CS 240 Summer 2017 (Park)

PASSING ARGUMENTS BY VALUE VS. REFERENCE

1. Simple input/output

As discussed in class, even the mundane act of reading and writing an integer using stdio functions scanf() and printf() introduces complications that are not usually found in other programming languages that hide these details from the programmer. This is good and bad. Good, because for many things hiding the details of how the actual computing system does its job is unnecessary for application programming. Bad, because for some types of application programming the features provided by C are necessary to get the job done. That is why most operating systems, including Linux, are mostly written in C, but so are many other important software including the Java Virtual Machine which provides the software run-time environment of Java bytecode.

2. Printing to terminal using printf()

Suppose main() uses stdio's printf() to print an integer on the terminal:

```
#include <stdio.h>

int main()
{
int x;

  x = 2;
  printf("%d",x);

}
```

We did not write printf() but it is just another function that takes arguments and then performs output as communicated by the caller, main(). Before we can understand how main() and printf() communicate with each other to accomplish the goal of printing 2 on the terminal, we have to consider the layout of executable machine code loaded into memory.

3. Memory layout of executable machine code

As discussed in class, after gcc has compiled and linked C code into an executable binary (i.e., machine code), when we "run" the binary, say, from a shell

```
% a.out
```

the machine code and its data in the file a.out are loaded into RAM. In the above example, the machine code for main() occupies a portion of memory and so does the machine code of printf(). These are the code (also called text) segments.

The integer variable x which is defined inside main() constitutes a local (or private) variable of main(). Each C function is allocated a data area (its working space) in RAM where it stores local variables and other items it needs. In our example, the function main() gets is data area, so does printf(). When global (or public) variables/data structures are used in C programs, an additional data area is allocated in RAM where the

shared global variables are stored. Our example program does not use
global variables so there is no separate data area in RAM for globals.

4. Back to terminal output using printf()

For the discussion ahead, let's assume that main(), printf(), and their
local data areas are located in RAM in a 32-bit architecture as follows:

code of main():
address 1000-1100

code of printf():
address 5000-7000

data of main():
address 12000
// Note that main() has only one local variable of type integer which
// is 4 bytes long. Hence it fits exactly at address 12000 which holds
// 32 bits.

data of printf():
address 15000-16000

Part of the data area of printf() is used to communicate the arguments
passed from main(). Suppose that address 15000 is used to pass the second
argument of printf(), x.

Thus the original value of x, which is 2, is stored at address 12000 in
the data area of main(). But a copy of it is also stored at address 15000
in the data area of printf(). To print the value of x, printf() just
needs to look up what address 15000 contains and output the value to the
terminal. If printf() knew that x is located at 12000, it could directly
access address 12000. But it doesn't since local variables are used
to keep data private.

The above means of coordinating between caller (main()) and callee
(printf()) to communicate arguments is called passing by value. The system
with the help of gcc makes a copy of the value of x in printf()'s data
area which then printf() is able to access and print.

5. Reading from standard input using scanf()

As discussed in class, the situation for reading from standard input
(i.e., keyboard) is different. In the C code

```
#include <stdio.h>

int main()
{
int x;

  x = 2;
  scanf("%d",&x);
  printf("%d",x);

}
```

we inserted scanf() after the statement x=2 which should overwrite the
content 2 at address 12000 with whatever integer value we read in.
printf() should confirm this. With scanf(), now we have a third function

and its data area that must be allocated in RAM. Suppose the locations are:

code of scanf():
address 9000-11000

data of scanf():
address 17000-18000

We are assuming that the code and data areas of main() and printf() remain the same.

Suppose address 17000 in the data area of scanf() is used to store the second argument passed by main() to scanf(). That is, x preceded by ampersand &, i.e., &x. In C, the expression &x means the "address of x" which in our example is 12000.

Thus at memory address 17000, the content is 12000 which is another address! That is, the content of an address need not be an integer, float or other "normal" data type but can be another address. In C, we call addresses that contain other addresses pointers. Hence memory location 17000 is a pointer.

So, what does the above accomplish? Address 17000 which is part of scanf()'s data area contains 12000, the address of integer variable x which is part of main()'s data area. This means that scanf() now can look up the content of address 17000, which yields the address 12000, to directly write the integer input from the keyboard to RAM address 12000. So if the use types in 7, scanf() ends up writing 7 into address 12000. Therefore, when main() examines the content of its local variable x by printing its value using printf() it will find 7.

Thus the intent of the C programmer to read input from keyboard and make it available in a local variable of main() has been accomplished. The ampersand was needed in the C program for the caller main() to tell the callee scanf() that it is passing the address of x, not its value. This is therefore called passing by address, also passing by reference.