COMP 206 – Introduction to Software Systems

Lecture 6 – C Programming Basics

Septh 19th, 2018

Photo Challenge

Photo Challenge

- Left: Ken Thompson
  - Unix
  - B
  - Go
  - Now at Google
  - Username: ken
- Right: Dennis Ritchie
  - C
  - Unix
  - Lucent Technologies
  - Username: dmr
- Both: Turing Medal, national science award, have changed the world

Hello again, world

```
#include <stdio.h>


void main( )
{
    printf( "Hello, world.\n" );
}
```

```
$ gcc hello.c
$./a.out
Hello, world.
```

The History of C

- Invented in 1972 by Denis Ritchie
- Used to program early UNIX versions
- Defined by the standard in K&R text first version
- Standardized by ANSI organization as C90 – matches the version described in K&R text second version which we use
- C99, C11 and C17 updates include more modern features, so the book is slightly out of date. We will try to highlight differences when they occur, but they are typically not major. C90 is still a perfectly great standard to program with.

The C Programming Language

- General purpose, imperative, typed language designed to support cross-platform code with minimal requirements

- A compiled language: source code cannot be run directly, but rather needs to be processed by a *compiler* which produces machine code.
  - Contrast with shell programs and Python
  - Sign of a compiled language, after compilation, the programs can run even if the compiler is not present on the system
  - This is important for creating low-level components like the kernel

- By having a compiler available for each machine, the identical C code is guaranteed to work everywhere => “cross-platform”

C Philosophy

- C statements are close to the low level features provided by the computer and OS. C gives us access to these, without getting in the way too much.
- The programmer has the power:
  - You have access to low-level memory
  - You are allowed to change the types used to interpret data
  - The language often provides several ways to accomplish the same operation
- The advantage: C code is easily portable across systems, even for low-level operations (example: Linux kernel)
- The risks:
  - The programmer is free to make mistakes (practice and care are needed!)
  - C code can sometimes be "ugly" (practice and care are needed!)
  - C code can be unsafe to run (practice and care are needed!)

Example: Compute Fahrenheit-Celsius table

- Formula is C=(5/9)(F-32)
- We want to represent both values for each matching temperature, and loop over a set of temperatures in the usual range we'd expect in Montreal (wait: it's always hot here right?)
- This gives us a first practice with representing numbers and looping
- From K&R book (1. on course outline) and posted to Github examples

Using simple variables

```
#include <stdio.h>
void main(){
   int fahr, celsius;
   int lower, upper, step;

   lower=0;
   upper=300;
   step=20;
   fahr = lower;
   while( fahr <= upper ){
      celsius = 5 * (fahr-32)/9;
      printf( “%d\t%d\n”, fahr, celsius);
      fahr = fahr + step;
  }
}
```

```
$ gcc temp.c
$ ./a.out

0 -17
20 -6
40 4
60 15
... (lines removed to fit slide)
240 115
260 126
280 137
300 148
```

C Datatypes

- The language provides a number of built-in types. But perhaps less than you have seen in other languages:
  - Integer numbers: int
  - Floating point numbers of single precision: float
  - Floating point numbers of double precision: double
  - Characters to represent text: char
- Several modifiers are used to give more flexibility:
  - Short and long: change the number of bits used to represent an int
  - Signed and unsigned: determine whether one can represent negative numbers
  - Pointers store addresses in the computer's memory

An equivalent program with for

```
#include <stdio.h>
int main(){
        int fahr;
        for( fahr=0; fahr<=300; fahr = fahr + 20 )
                printf( "%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

What about changing types?

```
void main( int argc, char *argv[] )
{
    int i = 5;
    float f = 3.9;
    int result = i + f;
    printf( "Result is %d.\n", result );
}
```

```
$ gcc nums.c
$ ./a.out
Result is 8.
```

Accessing Program Arguments

```
#include <stdio.h>


void main( int argc, char *argv[] )
{
    printf( "I have %d args.\n", argc );
    printf( "The first is %s.\n", argv[0] );
    printf( "The second is %s.\n", argv[1] );
}
```

```
$ gcc args.c
$./a.out elephant
I have 1 args.
The first is ./a.out.
The second is elephant.
```

Functions in C

```
int sum( int start, int end )
{
   int val;
   int sum = 0;
   for( val=start; val <= end; val++ )
   {
      sum += val;
   }
   return sum;
}

int main(){
   printf( "The sum from 1 to 5 is %d.\n", sum(1,5) );
}
```

Function components

- A return type (can be void)
- A name (must be unique in the program)
- An argument list with types and variable names (can be empty)
- A function body enclosed in {}

C Function Properties

- Name must be unique
- Pass arguments by value
- Define a local scope for variables

Pass-by-value example

```
int increment( int fnvar)
{
   fnvar++;
   return fnvar;
}

int main(){
   int a = 5;
   increment(a);
   printf( "The value of a is now %d.\n", a );
}
```

So how do we change the data?

```
int increment( int fnvar)
{
   fnvar++;
   return fnvar;
}

int main(){
   int a = 5;
   a = increment(a);
   printf( "The value of a is now %d.\n", a );
}
```

What about changing multiple values within a function?

- We must instead pass the address to the variable, called a pointer.
- We'll see this often in the course, but not today.

Array data types

- In order to store more than one value within the same variable, C provides simple arrays:

  ```
  int hourly_prices[24];
  float account_balances[100];
  ```

- We must provide the size when declaring the variable, but it can be left off when specifying a function argument:
  - int main(int argc, char *argv[])
  - In this case, the size is decided by the calling function (e.g., here we can type any number of arguments

Array view in memory

- An array variable gives us “space” to store N elements of the declared type
- In C this does not come with much extra help:
  - array.len() is not defined
  - array.sort() is not defined
  - sizeof(array) is a dangerous operation
- This is an example of why C is “low level”. Having no extra info makes a C array efficient, but makes the programmer work a bit harder.

```
int array[4];
array[0] = 1;
array[1] = 3;
array[2] = 4;
array[3] = 2;
```

How large is my array?

- C does not provide any safety mechanisms for programmers regarding array length. It is up to you to remember, and it is possible to accidentally process past the end of the array:

```
int array[10];
for( int i=0; i<100; i++ )
    printf( "Value of array at %d is %d.\n", i, array[i] );
```

- The scariest thing: there are no guarantees about what this code might do!
  - It can run “fine”
  - It can produce a sensible error
  - It can allow you to read private Operating System data! (only if you run it in a clever way)

Beyond the end of an array...

- You may see any of the following:

  *** stack smashing detected ***: ./a.out terminated
  Aborted (core dumped)

  The program continues to run, but some other variable has been changed and problem occur later

  Segmentation fault (core dumped)

  There may be no detectable change and everything seems fine (if you are very lucky/unlucky)

- These outcomes can change quite arbitrarily as you continue to develop your code!

Best practice #1

- Define a constant for the array size. Use it both to set up the array and as a limit any time you use the data:

```
const int array_length = 10;
float account_balances[array_length];
for( pos=0; pos<array_length; pos++ )
  …
```

Best practice #2

- Use "pre-processor replacement" to specify the size:

```
#define array_length 10
float account_balances[array_length];
for( pos=0; pos<array_length; pos++ )
  …
```

- This is not our first look at the pre-processor. We already say #include. Every "#" directive in C calls to the pre-processor, which takes some effect on the code prior to compilation.
  - #define is like find/replace. All instances of array_length become 10
  - #include copies the contents of the header to that location

Best practice #3

- Keep the length of your array in a variable.

```
int array_length = 10;
float account_balances[array_length];
for( pos=0; pos<array_length; pos++ )
  …
```

- This is more powerful because the array can take different lengths at run-time. Let's say based on a user's keyboard input.
- But, it’s a bit less safe because you can change the array_length variable by mistake after using it
  - Then you would be left with no record of the array’s length!

Reading for today

- K&R book Chapter 1 gets you up to where we are now

- K&R book Chapters 2-4 & 7 get you ahead, for this week and the start of next

Exercise: A programming challenge

- Using what we saw so far, can you:
  - Start with the array shown
  - Write code to sort the array in increasing order
  - Output the sorted list on the terminal
- Helper: linear sort specification
  - For each position in the array, pos
    - Let curr be the current value at pos
    - Find the position of the minimal element from pos to the end: min_val, min_pos
    - Swap the values of pos and min_pos

```
int array[4];
array[0] = 1;
array[1] = 3;
array[2] = 4;
array[3] = 2;
```