Activation Records and the Stack in C/C++
ENCM 339 Class Handout, Fall 2014

Originally written by Steve Norman in Sept 1995 for similar courses, and currently updated in Sept 2014

Memory allocation

To *allocate* memory for a variable or array means to reserve a piece of a program's memory space to store the value of the variable or the values of the array elements.

To *deallocate* memory is to ``turn off'' the reservation that was made when the memory was allocated. Deallocated memory is recycled - it may be used again for storage of other variables or arrays.

There are three types of memory allocation in C and C++ programs: *automatic* allocation, *static* allocation and *dynamic* allocation.

Automatic allocation is the main topic of this handout, and is discussed in detail starting in the next section.

Static allocation is used for global variables, `static` local variables, and string constants. Allocation of static storage takes places just before execution of the program starts. Deallocation does not occur until after the program has finished running.

Dynamic allocation is controlled by the library functions `malloc` and `free`. Allocation and deallocation of dynamic storage is under the programmer's direct control.

Another handout will say a bit more about static and dynamic allocation.

Activation records

In a C/C++ program, multiple functions can be *active* at the same time. For example, suppose `main` calls `f`, which in turn calls `g`. Then while `g` is executing, `main`, `f`, and `g` are all active.

Each active function has an *activation record*, a chunk of computer memory which holds the arguments and ``normal'' local variables of the function. (Another widely-used term for *activation record* is ``stack frame''.) In this context ``normal'' means ``not declared to be `static` or `register`.'' The official C/C++ terminology for ``normal local variable'' is *automatic variable*, because activation records are automatically allocated when a function is called and automatically deallocated when a function returns to its caller.

(In addition to arguments and local variables, an activation record is likely to contain two other pieces of data: a ``return address'' and a ``saved frame pointer''. To understand how C/C++ programs work at a machine language level, it is important to know what these things are. However, this document is treating C/C++ with a slightly higher level of abstraction, so these two pieces of data will be left out of the discussion and left out of the diagrams.)

The activation records for all of the active functions are stored in the region of computer memory called *the stack*. A good way to remember this is to think of the stack as ``the region of memory where all the activation records are stacked''.

The activation record is temporary - it is allocated when the function is called (becomes active) and it is deallocated when the function returns (becomes inactive). Once an activation record is deallocated, the memory it occupied is available for use when new activation records are created.

C and C++ use *call by value*, which means that values of each argument in a function call gets *copied* into a location in the activation record. Data transmission through the argument list is *one-way*, because if the values of the arguments change in the called function, these changes are *not* copied back to the calling function.

Functions cannot *directly* access the activation records of other functions, nor can they *directly* access the `static` local variables of other functions. The *scope* of a function's arguments and local variables is limited to the block that defines the function.

Pointer arguments can be used to give functions *indirect* access to the local data (variables and arguments) of other functions.

In C++, reference arguments can also be used to give functions access to the local data of other functions, but *C does NOT have reference types*.

Conventions for stack diagrams in ENCM 339

When drawing stack diagrams, please use the conventions described here.

Activation records should be drawn as rectangles. There should be a horizontal line dividing the activation record into two regions. Function arguments should appear below the line and local variables above the line.

If a function has no arguments, write ``no arguments'' in the argument area. If there are no local variables, write ``no local variables'' in the local variable area.

Draw activation records for active functions only.

The activation records (ARs) should be drawn in a stack. The AR for `main` should be at the bottom of the stack, and the AR for the currently executing function should be at the top. The ordering of the ARs in between the bottom and the top should reflect the sequence of function calls that led from `main` to the currently executing function.

These conventions are used to draw stack diagrams for the examples in the next section.

Example program: introduction and listing

Listed below is a small example program. The purpose of this program is to demonstrate how activation records work; the program is *not* intended to serve as a good example of style or program organization.

There are comments in the program marking `point 1`, `point 2`, etc. For example, the program is at `point 1` just *before* the statement `i = -8;` is executed. The program passes through these points in order. Diagrams showing what the stack looks like at each of these are shown further down in this document. If you are using a Web browser to read this document, you can use hypertext links to jump back and forth between the listing and the diagrams.

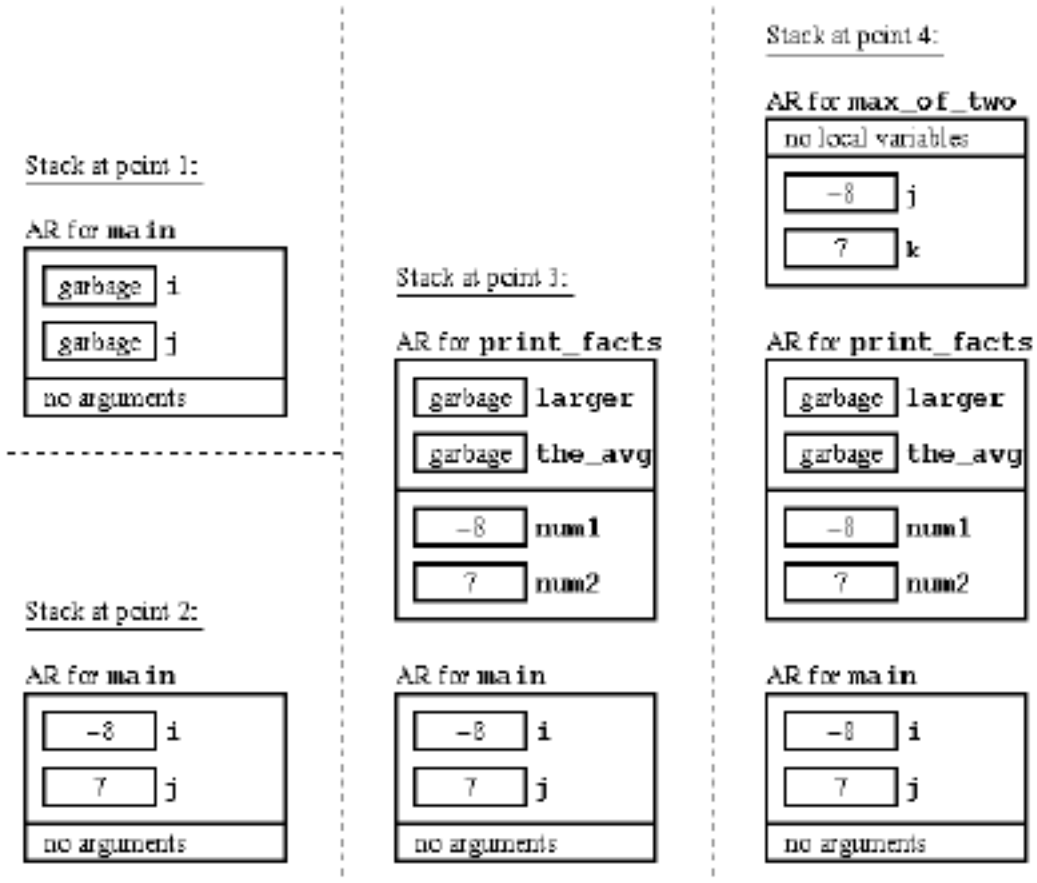
(Note that comments such as `/* point 1 */` do not always correspond to a unique moment in the execution of a program. For example, if the comment is inside a loop or inside a function that is called more than once, the program may pass through a marked point more than once. This situation will come up near the end of the course when you will be asked to illustrate the progress of recursive functions.)

```
#include <stdio.h>

void print_facts(int num1, int num2);
int max_of_two(int j, int k);
double avg_of_two(int c, int d);

int main(void)
{
  int i;
  int j;
  /* point  1 */
  i = -8;
  j = 7;
  /* point  2 */
  print_facts(i, j);
  /* point 10 */
  return 0;
}

void print_facts(int num1, int num2)
{
  int larger;
  double the_avg;
  /* point  3 */
  larger = max_of_two(num1, num2);
  /* point  6 */
  the_avg = avg_of_two(num1, num2);
  /* point  9 */
  printf("For the two integers %d and %d,\n", num1, num2);
  printf("the larger is %d and the average is %g.\n",
         larger, the_avg);
}

int max_of_two(int j, int k)
{
  /* point  4 */
  if (j < k)
    j = k;
  /* point  5 */
  return j;
}

double avg_of_two(int c, int d)
{
  double sum;
  /* point  7 */
  sum = c + d;
  /* point  8 */
  return (c + d) / 2.0;
}
```

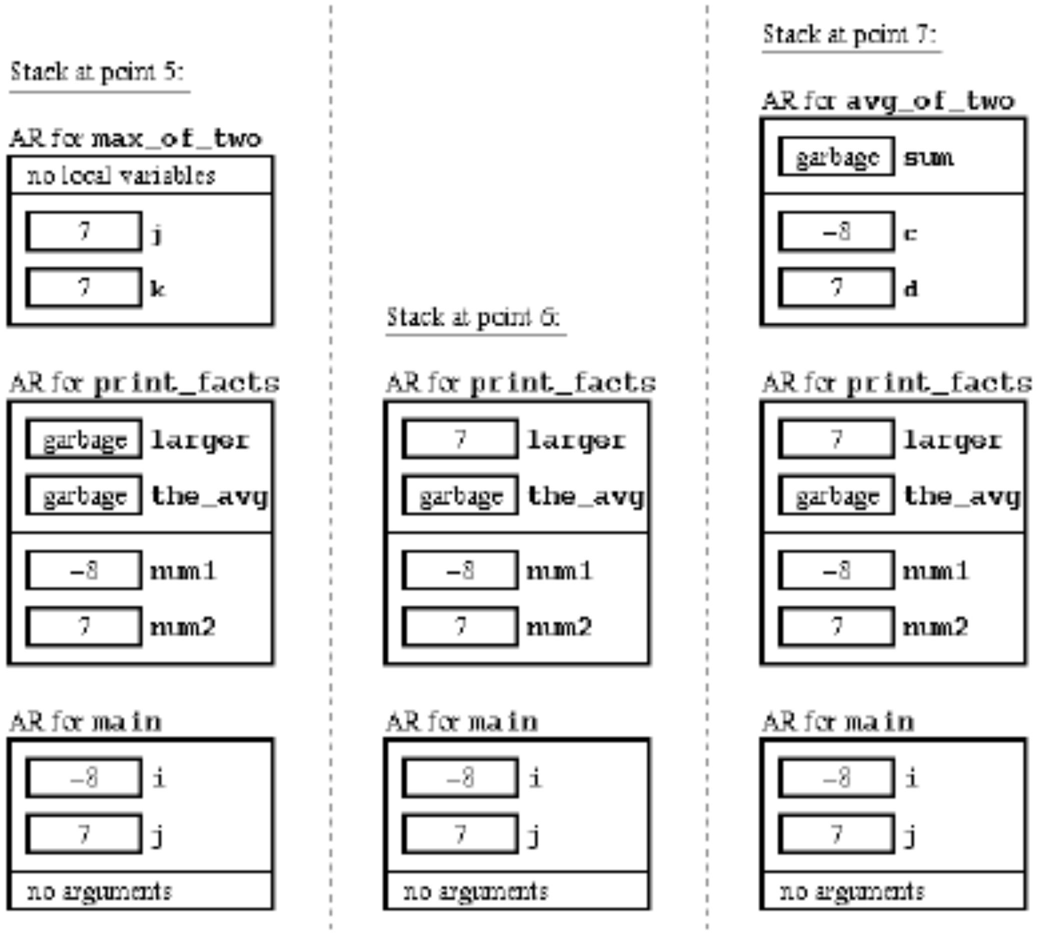
You should be able to trace through the execution of the program, following the appearance and disappearance of activation records and the changes to the values of variables. Here are some specific things to note:

> `main` is a function, so it has an activation record like any other function.
>
> The effect of call by value is demonstrated several times. The first time is at `point 3`, where you can see that the values from `i` and `j` of `main` have been copied into `num1` and `num2` of `print_facts`. It is important to realize that `i` and `num1` have distinct storage locations, as do `j` and `num2`.
>
> Immediately after a function is called, its local variables contain ``garbage'' values. (It gets tedious writing ``garbage'' over and over again--you may write ``??'' instead to indicate that a variable has an unknown value.) The garbage remains there until it is overwritten with valid data. Variables are never ``empty'' - they contain either garbage data or meaningful data.
>
> `main` has a variable named `j` and `max_of_two` has an argument named `j`. There is no conflict here because the scopes of the two `j`'s do not overlap. (On the other hand, it's confusing style, especially since the `j` of `max_of_two` ends up receiving the value of the `i` of `main`.)
>
> The strange style of the definition of `max_of_two` shows that an argument can be used like a local variable. Note how `j` gets a new value in `max_of_two`. Note also that this *does not* affect the value of `num1` in `print_facts`. Once the value of `num1` gets copied into `j`, there is no further connection between `num1` and `j`.

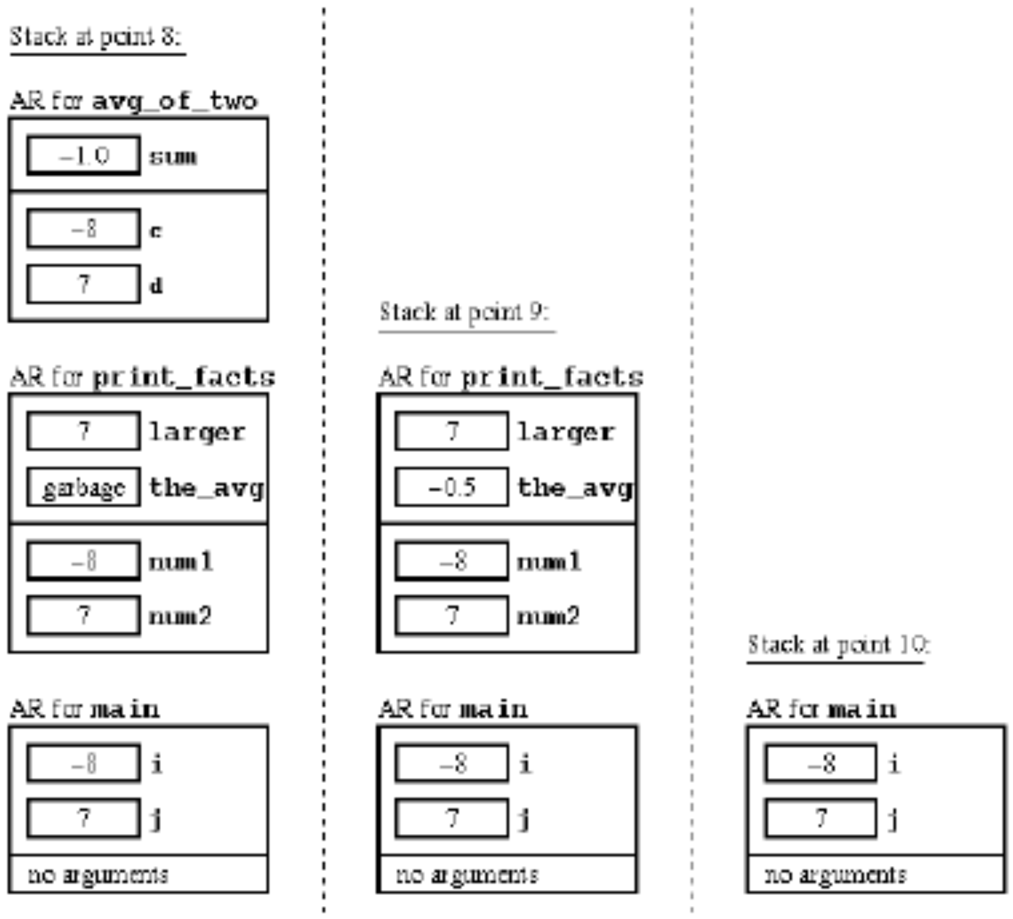
Example program: diagrams for points 1, 2, 3 and 4



Example program: diagrams for points 5, 6 and 7



Example program: diagrams for points 8, 9 and 10



Complications - blocks within blocks

If you have read your C text carefully, you know that it is legal to declare variables within blocks contained within other blocks. For example:

```
void SquareTable(int lower, int upper)
{
  int n;
  for (n = lower; n <= upper; n++)
  {
    int square;
    square = n * n;
    printf("%8d%8d\n", n, square);
  }
  printf("----------------\n");
}
```

You might wonder (a) When does `square` get allocated and deallocated? and (b) How should the memory diagram be drawn?

The answer to (a) is: The memory is allocated and deallocated on each pass through the inner block. You should not assume that `square` maintains its value from one pass to the next.

The answer to (b) is: Don't worry about it for now; later in the course I'll provide a detailed description of how to handle this kind of situation.

More to come

This handout is *just the beginning* of coverage of how memory is organized in running C and C++ programs. Here are some important topics yet to be covered:

- local variables in C++
- references in C++
- pointers in both C and C++
- many issues related to C++ `class` types
- static memory allocation in both C and C++
- dynamic memory allocation in both C and C++