Lecture 4: Memory
Announcements

- Matt’s OH: Mon 12-1, Tues 12-2
- Hank’s OH: Weds 3:45-4:45, Fri 12:30-1:30
Outline

• Review
• Intro to Memory Segments
• Memory Allocation Basics
• Project 2A
• Advanced Memory Concepts / Memory Errors
Outline

• Review
• Intro to Memory Segments
• Memory Allocation Basics
• Project 2A
• Advanced Memory Concepts / Memory Errors
Build: The Actors

- **File types**
  - Source code
  - Object code
  - Executable code

- **Programs**
  - Compiler
  - Linker
Compilers, Object Code, and Linkers

• Compilers transform source code to object code
  – Confusing: most compilers also secretly have access to linkers and apply the linker for you.
• Object code: statements in machine code
  – not executable
  – intended to be part of a program
• Linker: turns object code into executable programs
GNU Compilers

- GNU compilers: open source
  - gcc: GNU compiler for C
  - g++: GNU compiler for C++
gcc flag “-c”: make object code, but don’t link

• gcc –c file1.c
  – makes object code file “file1.o”
Multi-file development: example

```c
fawcett:330 childs$ cat t1.c
int doubler(int x)
{
    return 2*x;
}
fawcett:330 childs$ cat t2.c
int main()
{
    return doubler(5);
}
fawcett:330 childs$ gcc -c t1.c
fawcett:330 childs$ gcc -c t2.c
fawcett:330 childs$ gcc -o both t2.o t1.o
fawcett:330 childs$ ./both
fawcett:330 childs$ echo $? 10
```

cat is a Unix command that prints the contents of a file

$? is a shell construct that has the return value of the last executed program
Multi-file development: example

```c
fawcett:330 childs$ cat t1.c
int doubler(int x)
{
    return 2*x;
}
fawcett:330 childs$ cat t2.c
int main()
{
    return doubler(5);
}
fawcett:330 childs$ gcc -c t1.c
fawcett:330 childs$ gcc -c t2.c
fawcett:330 childs$ gcc -o both t2.o t1.o
fawcett:330 childs$ ./both
fawcett:330 childs$ echo $?
10
```
Multi-file development: example

```bash
fawcett:330 childs$ cat t1.c
int doubler(int x)
{
    return 2*x;
}
fawcett:330 childs$ cat t2.c
int main()
{
    return doubler(5);
}
fawcett:330 childs$ gcc -c t1.c
fawcett:330 childs$ gcc -c t2.c
fawcett:330 childs$ gcc -o both t1.o t2.o
fawcett:330 childs$ ./both
fawcett:330 childs$ echo $?
10
```

Linker order matters for some linkers (not Macs). Some linkers need the .o with "main" first and then extract the symbols they need as they go. Other linkers make multiple passes.
Libraries

• Library: collection of “implementations” (functions!) with a well defined interface

• Interface comes through “header” files.

• In C, header files contain functions and variables.
  – Accessed through “#include <file.h>”
Making a static library

Note the ‘#’ is the comment character

```bash
C02LN00GFD58:multiplier hank$ cat doubler.h # here's the header file
int doubler(int);
int tripler(int);
C02LN00GFD58:multiplier hank$ cat doubler.c # here's one of the c files
int doubler(int x) {return 2*x;}
C02LN00GFD58:multiplier hank$ cat tripler.c # here's the other c files
int tripler(int x) {return 3*x;}
C02LN00GFD58:multiplier hank$ gcc -c doubler.c # make an object file
C02LN00GFD58:multiplier hank$ ls doubler.o # we now have a .o
C02LN00GFD58:multiplier hank$ gcc -c tripler.c
C02LN00GFD58:multiplier hank$ ar r multiplier.a doubler.o tripler.o
C02LN00GFD58:multiplier hank$ (should have called this libmultiplier.a)
```
Typical library installations

• Convention
  – Header files are placed in “include” directory
  – Library files are placed in “lib” directory

• Many standard libraries are installed in /usr
  – /usr/include
  – /usr/lib

• Compilers automatically look in /usr/include and /usr/lib (and other places)
Installing the library

```
C02LN00GFD58:multiplier hank$ mkdir ~/multiplier
C02LN00GFD58:multiplier hank$ mkdir ~/multiplier/include
C02LN00GFD58:multiplier hank$ cp multiplier.h ~/multiplier/include/
C02LN00GFD58:multiplier hank$ mkdir ~/multiplier/lib
C02LN00GFD58:multiplier hank$ cp doubler.c multiplier.a tripler.c
doubler.o multiplier.h tripler.o
C02LN00GFD58:multiplier hank$ cp multiplier.a ~/multiplier/
C02LN00GFD58:multiplier hank$ mv multiplier.a libmultiplier.a
C02LN00GFD58:multiplier hank$ cp libmultiplier.a ~/multiplier/lib/
```

“mv”: unix command for renaming a file
Example: compiling with a library

```c
C02LN00GFD58:CIS330 hank$ cat t.c
#include <multiplier.h>
#include <stdio.h>
int main()
{
    printf("Twice 6 is %d, triple 6 is %d\n", doubler(6), tripler(6));
}
C02LN00GFD58:CIS330 hank$ gcc -o mult_example t.c -I/Users/hank/multiplier/inclu
de -L/Users/hank/multiplier/lib -lmultiplier
C02LN00GFD58:CIS330 hank$ ./mult_example
Twice 6 is 12, triple 6 is 18
C02LN00GFD58:CIS330 hank$
```

- gcc support for libraries
  - "-I": path to headers for library
  - "-L": path to library location
  - "-lname": link in library libname
Makefiles

• There is a Unix command called “make”
• make takes an input file called a “Makefile”
• A Makefile allows you to specify rules
  – “if timestamp of A, B, or C is newer than D, then carry out this action” (to make a new version of D)
• make’s functionality is broader than just compiling things, but it is mostly used for computation

Basic idea: all details for compilation are captured in a configuration file ... you just invoke “make” from a shell
Makefiles

• Reasons Makefiles are great:
  – Difficult to type all the compilation commands at a prompt
  – Typical develop cycle requires frequent compilation
  – When sharing code, an expert developer can encapsulate the details of the compilation, and a new developer doesn’t need to know the details ... just “make”
Makefile syntax

• Makefiles are set up as a series of rules
• Rules have the format:
  target: dependencies
  [tab] system command

Target: what to build. (a name you give to make.)
Dependencies: what it depends on (files in the filesystem or other rules)
System command: gcc ...
Makefile example: multiplier lib

C02LN00GFD58: code hank$
lib: doubler.o tripler.o
  ar r libmultiplier.a doubler.o tripler.o
  cp libmultiplier.a ~/multiplier/lib
  cp multiplier.h ~/multiplier/include

doubler.o: doubler.c
  gcc -c doubler.c

tripler.o: tripler.c
  gcc -c tripler.c
C02LN00GFD58: code hank$
C02LN00GFD58: code hank$
C02LN00GFD58: code hank$
C02LN00GFD58: code hank$ make
ar r libmultiplier.a doubler.o tripler.o
cp libmultiplier.a ~/multiplier/lib
cp multiplier.h ~/multiplier/include
C02LN00GFD58: code hank$
C02LN00GFD58: code hank$ touch doubler.c
C02LN00GFD58: code hank$ make
gcc -c doubler.c
ar r libmultiplier.a doubler.o tripler.o
cp libmultiplier.a ~/multiplier/lib
cp multiplier.h ~/multiplier/include
C02LN00GFD58: code hank$
Configuration management tools

• Problem:
  – Unix platforms vary
    • Where is libX installed?
    • Is OpenGL supported?

• Idea:
  – Write problem that answers these questions, then adapts build system
    • Example: put “-L/path/to/libX –lX” in the link line
    • Other fixes as well
Two popular configuration management tools

• Autoconf
  – Unix-based
  – Game plan:
    • You write scripts to test availability on system
    • Generates Makefiles based on results

• Cmake
  – Unix and Windows
  – Game plan:
    • You write .cmake files that test for package locations
    • Generates Makefiles based on results

CMake has been gaining momentum in recent years, because it is one of the best solutions for cross-platform support.
Unix command: tar

• Problem: you have many files and you want to...
  – move them to another machine
  – give a copy to a friend
  – etc.

• Tar: take many files and make one file
  – Originally so one file can be written to tape drive

• Serves same purpose as “.zip” files.
Unix command: tar

- tar cvf 330.tar file1 file2 file3
- scp 330.tar @ix:~
- ssh ix
- tar xvf 330.tar
- ls
  - file1 file2 file
CIS 330: Project #1C
Assigned: April 8th, 2015
Due April 12th, 2015
(which means submitted by 6am on April 13th, 2015)
Worth 2% of your grade

Assignment: Download the file “Proj1C.tar”. This file contains a C-based project. You will build a Makefile for the project, and also extend the project.
Project 1C

== Build a Makefile for math330 ==

Your Makefile should:
   1. create an include directory
   2. copy the Header file to the include directory
   3. create a lib directory
   4. compile the .c files in trig and exp as object files (.o’s)
   5. make a library
   6. install the library to the lib directory
   7. compile the “cli” program against the include and library directory

== Extend the math330 library ==

You should:
   1. add 3 new functions: arccos, arcsin, and arctan (each in their own file)
   2. Extend the “cli” program to support these functions
   3. Extend your Makefile to support the new functions
Outline

• Review
• Intro to Memory Segments
• Memory Allocation Basics
• Project 2A
• Advanced Memory Concepts / Memory Errors
Memory Segments

• Von Neumann architecture: one memory space, for both instructions and data
• so break memory into “segments”
  – ... creates boundaries to prevent confusion
• 4 segments:
  – Code segment
  – Data segment
  – Stack segment
  – Heap segment
Code Segment

• Contains assembly code instructions
• Also called text segment
• This segment is modify-able, but that’s a bad idea
  – “Self-modifying code”
    • Typically ends in a bad state very quickly.
Data Segment

• Contains data not associated with heap or stack
  – global variables
  – statics (to be discussed later)
  – character strings you’ve compiled in
    char *str = “hello world\n”
Stack: data structure for collection

• A stack contains things
• It has only two methods: push and pop
  – Push puts something onto the stack
  – Pop returns the most recently pushed item (and removes that item from the stack)
• LIFO: last in, first out

Imagine a stack of trays.
You can place on top (push).
Or take one off the top (pop).
Stack

• Stack: memory set aside as scratch space for program execution

• When a function has local variables, it uses this memory.
  – When you exit the function, the memory is lost
Stack

• The stack grows as you enter functions, and shrinks as you exit functions.
  – This can be done on a per variable basis, but the compiler typically does a grouping.
    • Some exceptions (discussed later)

• Don’t have to manage memory: allocated and freed automatically
Heap

• Heap (data structure): tree-based data structure

• Heap (memory): area of computer memory that requires explicit management (malloc, free).

• Memory from the heap is accessible any time, by any function.
  – Contrasts with the stack
Memory Segments

- text (fixed size)
- data (fixed size)
- stack | growth
- free
- heap | growth

Source: http://www.cs.uwm.edu/classes/cs315/Bacon/
Outline

• Review
• Intro to Memory Segments
• Memory Allocation Basics
• Project 2A
• Advanced Memory Concepts / Memory Errors
C: must manage your own memory

• This is a big change from other programs
• You keep track of memory
  – Allocation
  – How much there is / indexing memory
  – Deallocation
malloc

• malloc: command for allocating memory

```c
#include <stdlib.h>

void *
calloc(size_t count, size_t size);

void
free(void *ptr);

void *
malloc(size_t size);

void *
realloc(void *ptr, size_t size);

void *
reallocf(void *ptr, size_t size);

void *
valloc(size_t size);
```

DESCRIPTION

The `malloc()`, `calloc()`, `valloc()`, `realloc()`, and `reallocf()` functions allocate memory. The allocated memory is aligned such that it can be used for any data type, including Altivec- and SSE-related types. The `free()` function frees allocations that were created via the preceding allocation functions.

The `malloc()` function allocates `size` bytes of memory and returns a pointer to the allocated memory.
Allocation / Deallocation Example

```c
#include <stdlib.h>
int main()
{
    int stack_varA;
    int stack_varB[2];
    int *heap_varA;
    int *heap_varB;
    heap_varA = malloc(sizeof(int));
    heap_varB = malloc(sizeof(int)*2);
    free(heap_varA);
    free(heap_varB);
}
```

Automatic allocation on the stack. (Deallocation occurs when out of scope.)

Explicit allocation from the heap. (Deallocation occurs with “free” call.)
• `sizeof`: gets size of type

• Usually:
  – `sizeof(int) == 4`
  – `sizeof(float) == 4`
  – `sizeof(double) == 8`
  – `sizeof(unsigned char) == 1`
  – `sizeof(char) == 1`

• → array of 10 ints → `malloc(10* sizeof(int))`
Hexadecimal

- Binary: 2 values
- Decimal: 10 values
- Hexadecimal: 16 values
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- 0x: prefix for hexadecimal
  - 0x10 = 16
  - 0x101 = 257
Memory Addresses

• Every location in memory has an address associated with it
• Locations in memory are represented in hexadecimal

Memory addresses descend in the stack, ascend in the heap.
Pointers

• Pointers store locations in memory
Pointers

• Pointers store locations in memory

• “&”: unary operator that gives the address of a variable.

int x;
int *yp = &x;
Pointers

- Pointers store locations in memory

```c
#include <stdio.h>

int main()
{
    int x, y;
    printf("The location of x is %p and the location of y is %p\n", &x, &y);
}
```

```
C02LN00GFD58:330 hank$ cat pointer.c
#include <stdio.h>

int main()
{
    int x, y;
    printf("The location of x is %p and the location of y is %p\n", &x, &y);
}
```

```
C02LN00GFD58:330 hank$ gcc pointer.c
C02LN00GFD58:330 hank$ ./a.out
The location of x is 0x_7fff56d26bce and the location of y is 0x7fff56d26bc8
```

`printf` prints pointers with "%p"
NULL pointer

- NULL: defined by compiler to be a location that is not valid.
  - Typically 0x00000000
- You can use NULL to initialize pointers, and also to check to see whether a pointer is set already.

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *ptr = NULL;
    while (1)
    {
        char c;
        c = getchar();
        if (c == 'A')
        {
            if (ptr == NULL)
            {
                printf("Allocating!\n");
                ptr = malloc(100*sizeof(int));
            }
            else
                printf("Already allocated\n");
        }
    }
}
```

IBM team I worked on used 0xDEADBEEF, not NULL
‘*’ operator

• Let “ptr” be a pointer
• Then “*ptr” returns value in the address that pts points to.
• * = “dereference operator”

```c
#include <stdio.h>

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

```
C02LN00GFD58:330 hank$ cat ptr.c
C02LN00GFD58:330 hank$ gcc ptr.c
C02LN00GFD58:330 hank$ ./a.out
x = 3, z = 3
```
Behavior of dereference

• When you dereference, you get the value at that moment.
  – Whatever happens afterwards won’t have effect.

```c
#include <stdio.h>

int main()
{
    int x = 3;
    int *y = &x;
    int z = *y;
    x = 4;
    printf("x = %d, y = %d, z = %d\n", x, *y, z);
}
```

C02LN00GFD58:330 hank$ gcc ptr2.c
C02LN00GFD58:330 hank$ ./a.out
x = 4, y = 4, z = 3
Pointer Arithmetic

• You can combine pointers and integers to get new pointer locations

```c
#include <stdio.h>

int main()
{
    int x = 3;
    int *y = &x;
    int *z = y+1;
    char a = 'A';
    char *b = &a;
    char *c = b+1;
    printf("x = %d, y = %p, z = %p\n", x, y, z);
    printf("a = %c, b = %p, c = %p\n", a, b, c);
}
```

```bash
$ gcc ptr_arith.c
$ ./a.out
x = 3, y = 0x7fff5d397bcc, z = 0x7fff5d397bd0
a = A, b = 0x7fff5d397bb7, c = 0x7fff5d397bb8
```
Arrays

- Arrays: container that has multiple elements of identical type, all stored in contiguous memory

```c
int A[10];
```

→ 10 integers, stored in 40 consecutive bytes (assuming `sizeof(int) == 4`)

**Arrays are just pointers. You can use arrays and pointers interchangeably.**
[ ] operator

• [ ] is a way of dereferencing memory
  – Recall that ‘*’ is the dereference operator

• A[0] <= => *A
• A[5] <= => *(A+5);
More array relationships

```c
int A[10];
int *B;

B = &(A[0]) → B = A
B = &(A[5]) → B = A + 5
```
Pointers to pointers

• Remember: pointer points to a location in memory
  – We’ve been considering cases where locations in memory are arrays of integers
  – But locations in memory could be pointer themselves
Simple pointers to pointers example

C02LN00GFD58:330 hank$ cat prptrpr.c
#include <stdlib.h>
int main()
{
    int **X = malloc(sizeof(int *)*4);
    X[0] = malloc(sizeof(int)*6);
    X[1] = malloc(sizeof(int)*4);
    X[2] = malloc(sizeof(int)*8);
    X[3] = malloc(sizeof(int)*10);
}
C02LN00GFD58:330 hank$ gcc prptrpr.c
C02LN00GFD58:330 hank$ ./a.out
What’s the difference between these two programs?

```c
int main()
{
    int **X = malloc(sizeof(int *)*4);
    X[0] = malloc(sizeof(int)*6);
    X[1] = malloc(sizeof(int)*4);
    X[2] = malloc(sizeof(int)*8);
    X[3] = malloc(sizeof(int)*10);
}
```

Answer: X is on the heap on the left, and on the stack on the right. But they are both pointers-to-pointers.
What’s the difference between these two programs?

**Program on left**

```c
#include <stdlib.h>

int main()
{
    int **X = malloc(sizeof(int *)*4);
    X[0] = malloc(sizeof(int)*6);
    X[1] = malloc(sizeof(int)*4);
    X[2] = malloc(sizeof(int)*8);
    X[3] = malloc(sizeof(int)*10);
}
```

**Program on right**

```c
#include <stdlib.h>

int main()
{
    int **X = malloc(sizeof(int *)*4);
    int num = 6+4+8+10;
    int *allMem = malloc(sizeof(int)*num);
    X[0] = allMem;
    X[1] = X[0]+6;
}
```

Answer: program on left makes one allocation for each pointer, program on right makes one allocation for whole program & each pointer points at locations within that allocation.
Call by reference / call by value

• Refers to how parameters are passed to a function.
  – Call by reference: send a reference (pointer) as a function parameter
    • Side effects in that function affect the variable in the calling function
  – Call by value: send the value of the variable as a function parameter
    • Side effects in that function don’t affect the variable in the calling function
Call by Reference

C02LN00GFD58:330 hank$ cat cbr.c
#include <stdio.h>

void foo(int *x)
{
    *x = *x+1;
}

int main()
{
    int x = 2;
    foo(&x);
    printf("X is \%d\n", x);
}
C02LN00GFD58:330 hank$ gcc cbr.c
C02LN00GFD58:330 hank$ ./a.out
X is 3
Call by Value

```c
C02LN00GFD58:330 hank$ cat cbv.c
#include <stdio.h>

void foo(int x)
{
    x = x+1;
}

int main()
{
    int x = 2;
    foo(x);
    printf("X is %d\n", x);
}
C02LN00GFD58:330 hank$ gcc cbv.c
C02LN00GFD58:330 hank$ ./a.out
X is 2
```
Outline

• Review
• Intro to Memory Segments
• Memory Allocation Basics
• Project 2D
• Advanced Memory Concepts / Memory Errors
Called Project 2D since I screwed up and printed a bunch of them

CIS 330: Project #2D
Assigned: April 10th, 2015
Due: 2pm April 15th, 2015. NO LATE PASSES, THIS IS DUE IN CLASS
Worth 3% of your grade.

Assignment: fill out this worksheet.

<table>
<thead>
<tr>
<th>Location</th>
<th>0x8000</th>
<th>0x8004</th>
<th>0x8008</th>
<th>0x800c</th>
<th>0x8010</th>
<th>0x8014</th>
<th>0x8018</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>Location</td>
<td>0x801c</td>
<td>0x8020</td>
<td>0x8024</td>
<td>0x8028</td>
<td>0x802c</td>
<td>0x8030</td>
<td>0x8034</td>
</tr>
<tr>
<td>Value</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>144</td>
<td>233</td>
</tr>
<tr>
<td>Location</td>
<td>0x8038</td>
<td>0x803c</td>
<td>0x8040</td>
<td>0x8044</td>
<td>0x8048</td>
<td>0x804c</td>
<td>0x8050</td>
</tr>
<tr>
<td>Value</td>
<td>377</td>
<td>610</td>
<td>987</td>
<td>1597</td>
<td>2584</td>
<td>4181</td>
<td>6765</td>
</tr>
</tbody>
</table>

Code:
int *A = 0x8000;

Note: “NOT ENOUGH INFO” is a valid answer.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Your Answer</th>
<th>Variable</th>
<th>Your Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0x8000</td>
<td>(A+6)-(A+3)</td>
<td></td>
</tr>
<tr>
<td>&amp;A</td>
<td>NOT ENOUGH INFO</td>
<td><em>(A+6)-</em>(A+4)</td>
<td></td>
</tr>
</tbody>
</table>
Outline

• Review
• Intro to Memory Segments
• Memory Allocation Basics
• Project 2A
• Advanced Memory Concepts / Memory Errors
# Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Dealocation</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
</tbody>
</table>
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A) {
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main() {
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A) {
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main() {
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```

Return copies into location specified by calling function
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```
This code is very problematic ... why?

```c
int *foo()
{
    int stack_varC[2] = { 0, 1 };
    return stack_varC;
}

int *bar()
{
    int stack_varD[2] = { 2, 3 };  // foo and bar are returning addresses that are on the stack ... they could easily be overwritten (and bar’s stack_varD overwrites foo’s stack_varC in this program)
    return stack_varD;
}

int main()
{
    int *stack_varA, *stack_varB;
    stack_varA = foo();
    stack_varB = bar();
    stack_varA[0] *= stack_varB[0];
}
```
int main() {
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
You can create new scope within a function by adding '{' and '}'.

```c
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
```
# Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deallocation</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td>Access</td>
<td>Fast</td>
<td>Slower</td>
</tr>
</tbody>
</table>

Memory pages associated with stack are almost always immediately available.

Memory pages associated with heap may be located anywhere ... may be caching effects.
# Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deallocation</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td>Access</td>
<td>Fast</td>
<td>Slower</td>
</tr>
<tr>
<td>Variable scope</td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
</tbody>
</table>
Variable scope: stack and heap

```c
int *foo()
{
    int stack_varA[2] = { 0, 1 };
    return stack_varA;
}

int *bar()
{
    int *heap_varB;
    heap_varB = malloc(sizeof(int)*2);
    heap_varB[0] = 2;
    heap_varB[1] = 2;
    return heap_varB;
}

int main()
{
    int *stack_varA;
    int *stack_varB;
    stack_varA = foo(); /* problem */
    stack_varB = bar(); /* still good */
}
```

bar returned memory from heap

The calling function – i.e., the function that calls `bar` – must understand this and take responsibility for calling `free`.

If it doesn’t, then this is a “memory leak”.

Memory leaks

It is OK that we are using the heap ... that’s what it is there for.

The problem is that we lost the references to the 49 allocations on heap.

The heap’s memory manager will not be able to re-claim them ... we have effectively limited the memory available to the program.

```c
{
    int i;
    int stack_varA;
    for (i = 0 ; i < 50 ; i++)
        stack_varA = bar();
}
```
Running out of memory (stack)

```c
int endless_fun()
{
    endless_fun();
}

int main()
{
    endless_fun();
}
```

stack overflow: when the stack runs into the heap.
There is no protection for stack overflows.
(Checking for it would require coordination with the heap’s memory manager on every function calls.)
Running out of memory (heap)

```c
int *heaps_o_fun()
{
    int *heap_A = malloc(sizeof(int)*1000000000);
    return heap_A;
}

int main()
{
    int *stack_A;
    stack_A = heaps_o_fun();
}
```

If the heap memory manager doesn’t have room to make an allocation, then `malloc` returns `NULL` .... a more graceful error scenario.
## Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deallocation</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td>Access</td>
<td>Fast</td>
<td>Slower</td>
</tr>
<tr>
<td>Variable scope</td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Fragmentation</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Memory Fragmentation

• Memory fragmentation: the memory allocated on the heap is spread out of the memory space, rather than being concentrated in a certain address space.
Memory Fragmentation

```c
int *bar()
{
    int *heap_varA;
    heap_varA = malloc(sizeof(int)*2);
    heap_varA[0] = 2;
    heap_varA[1] = 2;
    return heap_varA;
}

int main()
{
    int i;
    int stack_varA[50];
    for (i = 0; i < 50; i++)
        stack_varA[i] = bar();
    for (i = 0; i < 25; i++)
        free(stack_varA[i*2]);
}
```

Negative aspects of fragmentation?
(1) can’t make big allocations
(2) losing cache coherency
Even if there is lots of memory available, the memory manager can only accept your request if there is a big enough contiguous chunk.
# Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deallocation</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td>Access</td>
<td>Fast</td>
<td>Slower</td>
</tr>
<tr>
<td>Variable scope</td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Fragmentation</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Memory Errors

• Array bounds read
  
  ```java
  int main()
  {
    int var;
    int arr[3] = { 0, 1, 2 };
    var=arr[3];
  }
  ```

• Array bounds write
  
  ```java
  int main()
  {
    int var = 2;
    int arr[3];
    arr[3]=var;
  }
  ```
Memory Errors

• Free memory read / free memory write

```c
int main()
{
    int *var = malloc(sizeof(int)*2);
    var[0] = 0;
    var[1] = 2;
    free(var);
    var[0] = var[1];
}
```

When does this happen in real-world scenarios?
Memory Errors

- Freeing unallocated memory

```c
int main()
{
    int *var = malloc(sizeof(int)*2);
    var[0] = 0;
    var[1] = 2;
    free(var);
    free(var);
}
```

When does this happen in real-world scenarios?

Vocabulary: “dangling pointer”: pointer that points to memory that has already been freed.
Memory Errors

• Freeing non-heap memory

```c
int main()
{
    int var[2]
    var[0] = 0;
    var[1] = 2;
    free(var);
}
```

When does this happen in real-world scenarios?
Memory Errors

• NULL pointer read / write
  int main()
  {
    char *str = NULL;
    printf(str);
    str[0] = 'H';
  }

• NULL is never a valid location to read from or write to, and accessing them results in a “segmentation fault”
  – .... remember those memory segments?

When does this happen in real-world scenarios?
Memory Errors

• Uninitialized memory read

```c
int main()
{
    int *arr = malloc(sizeof(int)*10);
    int V2=arr[3];
}
```