The C Runtime System

I have to say thanks to Jason Maasson from Frije Universiteit Amsterdam for his excellent script and slides. I've translated most of the slides and added some graphics and text.I would also like to thank Marshall Brain, founder of "www.howstuffworks.com" for his wonderful article on "How C Programming Works" which explains also the c-runtime environment.And last but not least Steven Simpson from Lancaster University for pointing out the differences between both languages on a few excellent pages.Look at the resource section at the end for links.

Introduction

Goals

1. **A word on debugging and generating assembler code**
2. **learn how function calls work**
3. **Learn what happens when a C program is executed**
4. **Understand memory management issues and performance problems**
5. **Frequent bugs and errors in C programs**

Roadmap

1. **How to create assembler code**
2. **Assembler code for a funtion call**
3. **How malloc works and when OS support is needed**
4. **Purify to the rescue: when your program leaks memory**
5. **How the system call execve starts a C program**
6. **How to debug your programs**

Assembler code

Generating assembler code

Sometimes when you are working on a device driver or some kernel functions - or just if you need to speed-up one small function in your program you might want to create the function in assembler code. Writing assembler is hard but you can let the C compiler do the hard stuff for you and then just optimize the result. Sometimes you might hit an optimizer bug which forces you to look at the generated code to see where the problem is - even compilers have bugs.

```
gcc -S fCall.c
```

produces assembler output.

A C code example to demonstrate assembly language

```
extern int myFunc(int intPar, char* stringPar);

int main(void) {
    char * string = "hello";  // a local string
    int result = 0;           // a local integer
    int arg1 = 1;             // a local integer
    result = myFunc(arg1, string); // 2 parameter, 1 return parameter
    return 0;
}

int myFunc(int intPar, char* stringPar) {

    int localVar = intPar;        // parameter used to init local var.
    char* localString = stringPar; // same here
    return 44;                   // return value needs to go back to caller
}
```

A word on assembler code

The example below uses assembler code for intel type CPUs generated by the GNU assembler. Unlike most Intel assembler this tools moves things from the left side to the right side in an expression. Some things to know for reading the output:

.text like .date or .bss this statement simply tells where the linker finally has to put the following code or data. In our case we will see that sometimes even data (e.g. a string "hello") might go into the non-writeable text segment.

.globl A statement that makes stuff included here available for public access (linker can use it to resolve externals)

eax, ebx... The register names of Intel CPUs. eax is really the workhorse of those CPUs (accumulator). As you can see a typical access goes like this: move something from stack into eax. Move it from eax somewhere else on the stack. This is also the way function parameters are accessed and assigned to local variables

Assembler output listing

My comments are tagged with the C comment characters.

```
.file "fCall.c"    // the original source file
.def ___main; .scl 2; .type 32; .endef
.text              // this goes into the text segment
LC0:                 // a gnerated label to store the string value
.ascii "hello\0"   // note that the string goes into TEXT segment!
.align 2           // tells linker to align this section on a two byte border
.globl _main         // puts our main in the .global section
.def _main; .scl 2; .type 32; .endef
_main:               // this is where our main really starts
pushl %ebp           // save frame (base) pointer on stack
movl %esp, %ebp     // store stackpointer in framepointer
subl $24, %esp      // decrement stack pointer to make room
                     // for main's local variables and future call arguments
andl $-16, %esp
movl $0, %eax       // put 0 into eax register
movl %eax, -16(%ebp) // move content of eax to where framepointer points
                      // but with offset -16
movl -16(%ebp), %eax // get framepointer offset -16 into eax ???
call __alloca       // call a c-runtime function to allocate memory
call ___main        // call a c-runtime function __main

movl $LC0, -4(%ebp) // get the address of LC0 (our string) and put it to
                     // framepointer -4 on the stack. This is not surprisingly
                     // where our first local variable "string" lives.
```

Assembler output listing (Continued)

```
movl $0, -8(%ebp)   // put a 8 at framepointer -8 where our second var lives
                     // (result)
movl $1, -12(%ebp)  // put 1 at framepointer -12 where our third var (arg1)lives
movl -12(%ebp), %eax// move arg1 from stack into register eax
movl %eax, (%esp)   // put eax content on stack. Note, this is the leftmost
                     // argument for function call!!!
movl -4(%ebp), %eax // put the string address from stack into eax
movl %eax, 4(%esp)  // and put it from there onto the stack
                     // now we have set up the stack with two arguments and can
                     // call our function.
                     // You will need to look at _myFunc to see how the arguments
                     // are processed by our function.
call _myFunc        // tell CPU to push the current instruction address on the
                     // stack (saving it for return) and move the address where
                     // _myFunc lives into the instruction register
//-------------------------------------------------------
// after we come back from the call to _myFunc:
movl %eax, -8(%ebp) // in eax is the return value from our function call
                     // with a debugger you would find 44 here
movl $0, %eax       // generate main's return value (0 for everything OK)
leave                // restore framepointer and stackpointer to how they
                     // where BEFORE main function was running
ret                  // tell CPU to get return address (the address from where
                     // our main was called) and jump to it, effectively continuing
                     // callers processing right after the function call to main.
                     // in this case this would mean jumping back into the
                     // C-runtime code which started our program.
```

Assembler output listing (Continued)

```
//-----------------------------------------------------
.align 2
.globl _myFunc
.def _myFunc; .scl 2; .type 32; .endef
myFunc:
pushl %ebp           //save framepointer from caller (prolog)
movl %esp, %ebp     // put stackpointer into framepointer which
                     // is now our vehicle to access OUR (myFunc's) local
                     // variables and arguments
subl $8, %esp       // make room on stack for 8 bytes
movl 8(%ebp), %eax  // move 4 bytes from framepointer offset 8 into eax register
movl %eax, -4(%ebp) // move those bytes now into local variable at offset -4
                     // what does: int localVar = intPar;
movl 12(%ebp), %eax // again move 4 bytes from offset 12 into eax
movl %eax, -8(%ebp) // and store them at offset -8 which makes:
                    //     char* localString = stringPar;
movl $44, %eax     // put decimal 44 into register eax which
                    // creates our RETURN value (must be in eax as we
                    // will see when we look at how the caller receives this
                    // value: return 44;
leave               // clean up the local calling frame (restore framepointer and
                    // stack pointer to HOW THE CALLER LEFT THEM. Any surprise here
                    // is not really funny.
ret                 // tells the cpu to take the callers return address from stack
                    // into the instruction pointer register and continue executing
```

Assembler output listing (Continued)

How function calls work

Why is this assembler stuff important?

A close look at how a function call really works will teach you several things:

Call by Reference vs. call by Value

Look at the function arguments arg1 (an integer) and string (a character pointer). The value from the integer is COPIED onto the stack and received as a value from myFunc. At NO time does myFunc access the memory where arg1 really lives on the stack because this stack area belongs to the main function. myFunc works on a COPY of the integer. But now look at the string variable. It is NOT copied onto the stack. Instead, the MEMORY ADDRESS where the string lives is COPIED with the following instruction onto the stack: movl $LC0, -4(%ebp). If the receiving function myFunc would do anything with the string parameter it would go DIRECTLY to where the string lives - leaving its own memory area.

Speed and Pointers

The string from above is a pointer and you have seen it is transported by reference (effectively only its address is copied, not the whole string). This is of course much faster than performing a whole string copy operation across the stack. But it also shows the danger of pointers (or references in general). There are now 2 places which can access the string: main and myFunc. Main cannot prevent myFunc from abusing its variable

Why is this assembler stuff important? (Continued)

string through the pointer (not true because of the text segment here but if string would not be a literal it would be true).

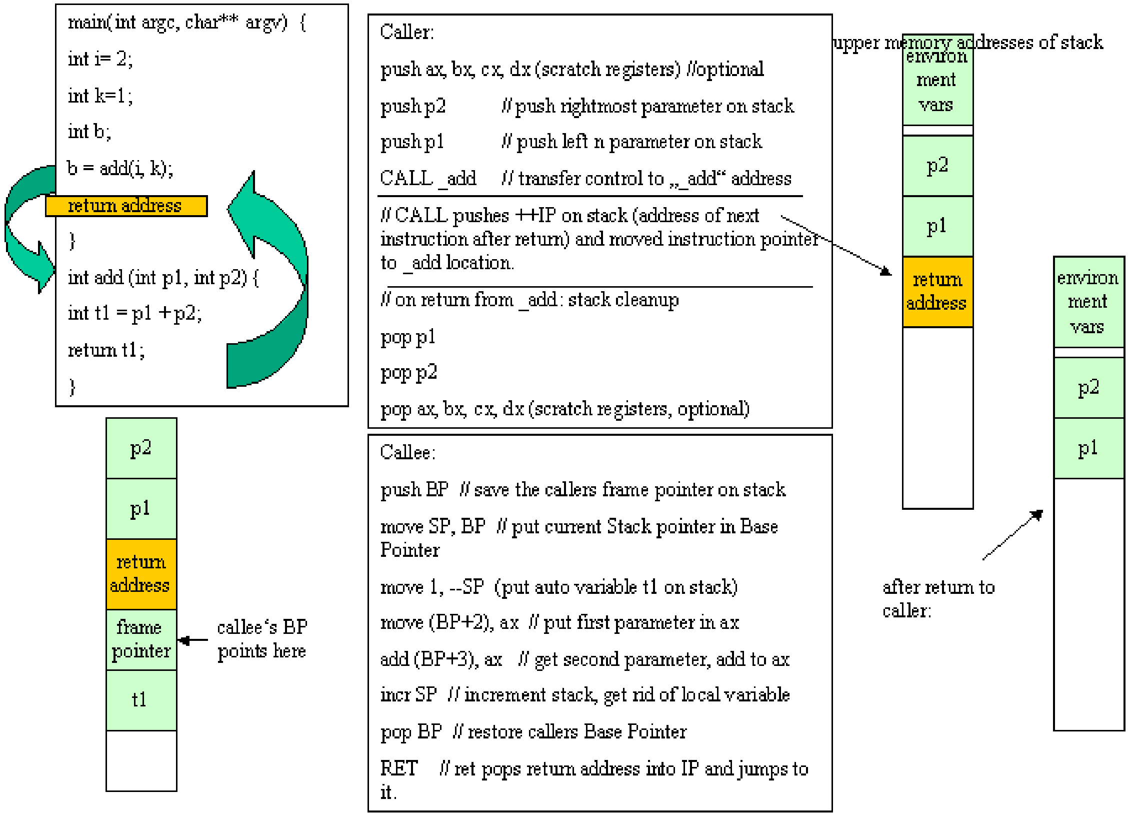
Remote Procedure Calls

In distributed systems we will see how this fast function call mechanism is broken up into 2 parts on different machines. We will call the callers part a "stub" and the receiving functions part a "skeleton" and complicated middleware will move the parameters back and forth. And this will be NOT AT ALL FAST because it involves a lot of networking.

Buffer Overflows

Most security attacks involve buffer overflows on the stack. To really understand how those attacks work you need to understand the stack mechanism, especially the return from a function call where an address from the stack becomes the new instruction pointer. If the stack is overwritten with buffer overflow code, an attacker can point the program at return right to its own virus code.

C Function Calling Convention



```
main( int argc, char** argv)  {
int i= 2;
int k=1;
int b;
b = add(i, k);
return address
}
int add (int p1, int p2) {
int t1 = p1 +p2;
return t1;
}
```

Caller:

```
push ax, bx, cx, dx (scratch registers) //optional
push p2        // push rightmost parameter on stack
push p1        // push left n parameter on stack
CALL _add      // transfer control to „_add“ address
```

// CALL pushes ++IP on stack (address of next instruction after return) and moved instruction pointer to _add location.

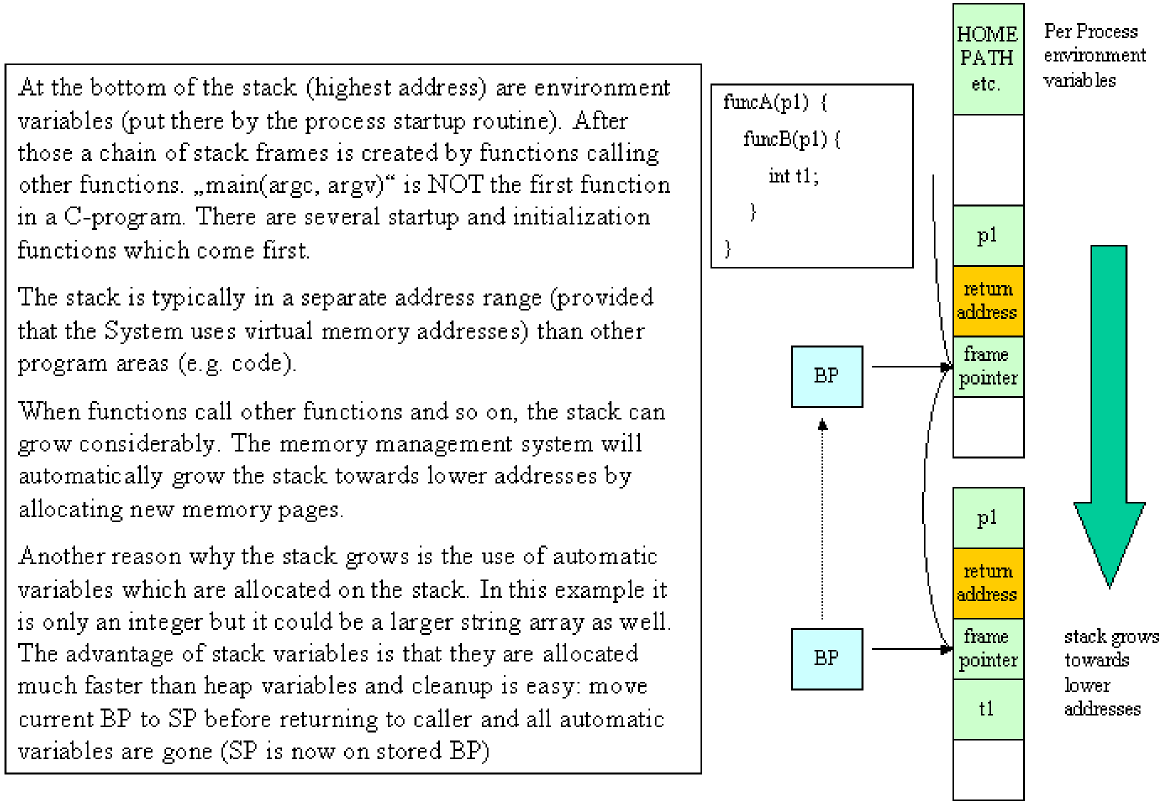
// on return from _add: stack cleanup

```
pop p1
pop p2
pop ax, bx, cx, dx (scratch registers, optional)
```

Callee:

```
push BP  // save the callers frame pointer on stack
move SP, BP  // put current Stack pointer in Base Pointer
move 1, --SP  (put auto variable t1 on stack)
move (BP+2), ax  // put first parameter in ax
add (BP+3), ax   // get second parameter, add to ax
incr SP  // increment stack, get rid of local variable
pop BP  // restore callers Base Pointer
RET   // ret pops return address into IP and jumps to it.
```

C stack layout



At the bottom of the stack (highest address) are environment variables (put there by the process startup routine). After those a chain of stack frames is created by functions calling other functions. „main(argc, argv)“ is NOT the first function in a C-program. There are several startup and initialization functions which come first.

The stack is typically in a separate address range (provided that the System uses virtual memory addresses) than other program areas (e.g. code).

When functions call other functions and so on, the stack can grow considerably. The memory management system will automatically grow the stack towards lower addresses by allocating new memory pages.

Another reason why the stack grows is the use of automatic variables which are allocated on the stack. In this example it is only an integer but it could be a larger string array as well. The advantage of stack variables is that they are allocated much faster than heap variables and cleanup is easy: move current BP to SP before returning to caller and all automatic variables are gone (SP is now on stored BP)

```
funcA(p1) {
   funcB(p1) {
      int t1;
   }
}
```

Debugging

Program crashes and core files

On Unix platforms if a program crashes a so called "core file" is created. It contains the memory status of the program when it crashed. This information can be used for post-mortem debugging using a debugger like gdb.

For best debugging results a program needs to be compiled with debugging turned on. This creates additional debugging instructions in the object code of the program and preserves all symbol and line information. This allows a debugger to show the proper source code pieces.

`man gcc` lists a large number of debugging and optimization options. Some compilers do not allow mixing debugging and optimization.

Pointer Arithmetics

Pointer arithmetic is frequently used in C programs which means that a C programmer must know how to do this. The basic concept is that a pointer variable (which contains the address of some piece of memory) can be modified by the programmer. Here really shows that pointers are special types because incrementing a pointer can give some surprising results.

```
int k= 3;
int* ptr = &k;  // ptr now contains the address of k
ptr++;  // incrementing the pointer makes it point to
        // address of k PLUS 4 byte.
```

The reason for this is that pointer arithmetic calculates with the SIZE OF THE TYPE the pointer is pointing to. The formula is: `pointer + x == ptr + (x * sizeof(pointerType)`

Memory layout and pointer arithmetics

Pointers and Arrays revisited

```
#include <stdio.h>
int main(void) {
 int* ptr;
 int my_array[] = {11,22,33,44};
 ptr = &my_array[0];
 for (i=0;i<4;i++) {
  printf("my_array[%d] = %d   ", i, my_array[i]);
  printf("ptr + %d  = %d\n", i, (*ptr+1));
 }
 return 0;
}
```

```
my_array[0] = 11  ptr + 0 = 11;
my_array[1] = 22  ptr + 4 = 22;
my_array[2] = 33  ptr + 8 = 33;
my_array[3] = 44  ptr + 12 = 44;
```

This shows the following facts about pointers and arrays:

1. **The address of an array at element 0 (&my_array[0]) is the same as the array name.**
2. **The following statements are therefore equivalent:** `ptr = &my_array[0];` and `ptr = my_array;`.

Pointers and Arrays revisited (Continued)

3. **An array is a pointer to the first element of the array.**
4. **But an array cannot be re-assigned like a pointer. While my_array == ptr the following code does not work:**

```
int my_array[5];
int* prt;
ptr = my_array;
my_array = ptr; // ERROR: an array is NOT an lvalue
```

Strings revisited

A string is an array of characters and therefore also a pointer of type pointer to character.

```
char myArray[] = {'a', 'b', 'c'};
char c = my_array[1];  // gives b
c = *(my_array + 1);   // gives b

int i;
for (i=0; i%lt; 3;i++) {
  printf("%c", *my_array++;)
  }  // prints "abc"
```

Dynamic memory allocation

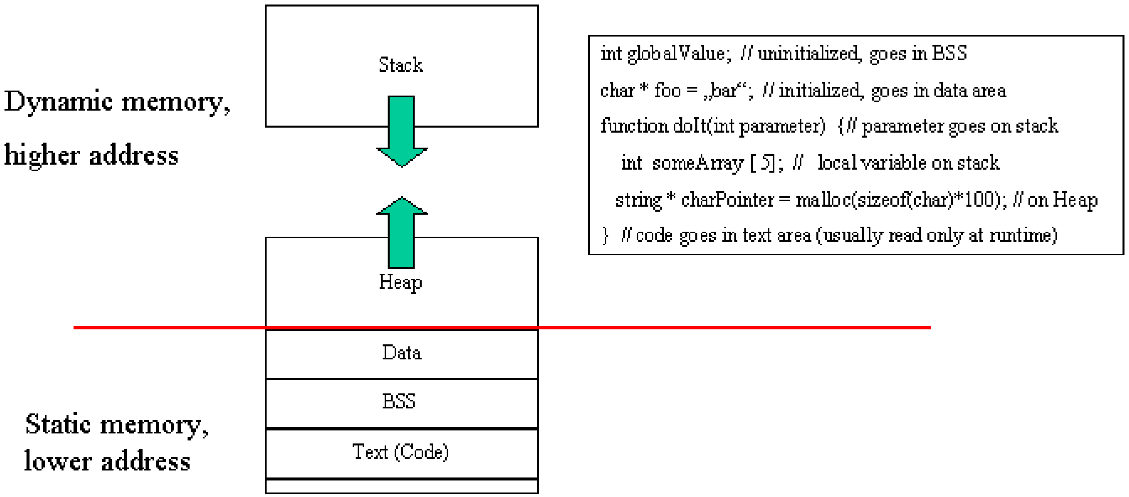
Almost every application needs to allocate memory dynamically. The data segment of an application does NOT grow. All variables there have been allocated at compiletime and the sizes are fixed. The area where the application gets dynamic memory is called heap. The heap starts at the end of the data segment and grows against the bottom of the stack. If both meet your program is in trouble and will be terminated by the operating system.

Note

Allocation of dynamic memory is THE area of application programming with the largest potential for bugs or very slow performance. At least the performance problem does also exist in languages with garbage collection.

Here is how dynamic memory allocation in C works: `void *malloc(size_t size)` allocates memory. Please check the return value for not being "null". A null indicates that the system is out of memory. And `void free(void *ptr)` gives the memory back to the pool.

Program Memory Areas



```
int globalValue; // uninitialized, goes in BSS
char * foo = „bar“; // initialized, goes in data area
function doIt(int parameter)  {// parameter goes on stack
   int  someArray [ 5]; //   local variable on stack
  string * charPointer = malloc(sizeof(char)*100); // on Heap
}  // code goes in text area (usually read only at runtime)
```

Please note that both stack and heap are dynamic. Heap grows upwards (caused by malloc() or new ClassXX() statements) and stack grows downwards. If the meet then you are in trouble. The tiny segment at the bottom of the virtual memory is called „zero page“ in some systems and serves to catch ininitialized pointer access. BSS content is usually initialized to 0 automatically at program start. Text is locked at program runtime to allow for pages being thrown out in low memory conditions. All other areas need to be stored in swap space if memory gets low because they may have changed during program runtime.

How malloc and free work

Some steps from a typical malloc call will show you how much work is performed here:

1. **A program does request memory from the heap with `int* ptr = (int*) malloc(1024 * sizeof(int));` . It expects a pointer back which points to a newly allocated areas on the heap which is at least big enough to hold 1024 integer values, no matter how big an integer on this platform is. If the program would ask for (1024 * 2) bytes it would assume 16 bit integer values and would not be portable to other hardware.**

2.

3. **The reason for malloc being a very expensive call has many facets. First, finding the proper area needs a clever memory organisation by malloc so that it will find those pieces fast. Remember, malloc does not know how much memory will be request. Could be one byte, could be thousands. The next problem appears when the current heap size becomes too small. Now malloc must ask the Operating System for more heap space, using a `brk or sbrk` system call. Now the operating system allocates physical memory and maps it into the process address space which belongs to the heap. Frequent allocations are expensive if done in small sizes, but how should malloc know?**

4. **Just to top it off: If this is a multi-threaded program chances are that malloc needs to protect its heap data structures from being corrupted by threads trying to allocate memory in parallel. This means expensive semaphore and monitor operations.**

Rules of thumb for application programmers in need of

1. **Malloc is generic and good. Many small allocations will work but they will slow your application down.**
2. **It may be better if the application allocates a large chunk of heap and maintains it by itself. It knows best what kind of allocations will happen. Try to avoid frequent small allocations via malloc.**
3. **Do not give memory back just to allocate it a few milliseconds later again. Keep your own memory pool and REUSE memory as much as possible. Caching and pooling is the name of the game for successful applications.**
4. **In languages without garbage collection: Define your own memory management policy and design a coding standard with function names that make it clear when somebody takes over responsibility for a piece of memory.**

```
char* value = getSomething(); // does the function expect the caller to call fre
                      // is the memory area a constant and the program
                      // will crash if free(value) is called?
                      // if value will be a parameter to another call, who
                      // will take over responsibility for free-ing it?
```

5.

Example of Memory Leak

```
#include <stdio.h>

int* create(int size) {
  return (int* malloc(size*sizeof(int));
}

int main(void) {
  int *arr = create(10);
  // do something useful with memory
  // e.g. maintain a dynamic list
  // free(arr); // this will cause a LEAK
  return 0;
}
```

This may look like stupid programming but it is actually very hard to maintain pairwise allocation/deallocation if both can happen in functions that are far away from each other in the software package.

Implementing a dynamic list with malloc

```
#include <stdio.h>
struct node {
    int i;
    struct node *next;
};
int main(void) {
 struct node* temp, *list = NULL;
 for(i=0;i<10;i++) {
  temp = (struct node *) malloc(sizeof(struct list));
  temp->i = i;
  temp->next = list;
  list = temp;
 }
  // do something with list and free nodes later
  return 0;
}
```

This is extremely common code in C programs. Freeing the list is easy because all nodes are hanging together via the next pointer.

C vs. Java Memory Management

At the first glance Java seems to solve many of the problems related to dynamic memory management in C. The garbage collector automatically collects objects which are no longer reachable.

```
public Class Foo {
 public int i;
 public String bar;

void someMethod() {
   Class Foo fooObject = new Foo(); // allocates a Foo object on heap
  } // and now the fooObject IS NO LONGER REACHABLE
    // It will be collected at some future time and
    // returned to the heap.
}
Foo someOtherMethod() {
   Class Foo fooObject = new Foo();
   return fooObject;// allocates a Foo object on heap
  } // fooObject is STILL reachable from the caller which
    // gets the reference to it as a return value
}
```

Things to know about Java Memory Management

stack vs. heap

ALL java object (except primitive types) are allocated on the HEAP. Creating a Class Foo instance means to allocate the 8 bytes for a Class Foo object from above (assuming that the VM uses 4 bytes to represent an integer and 4 bytes to represent a String reference. "Newing many small objects is exactly what we said is very bad for dynamic memory management. So don't think just because your objects are small allocating them costs nothing...

malloc vs. new

As a matter of fact, malloc is probably used to allocate the 8 bytes for a Class Foo object from above (assuming that the VM uses 4 bytes to represent an integer and 4 bytes to represent a String reference. You know now that malloc is expensive and so is new...

Garbage Collection vs. self-managed memory

There is ABSOLUTELY no doubt that garbage collection is superior for application programming. But this does not mean that it is free or that you can't have memory leaks with a GC. And the type of GC will impact your performance dramatically. (Compare a generational GC with an non-generational GC in the context of large caches for web applicationsl...) We will discuss this in depth in our Virtual Machine session. Frequent bugs with

Things to know about Java Memory Management

GCs are: Forgetting static variables which hold references to large objects which reference large objects and so on... Or calling system functions which keep references to user objects without telling about: Have a look at Java ThreadGroups...

So event with Java, get a memory leak checker which will also tell you if your singletons are no singletons or how long method calls really take. Be ready for some surprises...

Resources

The resources cover freely available information as well as excellent books right to the topic. All entries are commented to let you know what a paper or book is all about. I also expect participants to use this literature in case of questions.

Open Source Information on C programming

The following information is freely available and taken together is an excellent introduction to the subject.

1. Jason Maassen, C for Java Programmers. A complete introduction to C for Java people. I have used and extended his slides for this lecture but you should read the background information here as well.60+ pages. Good if you need to prepare for a test or need some more information about a feature from my slides. Find his C course with other materials here: http://www.cs.vu.nl/~jason/course.html [http://www.cs.vu.nl/~jason/course.html].
2. Steven Simpson, Learning C from Java. An experienced Java programmer will get the most from this short paper focussing on the differences. Excellent. http://www.comp.lancs.ac.uk/computing/users/ss/java2c/diffs.htm [http://www.comp.lancs.ac.uk/computing/users/ss/java2c/diffs.html].
3. Marshal Brain, How C programming works. Another excellent paper from www.howstuffworks.com. This really explains complicated memory problems using pointers, how array overwriting can happen etc. And many useful pointers to other computing related topics like memory organization, operating systems etc. www.howstuffworks.com

Books

I always try to have all recommended books available in our library. Also take a look at my special section there where I collect books which should be present at all times.

1. **Kernighan/Ritchie, Programming in C. The bible of c-programming from the inventors. A classic text. Very short- compare this to nowadays documentation bloat.**
2. **The C pocket reference. A short book covering the latest developments in C. In my library.**
3. **The C Puzzle Book. Alan R. Feuer. A very short and nice book full of examples with C.**