Understand Execution of a Program

Prof. Zhang

September 17, 2014

1 Program in Memory

When you execute a program, the program (i.e., executable code) is loaded into the **memory** (i.e., **main memory**, **RAM**) in order to be executed. Below we first review the memory, and then describe the memory map of a program.

1.1 Main Memory

The memory of a computer system is a huge array of **bytes** (each byte is 8 bits, i.e., binary digits). (It's a volatile memory, meaning its content will be lost once the program finishes execution, or if the computer loses power). When declaring or allocating memory space, it is always done in terms of bytes.

You can use function **sizeof** to find out the amount of memory (in the units of bytes) is used for a variable. See the following code segment and comments.

```
double a;
int b[20];

cout << "size of a" << sizeof (a) << endl;
// this will display 8, as double variable takes up 8 bytes, i.e., 64 bits


cout << "size of b" << sizeof (b) << endl;
// this will display 80: one int variable is 4 bytes....

cout << "size of char type" << sizeof (char) << endl;
// this will display 1, each char is represented by 8 bits ASCII code
```

1.2 Memory of a Program

When you run your program (or any program), the program is loaded into the memory (by the operating system). The size of the memory used for a program differs (depending

on how large is the executable code, how many variables are used.).

The memory a program uses is typically divided into four different areas:

- the **code area**, where the compiled program sits in memory.
- the **globals area**, where global variables are stored.
- The **heap**, where dynamically allocated variables are allocated from. We will learn more about **heap** later (after we studied pointers).
- The **stack** (also called **call stack**, **function call stack**. Every time a function is called, a block of memory named **call stack frame** is **pushed to the stack**. The call stack frame stores this function call's parameters, local variables, and **return address**. (The return address keeps track of where to resume execution in the caller function, when this function call returns). When the function returns (finishes its execution), the call stack frame is **popped out of the stack** (deleted from the stack). The program resumes execution at the statement indicated by the **return address**.

  The call stack is kind of like a office space shared by all function calls. Every function, when being called, needs certain memory space to hold its local variables and parameters, and to remember whom to report results to. That memory space is called **call stack frame**. *The main function is the boss (with its office space). When the boss calls upon someone to work on a task, that worker gets his own work space. When someone finishes the task, he cleans up the work space allocated to him; wakes up his boss (who will resumes his work based upon the return address).*

Example

For example: let's draw the snapshot of the memory of the following program, right at the time when func_b() is executing its second statement:

```
const int a;

void func_b()
{
  int i;
  cout <<"here\n";
}

void func_a(int a)
{
  cout <<"a=" << a << endl;
  func_b();
```

```
  cout <<"done\n";
}

int main()
{
  int counter=10;
  cout <<"calling func_a();
  func_a();
}
```

1.3 Runtime errors related to memory

Sometimes you will encounter runtime errors that are related to memory usages:

- **Segmentation Fault**: if you are accessing memory location that belongs to other program or operating system. This can happen if you access an array element at an index value out of its bounds:
- **Bus Error**: if you use an invalid address when accessing memory
- **Stack Overflow**: if the program has many levels of function calls, and run out of **call stack** space to hold the **call stack frames** for the ongoing calls.

2 Function Call Details

2.1 Funcation call syntax

Recall the syntax of calling a function:

```
function_name (list of actual arguments);

where the list of actual argument are separated by comma, for example:

   DisplayArray (NumArray, NumArraySize);

The number of actual arguments and their types need to match those in the
function's header (declaration and definition).
```

Please note the following two terms we use throughout the class:

- (Formal) parameters: these are variables and their types as specified in the function declaration and definition (listed in the function's head).

- (Actual) arguments: there are the expressions, constants, or variables that are supplied in the function call.

A Common logic error is to call a function that returns a value, without using the return value:

```
...
sum (10,20);
pow (2, 8);
```

The above two function calls have no effects: the functions are called and executed, but the returned values are not used.

Use return value of a function: if a function returns some value (i.e., return type is not **void**), then the function call can be used anywhere that value type is expected. For example,

```
if (ContainDuplicate (NumArray, NumArraySize)) // where the function returns bool
   cout << "Array contains duplicate\n";
else
   cout << "Array does not contain duplicate\n";

double x = sqrt (10)*34;
```

3 Tracing function calls: Putting it all together

When tracing a program, we follow the execution of a program with a set of sample input values, in order to identify logic errors, understand codes written by others, or to verify your design and code.

The keys to program tracing:

- **Remembering where you are using arrows**: draw an arrow to point to *the current statement in each function that is currently running.*

    1. Move the arrow following the logic structure (if/else statement, loop statement, and switch statement).
    2. When the arrow moves to a function call, draw a new arrow to point to the first statement in the funciton being called.
    3. When reaching **return** statement in **main**, or end of main's }, the program ends.

- **Keeping track of variables**

We have learnt to use a box, labelled with the name to represent a variable. Every time a value is assigned to the variable, we update the content of the box (so that we remember what value the variable has).

In order to trace function calls better, we now keep global variables in the **globals area**, and keep track of local variables in the **call stack**.

- **Tracing function call**: When we reach a function call:

  1. We draw a new **call stack frame** for the function call on the top of the **call stack**, and write down the function's local variables, parameters, and return address inside it.
  2. Passing parameters: actual arguments are passed into the function, meaning, actual arguments are assigned to formal parameter variables (based upon order).
     - **pass-by-value**: the **values** of arguments are assigned to parameters, i.e., the function parameter (which is a local variable) is initialized with the value of the arguments.
     - **pass-by-reference**: the **address** of arguments. are assigned to parameters. For tracing purpose, we just write something like, **main's counter** there. **All reference to the parameter are made to the acutal arguments.** It's like substituting all occurences of the pass-by-reference parameter by its acutual argument.
  3. Write down **return address**.
  4. Pass control: start to execute function body from the first statement in the body
  5. Return from function: function's execution ends whenever a return statement is executed, or the end of function body is reached. If a value is returned, the value is returned to the caller. The execution resumed from the next statement in the caller function. Erase (or cross out) the top call stack frame (All local variables of the function are forgotten).

4 Exercise in program tracing

1. **Function with both types of parameters**

```
void AddTax (double & amt_due, double rate);

int main ( )
{
   double total_order = 120.00;
   double rate = 0.05;
```

```
   cout <<Total order:  << total_order <<endl;
   AddTax (total_order, rate);
   cout <<With tax total due: << total_order<<endl;
   cout <<Tax rate is : << rate << endl;
}

void AddTax (double & amt_due, double rate)
{
  rate = rate+1.0;
  amt_due = amt_due*rate;
}
```

2. **Recursive Function**

```
int main()
{
  cout << Fibo (3);
}

int Fibo (int n)
{
   if (n==1)
      return 1;
   else if (n==2)
    return 1;
   else
   {
      int result = Fibo (n-1) + Fibo (n-2);
        return result;
   }
}
```