering
- Tools for inspecting binary: od, objdump

A Simple C Program

What program should we write first?

A Simple C Program

What program should we write first?

- Hello World: program to print a short message.
- We use the gcc compiler driver to compile the program.
- We assume our target is an x86-compatible machine.
- This program prints “Hello, world!” to its standard output.
- The program text is in file `hello.c`. How'd it get there?

```
/* Hello World program in C */

#include "stdio.h"

int main()
{
   printf("Hello, world!\n");
}
```

File Inclusion

A directive of the form:

`#include "filename"`

or

`#include <filename>`

is replaced (by the preprocessor) with the contents of the file `filename`.

If the filename is quoted, searching for the file begins in the local directory; if it is not found there, or if the name is enclosed in braces, seaching follows an implementation-defined rule to find the file.

Running the Program

Several steps are necessary to run the program.

- Invoke the gcc *compiler driver* to transform your text file (in this case called `hello.c`) into an executable image.
- Then ask the *operating system* to run the executable.

```
> gcc hello.c
> a.out
Hello, world!
>
```

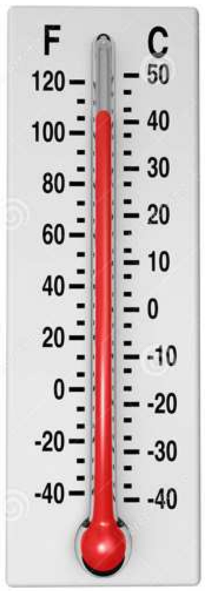
The single call to gcc actually invokes: the preprocessor, the compiler, the assembler, and the linker.

A More Complex Program

```
#include <stdio.h>

/* print Fahrenheit to Celsius [C = 5/9(F-32)]
   for fahr = 0, 20, ..., 300 */

main()
{
  int fahr, celsius;
  int lower, upper, step;

  lower = 0;     /* low limit of table */
  upper = 300;   /* high limit of table */
  step = 20;     /* step size */
  fahr = lower;
  while (fahr <= upper) {
    celsius = 5 * (fahr-32) / 9;
    printf("%d\t%d\n", fahr, celsius);
    fahr = fahr + step;
  }
}
```

Running the Temperature Program

```
> gcc -O2 temperature.c
> a.out
0       -17
20      -6
40      4
60      15
80      26
100     37
120     48
140     60
160     71
180     82
200     93
220     104
240     115
260     126
280     137
300     148
```



About Optimization Levels

Optimization *cannot* change the functional behavior of your program. It offers a tradeoff between execution efficiency, compilation time, and code size.

-O0: fast compilation time, straightforward code (default)

-O1: OK code size, slightly faster execution time

-O: same as -O1

-O2: code may be bigger, faster execution time

-O3: code may be bigger, even faster execution time

-Os: smallest code size

-Ofast: same as -O3, with fast math calculations

-Og: same as -O1, but better for debugging

The faster the execution the bigger and more obscure the code may be.

Specifying an Output Filename

```
> gcc -O2 -o tempConvert temperature.c
> tempConvert
0       -17
20      -6
40      4
60      15
80      26
100     37
120     48
140     60
160     71
180     82
200     93
220     104
240     115
260     126
280     137
300     148
```

TempConvert with For Loop

```
#include <stdio.h>

#define  LOWER 0    /* low limit of table */
#define  UPPER 300  /* high limit of table */
#define  STEP  20   /* step size */

/* print Fahrenheit to Celsius table
   for fahr = 0, 20, ..., 300 */

main()
{
  int fahr;
  double celsius;

  for (fahr = LOWER; fahr <= UPPER; fahr += STEP) {
    celsius = (5.0 / 9.0) * (fahr - 32);
    printf("%3d %6.1f\n", fahr, celsius);
  }
}
```

Running TempConvert2

```
> gcc -o tempConvert2 temp2.c
> tempConvert2
  0  -17.8
 20   -6.7
 40    4.4
 60   15.6
 80   26.7
100   37.8
120   48.9
140   60.0
160   71.1
180   82.2
200   93.3
220  104.4
240  115.6
260  126.7
280  137.8
300  148.9
```

Program with Environment Variables

Sometimes you'd like to get data from the command line, or from the operating environment.

- This program has environment input variables.
- Variables argc and argv reflect the command line.

```
#include <stdio.h>        // for the printf command

main( int argc, char *argv[])
{
  printf("Program has %d command line args.\n", argc);
}
```

```
> gcc countargs.c
> a.out 3 "hello" "why me?" 5
Program has 5 command line args.
> a.out 3 "hello" "why me?" 5 *
Program has 196 command line args.
```

Command Line Arguments

argc is the argument count, including the name of the program. argv is an array of those strings. Those names are conventional.

```
#include <stdio.h>

main( int argc, char *argv[] )
{
  int i;
  if( argc == 1 )
     printf( "The command line argument is:\n" );
  else
     printf( "The %d command line arguments are:\n", argc );

  for( i = 0; i < argc; i++ )
     printf( "Arg %3d: %s\n", i, argv[i] );
}
```

Running the Program

```
> gcc -o commargs commargs.c
> commargs "string with blanks" 1 a b 7.45
The 6 command line arguments are:
Arg   0: commargs
Arg   1: string with blanks
Arg   2: 1
Arg   3: a
Arg   4: b
Arg   5: 7.45
```

Note: Some command line arguments are treated specially by the OS. E.g., "*" expands to a list of files in the current directory.

Environment Variables

Variable env reflects the environment variables. It holds an array of strings maintained by the OS.

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

main( int argc, char *argv[], char *env[])
{
   int i;
   printf( "The environment strings are:\n" );

   i = 0;
   while( env[i] != NULL )
   {
      printf( "Arg %3d: %s\n", i, env[i] );
      i++;
   }
}
```

Note that the env parameter is not in the standard, but is widely supported. To include env, you also must have argc and argv.

Running the Program

```
> gcc -o envargs envargs.c
> envargs
The environment strings are:
Arg   0: PWD=/u/byoung/cs429/c
Arg   1: TERM=dumb
Arg   2: TERMCAP=
Arg   3: COLUMNS=80
Arg   4: EMACS=t
Arg   5: INSIDE_EMACS=23.3.1,comint
Arg   6: SHELL=/lusr/bin/tcsh
Arg   7: GROUP=prof
Arg   8: GPG_AGENT_INFO=/tmp/keyring-hZHfuV/gpg:0:1
# <lots more, 49 in all>
```

Accessing Environment Variables

Once you know the names of Environment variables, you can access them using the `getenv` function from `stdlib`. *You can use getenv even without the env parameter.*

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

int main() {
   char* user = getenv( "USER" );
   if( user != NULL )
       printf( "USER = %s\n", user );
   return 0;
}
```

```
> gcc testgetenv.c
> a.out
USER = byoung
```

The GNU gcc Compiler

gcc is a cross compiler

- It runs on many machines
- Input languages: C, C++, Fortran, Java, and others
- Many target languages: x86, PowerPC, ARM, MC680x0, others

Extensive documentation is available on-line.

verbose mode: gcc works in phases:

```
gcc -v -O2 -o <objectFile> <sourceFile>.c
```

Assembler Output from gcc

You can *stop at assembly* without producing machine code or linking.

```
gcc -S -O2 <sourceFile>.c
```

We'll be doing this a lot this semester!

```
int sum( int x, int y)
{
   int t = x + y;
   return t;
}
```

To generate the assembler in file `sum.s`:

```
gcc -S -O2 sum.c
```

sum.s

```
      .file     "sum.c"
      .text
      .p2align  4,,15
      .globl    sum
      .type     sum, @function
sum:
.LFB0:
      .cfi_startproc
      leal      (%rdi,%rsi), %eax
      ret
      .cfi_endproc
.LFE0:
      .size    sum, .-sum
      .ident   "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4"
      .section .note.GNU-stack,"",@progbits
```

Assembler Output for hello.c

```
> gcc -S -O2 hello.c
> cat hello.s
      .file       "hello.c"
      .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
      .string     "Hello, world"
      .section    .text.startup,"ax",@progbits
      .p2align    4,,15
      .globl      main
      .type       main, @function
main:
.LFB24:
      .cfi_startproc
      movl       $.LC0, %edi
      jmp        puts
      .cfi_endproc
.LFE24:
      .size      main, .-main
      .ident     "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4"
      .section   .note.GNU-stack,"",@progbits
```

Assembler Output from Binary

objdump can be used to disassemble binary output.

```
> gcc -O -c sum.c
> objdump -d sum.o
sum.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <sum>:
   0:   8d 04 37              lea    (%rdi,%rsi,1),%eax
   3:   c3                    retq
```

-c means to compile to machine code, but don't link. This is needed here since there's no `main` function.

Note that the assembly language syntax may differ slightly depending on how you generate it.

What Assembly?

Assembly code doesn't run on any computer! It's the binary code that runs.

Assembly (like high level languages) is just a convenient, humanly readable notation for writing programs. There are two common forms for x86 assembly:

- Gnu Assembly (GAS) format (what we'll be using)
- Intel/Microsoft Assembly format

You may notice variations in assembly code depending on what produced it, even within one format.

Show Bytes Program

```
#include <stdio.h>
typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, int len) {
  int i;
  for ( i = 0; i < len; i++ ) {
    printf("%.2x", start[i]); }
  printf("\n");
}

void main () {
  int i = 15213;
  float f = 15213.0;
  double d = 15213.0;
  int *p = &i;

  show_bytes((byte_pointer) &i, sizeof(i));
  show_bytes((byte_pointer) &f, sizeof(f));
  show_bytes((byte_pointer) &d, sizeof(d));
  show_bytes((byte_pointer) &p, sizeof(p));
}
```

Running show_bytes

Here's how you might compile and run that code:

```
> gcc -o showbytes showbytes.c
> showbytes
 6d 3b 00 00
 00 b4 6d 46
 00 00 00 00 80 b6 cd 40
 48 65 dc 42 ff 7f 00 00
```

Example with Malloc

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

typedef struct node {
  char *word;
  struct node *next;
} NODE;

int main( int argc, char *argv[] )
{
   NODE *newnode;
   NODE *list = NULL;

   int i;
   // Create a linked list containing
   // words from the command line (except program name).
   for ( i = 1; i < argc; i++ )
     {
       newnode = (NODE *)malloc( sizeof(NODE) );
       newnode->word = argv[i];
       newnode->next = list;
       list = newnode; }

   // continues on next slide
```

Example with Malloc (2)

```
   // continued from previous slide

   // Print them out (in reverse order)
   NODE *ptr;
   ptr = list;
   while (ptr)
     {
       printf("%s ", ptr->word);
       ptr = ptr->next;
       }
   printf("\n");

   // need code here to free the list

   exit( i );
}
```

```
> gcc mallocexample.c
> a.out fleas has dog My
My dog has fleas
```

Example with Malloc (3)

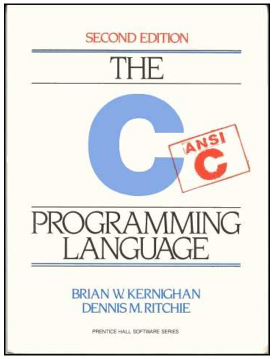
This would be bad programming because it doesn't free the list, resulting in a memory leak.

```
   // Should add this code

   // Free storage
   NODE *ptr2;
   ptr = list;
   while (ptr)
     {
       ptr2 = ptr;
       ptr = ptr->next;
       free(ptr2);
       }

   exit( i );
}
```

Note that valgrind might not indicate that there's a memory leak here because Linux frees all storage when a program terminates.

C Tutorials Available

The C Programming Language, 2nd edition, by Kernighan and Richie is a standard reference. There are versions available on-line.



Google "C tutorial" and you'll find lots of options. For example: `http://wwww.iu.hio.no/~mark/CTutorial/CTutorial.html`