Assignment 8 Answers
Ryan Levering
April 1, 2003

1) What is a scripting language? How is it different than a programming language like C, C++, Java, etc.?

   Actually, there is some variation on what is generally accepted as a scripting language. Ask any game developer and they'll tell you a different story as scripting languages are very popular with multiple levels of game development. However, the commonly accepted scripting language is a run-time interpreted language. Typically they are interpreted (parsed and executed) on the fly, running within their own pre-compiled environment. Because of this, they tend to be safer, but not as flexible. This also gives some reason to call Java a scripting language of a sort because originally it was interpreted from byte code. However, a language such as JavaScript is a much more clear example of what a scripting language is.

2) "Reference counts" is one algorithm that can be used in a runtime environment to collect garbage. Give an example of a data configuration that this technique may fail to collect.

   Reference counts is an algorithm that keeps track of how many references there are to each block/object. However, in cyclical structures, such as circular queues, the memory will never be returned to the heap.

3) Following the definition of binding a name with its value, specify the time and stability of the binding between C const variable and its value.

   An lvalue expression of a const-qualified type cannot be used to modify an object. This is, such an lvalue cannot be used as the left operand of an assignment expression or the operand of an increment or decrement operator… However, the language rules for const are not foolproof – that is, they may be bypassed or overridden if the programmer tries hard enough (Harbinson & Steele, p. 89-91) Also, the value is bound at run-time even though it's checked by the compiler.

4) Following the definition of binding a name with its value, specify the time and stability of the binding between C++ const variable and its value. What is the difference between the meaning of const in C and C++? Can you come up with an example where this difference causes code to compile in C++ and not in C?

   In C++, const value is bound as early as it can be determined. Thus
   const int x = 10; will be bound at compile time while
   void fun(int x) { const int y = x; } will be bound at run time

   C always binds at runtime with const keyword. Use #define for compile time constants.

   In C++ you can cast away they const keyword with references, while you can't in C.

So,
const int x = 10;
int* y = &x;
((int) y) = 5;

Furthermore,
C: const always creates storage, uses external linkages, and doesn't need initialization
C++: const doesn't have to create storage, uses internal linkages, and must initialize

http://www.acm.org/crossroads/xrds1-1/ovp.html
http://www.eskimo.com/~scs/C-faq/q11.8.html

5) What is void *? Why have it? void * does not behave exactly the same in C and C++. Provide an example (i.e. code) that will illustrate this difference between C and C++.

void* is a pointer to void, which represents a generic pointer. void* is good when you don't know the type of a pointer being passed.

C++ requires that you cast a void* to a specific pointer before you use it, whereas C will just assume it's the type you are using it as.
void* myPointer;
…
int* numPointer = myPointer; // Ahhh, C++ is freaking out!
int* numPointer = (int*) myPointer; // Much better

6) In C++, what is the difference between taking the address (via operation &) of a pointer and of a reference? (You can just write a little test program in C++).

Taking the address of a reference will return the location of the r-value. Taking the address of a pointer will return the location of the pointer's l-value. Or to give an example, if the data x = 5 is stored in memory location 506 and there is a pointer *y = &x, where y is located in memory location 1004 and has the data 506, then returning the address of the reference would return 506 while returning the address of the pointer would return 1004.

7) In ANSI C the keyword static has two different semantics depending on where it is used, either inside a function definition or outside of a function definition. What does static mean in both settings? Where in memory do we expect to find the L-value of static variables defined inside of a function definition?

Declaring a variable static inside a function creates an automatic variable, which only the function can access. However, unlike normal automatic variables, the static variable is not destroyed on exit from the function, instead its value is preserved and becomes available again when the function is next called. "The keyword 'static', when applied to a function or variable defined outside a function, does not actually indicate static storage class. It is used to hide the function or variable from code in other files

(http://occs.cs.oberlin.edu/faculty/jdonalds/341/CPrimer.html)." Otherwise, it's not exported to the linker. The L-value or address of static variables inside functions is kept in program static memory, separate from the stack and outside of the heap.

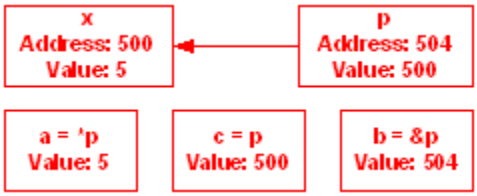
8) Run the following C program and explain the resulting output:

```
void q(int z) {
  { int x;
     printf("%d\n",x);
     x = 3;
   }
   { int y;
      printf("%d\n",y);
    }
}
int main ( ) {
   q(99);
}
```

You should get something like:
-12342435
3

The reason is because you are printing uninitialized values. First x is printing whatever random bits were in its space before. Then you set x = 3 and it goes out of scope, so when y is defined, it is defined in the space where x used to reside. So printing y results in the value of x being printed.

9) Compile and run hw8q7S03.c. Draw pictures to illustrate the binding with the values and variables and operators (*, &).



10) Given the following declaration in Ada:

```
TYPE Array_type IS ARRAY(2..7) OF CHARACTER;
TYPE CODE is record
    Level: CHARACTER;
    Status : CHARACTER;
    Salary: INTERGER;
end record;
TYPE Code_Array IS ARRAY(11..13) OF CODE;
arr : Array_type;
codeArr : Code_Array;
```

If INTEGER requires 4 bytes of storage and CHARACTER requires 2 byte of storage what is the effective address (using bytes as the addressable units) of variables below (show your calculations) You should assume that the ba (base address) for arr is 500 and the base address for codeArr is 300.
You may assume the memory is byte addressable and you may use decimal arithmetic.

Effective address for arr(4) is 500 + 2*2 = 504
Effective address for codeArr(12) is 300 + 2 + 2 + 4 = 308
What is the offset of Status in the record Code? 2
Effective addres for codeArr(12).Status is 308 + 2 = 310

11) Given the following declaration in Ada:

```
TYPE Array_type IS ARRAY(10..15,1..4) OF CHARACTER;
TYPE CODE is record
     Status : CHARACTER;
     Salary : Integer;
end record;
TYPE Code_Array IS ARRAY(1..5,10..15) OF CODE;
arr : Array_type;
codeArr : Code_Array;
```

If INTEGER requires 4 bytes of storage and CHARACTER requires 2 byte of storage what is the effective address (using bytes as the addressable units) of variables below (show your calculations) You should assume that the ba (base address) for arr is 4001 and the base address for codeArr is 5010. You may assume the memory is byte addressable and you may use decimal arithmetic.

Using row major the effective address for arr(10, 3) is 4001 + 2*2 = 4005
Using column major the effective address for arr(10, 3) is 4001 + 2*12 = 4025
Using row major the effective address for codeArr(2, 11) is 5010 + 6*6 + 6 = 5052
Using row major the effective address for codeArr(2, 11).Salary is 5052 + 2 = 5054

12) For the following C3 program fragment, show the snapshot of the run-time stack at each critical stage until the code returns to main.

In particular, show the static and dynamic links before each call. (Follow the presentation of run-times environment stack in class notes.)

```
int a = 299, j = 2, k = 1, i = 3;
int beta(int x ) {

      int f = 5;
      int alpha(int v ){
            int j = 4;
                        //++ <<<<<< Show stack here
            . . . .
            j = k + v;
            x = 100;
            beta(9 );
```

```
                    // +++ <<<<<< Show stack here
          ....
          return j;
     }; // END of alpha code

     // Beginning of beta code
          //<<<<<<<<<<<  show the stack here after call to beta
     if (j>3) return 55;  //Termination
     j = i + x;
     alpha(f);
         //<<<<<<<<< after returned from alpha

     return 88;
};   // END of beta code

int main(void ){
int  v = 22;
        ....  // ** stack before beta is called
     beta(9);
          // *** stack after beta is returned
        ....
return v;
}
```

Description	D addr	main **	beta1	alpha++	beta2	alpha+++	beta1	main ***
		Just	after	calling	function	Just	after	return
global a	1	299	299	299	299	299	299	299
global j	2	2	12	12	12	12	12	12
global k	3	1	1	1	1	1	1	1
global i	4	3	3	3	3	3	3	3
return value	5	?	?	?	?	?	?	?
main loc,v	6	22	22	22	22	22	22	22
return value	7		?	?	?	?	?	88
beta1 arg, x	8		9	9	100	100	100	
static link	9		0	0	0	0	0	
return add	10		main()	main()	main()	main()	main()	
fp	11		0	0	0	0	0	
beta1 loc, f	12		5	5	5	5	5	
return value	13			?	?	?	6	
alpha1 arg, v	14			5	5	5		
static link	15			9	9	9		
return add	16			beta()	beta()	beta()		
fp	17			7	7	7		
alpha1 loc, j	18			4	6	6		
return value	19				?	55		
beta2 arg, x	20				9			
static link	21				0			
return add	22				alpha()			
fp	23				13			
beta2 loc, f	24				5			
Frame Pt	0	7	13	19	13	7	0	
Stack Pt	7	13	19	25	19	13	7	
Instruct Pt	main()	beta()	alpha()	beta()	alpha()	beta()	main()	