CSE306
SOFTWARE QUALITY IN PRACTICE

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

www.cse.buffalo.edu/faculty/alphonce/FA24/CSE306

LATE JOINERS

- Today is the last day for Add/Drop
- TopHat and AutoLab rosters will reflect add/drop changes as of last night
- If you missed your lab session, do the LEX as soon as you can on your own time: post questions and requests for assistance in Piazza.
- We will NOT be strict on the deadlines for LEX01, LEX02, and LEX03 (to accommodate students registering through end of add/drop)

ANNOUNCEMENTS

- Team formation will be finalized by tomorrow.
    - if you wish to pick your teammates form your team before 5:00 PM today
    - after 5:00 PM today I will assign remaining students to teams
- PRE (the first team project) will be posted on the course website as soon as team assignments are completed.
- Every team will have a Piazza group useful for intra-team communication, necessary for team-staff communication.

TEXT, PG 8

The 13 Golden Rules of Debugging

1. Understand the requirements
2. Make it fail
3. Simplify the test case
4. Read the right error message
5. Check the plug
6. Separate facts from interpretation
7. Divide and conquer
8. Match the tool to the bug
9. One change at a time
10. Keep an audit trail
11. Get a fresh view
12. If you didn't fix it, it ain't fixed
13. Cover your bugfix with a regression test

CLASSIFICATION OF BUGS

- Common bugs (source code, predictable)
- Sporadic bugs (intermittent)
- Heisenbugs (averse to observation)
    - race conditions
    - memory access violations
    - (programmer) optimizations
- Multiple bugs - several must be fixed before program behavior changes - consider violating rule #9 "one change at a time"

WHY HEISENBUGS?
THE UNCERTAINTY PRINCIPLE...

...the uncertainty principle, also known as Heisenberg's uncertainty principle, is any of a variety of mathematical inequalities[1] asserting a fundamental limit to the precision with which certain pairs of physical properties of a particle, known as complementary variables, such as position x and momentum p, can be known.

https://en.wikipedia.org/wiki/Uncertainty_principle

OBSERVER EFFECT

…the term observer effect refers to changes that the act of observation will make on a phenomenon being observed. This is often the result of instruments that, by necessity, alter the state of what they measure in some manner.

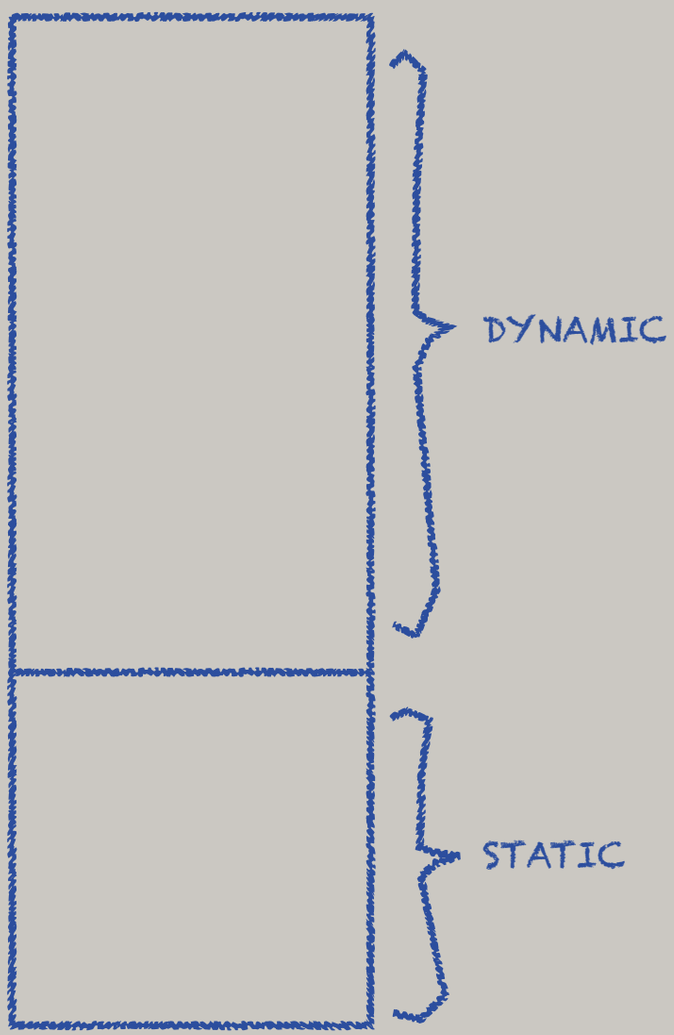
https://en.wikipedia.org/wiki/Observer_effect_(physics)

DEBUGGING TOOLS

- instrument code during compilation
- instrumented code may behave differently than uninstrumented code
- in other words: the act of using a debugger may mask a bug, causing its symptoms to disappear, only to reappear when run without instrumentation

MEMORY ORGANIZATION

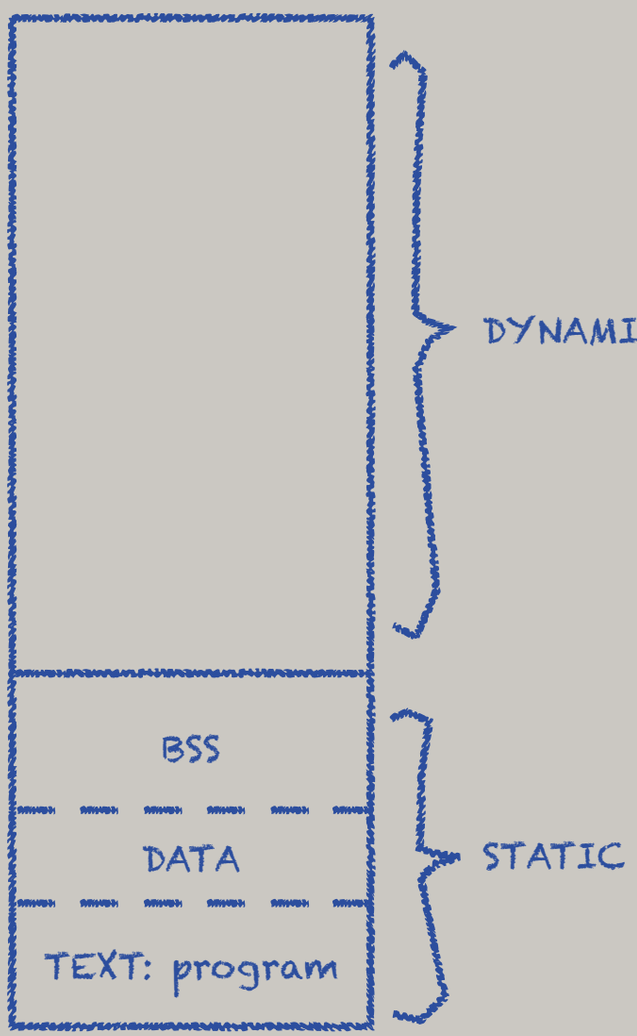
MEMORY ORGANIZATION



Each process (a running program) has a chunk of memory at its disposal.

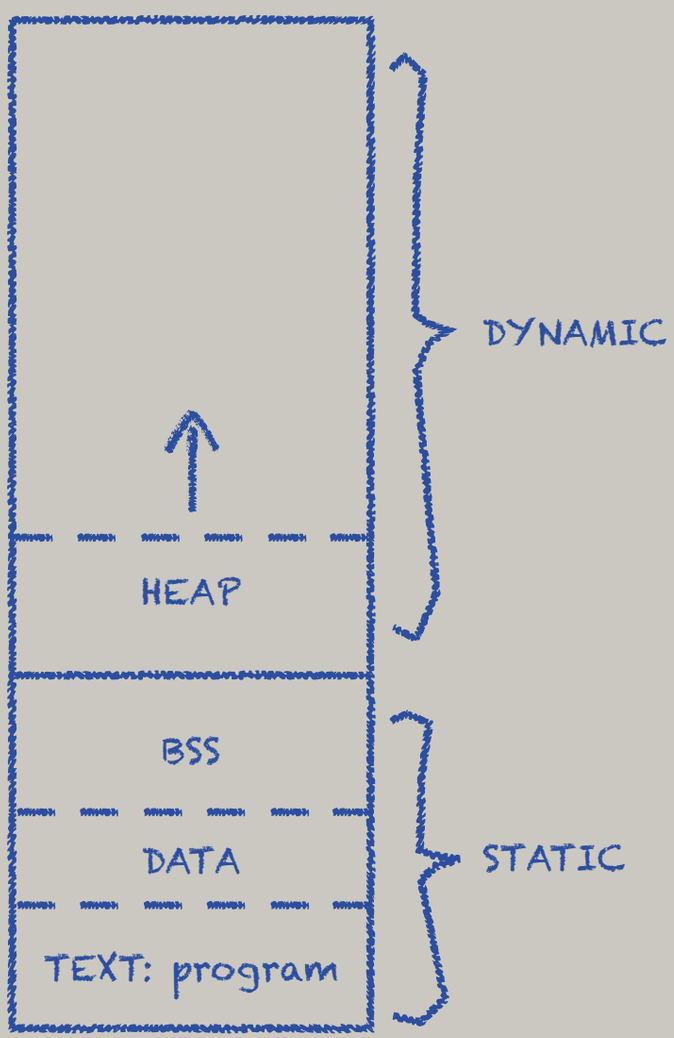
This memory is divided into "static" memory (allocated/structured before execution begins) and "dynamic" memory (allocated while the program executes.

MEMORY ORGANIZATION



The static segment is divided into a TEXT segment (holding the machine language instructions of the program), a DATA segment (for initialized variables), and a BSS segment (for uninitialized but implicitly zero-assigned values).

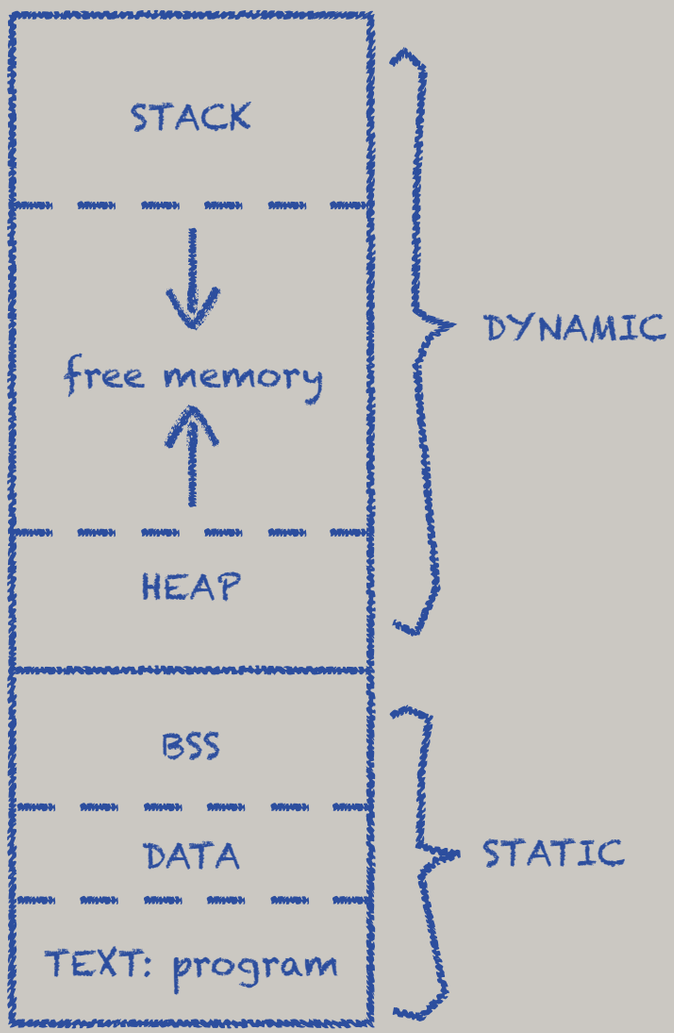
MEMORY ORGANIZATION



The dynamic segment is divided into STACK and a HEAP areas.

The HEAP is generally located adjacent to the STATIC segment, and grows "up" (to higher memory addresses).

MEMORY ORGANIZATION

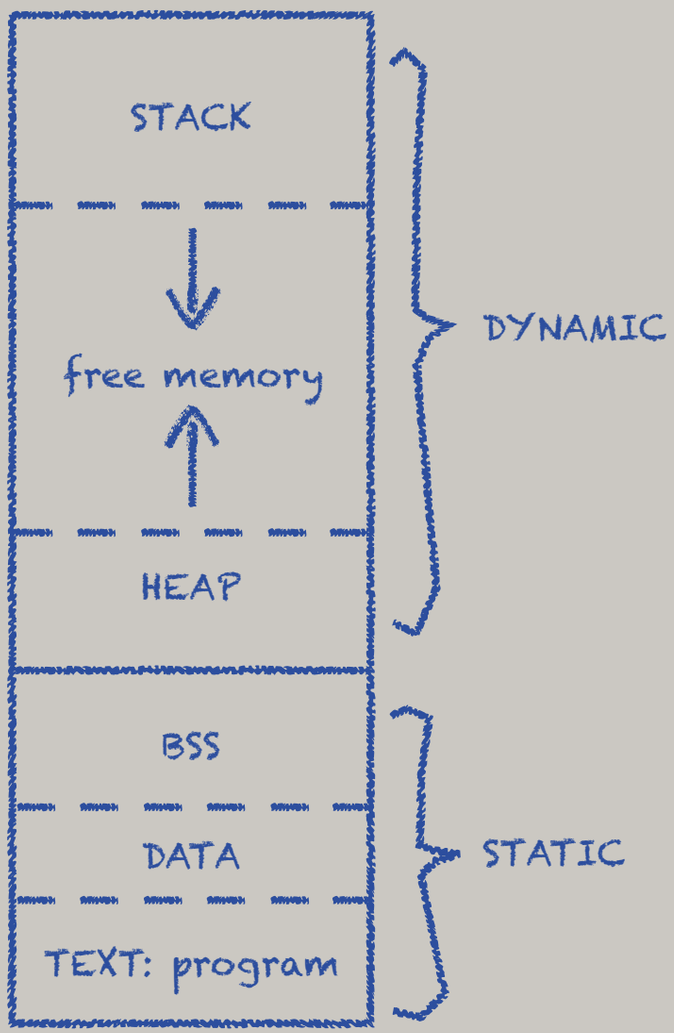


The STACK is generally located at the far end of memory and grows "down" (to lower memory addresses).

The area between the HEAP and the STACK represents available (free) memory in the processes' virtual memory.

It the HEAP and STACK collide we have an out-of-memory error.

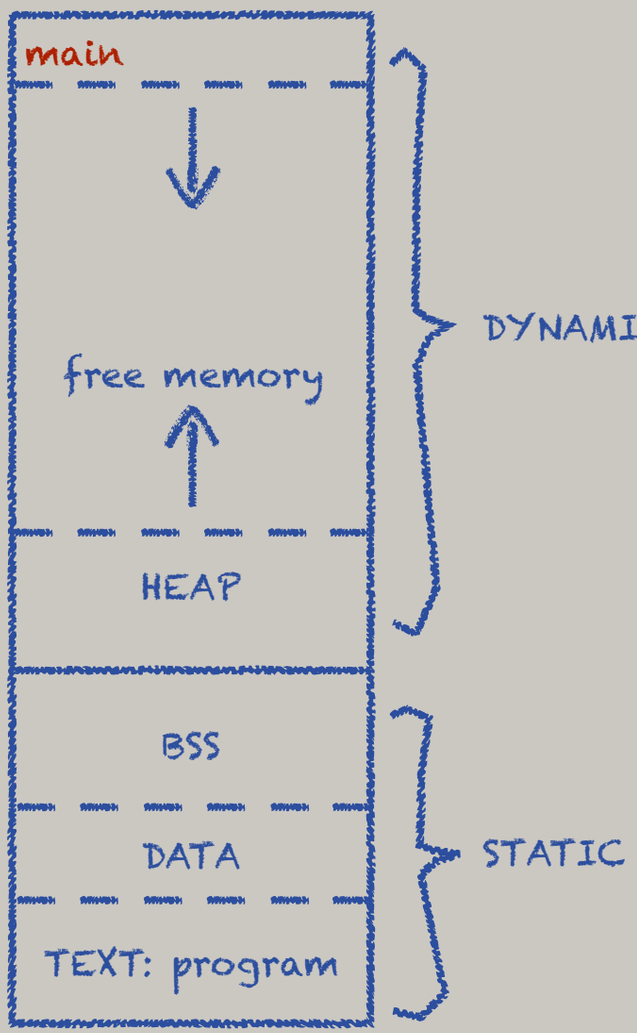
MEMORY ORGANIZATION



The STACK holds invocation records (also called stack frames).

An invocation record is created whenever a function is called. It has space for the function's parameters, local variables, any return value, as well as bookkeeping information related to the call itself (e.g. where to return to).

MEMORY ORGANIZATION



Consider this code:

void g(void) { … }

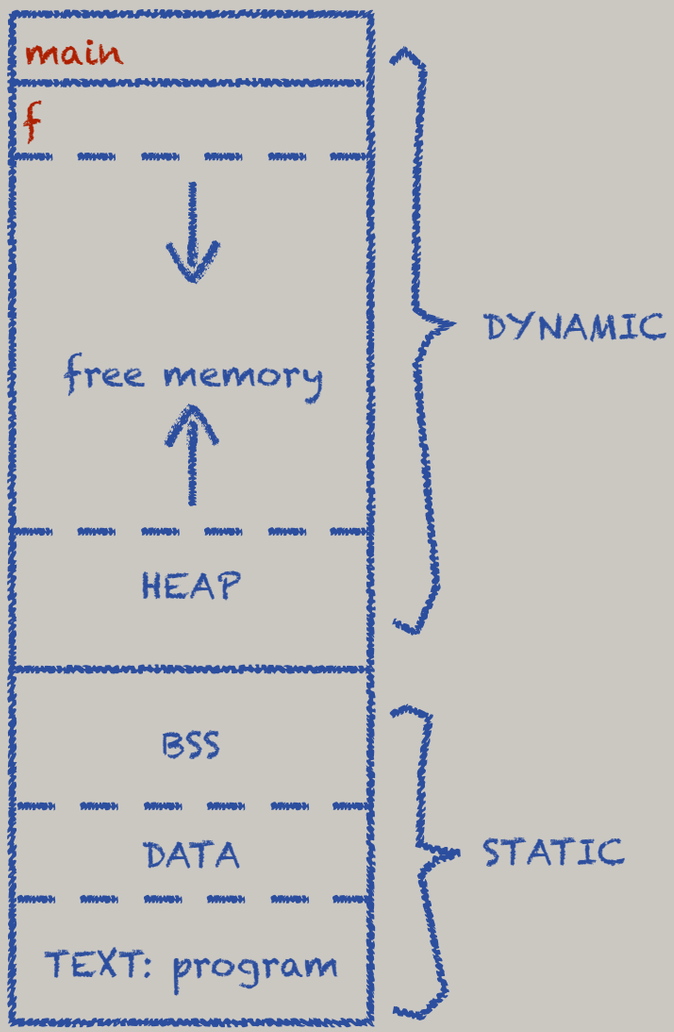
void f(void) { … g(); … }

int main(void) { … f(); … }

The invocation record for main is pushed on the stack as soon as execution begins.

main's record is the current/active one.

MEMORY ORGANIZATION



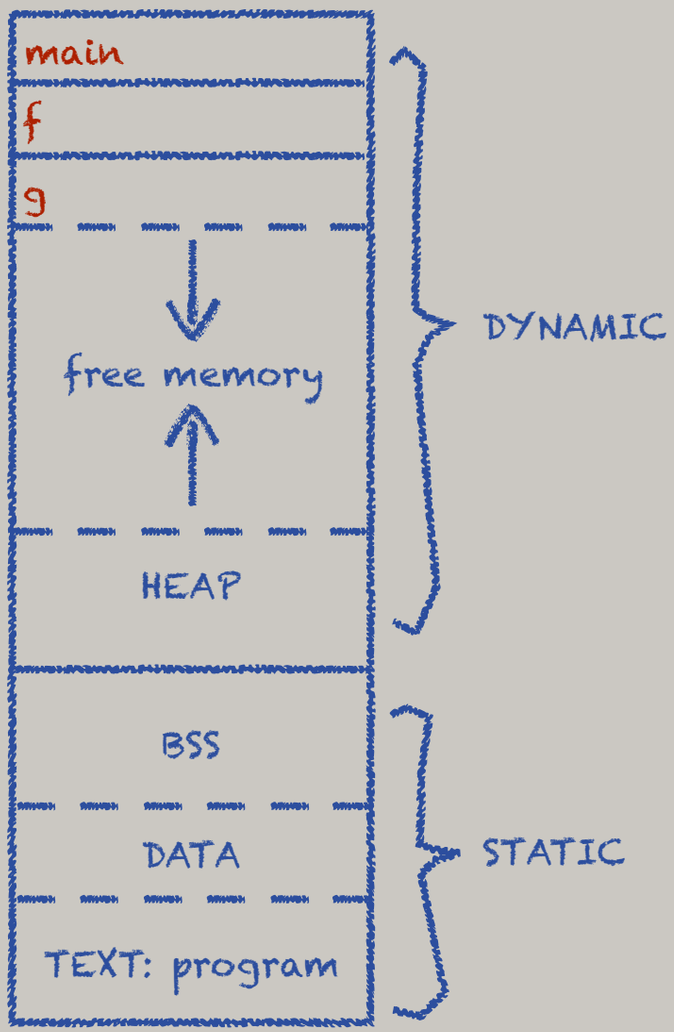
Consider this code:

```
void g(void) { … }

void f(void) { … g(); … }

int main(void) { … f(); … }
```

When f() is called, an invocation record for f is pushed to the top of the stack.

f's record is the current/active one.

MEMORY ORGANIZATION



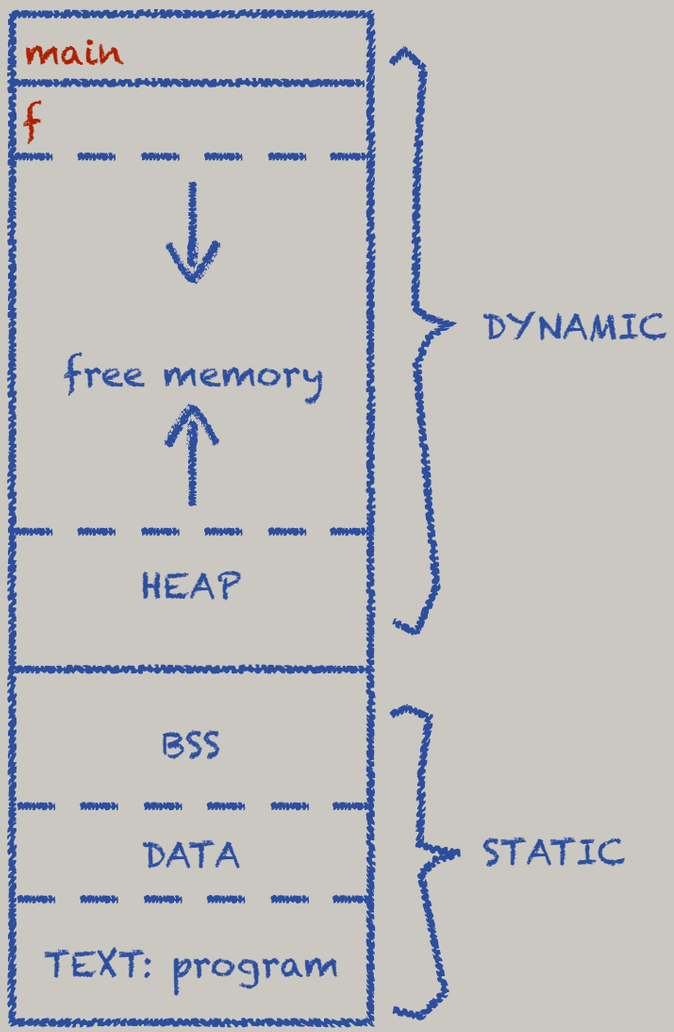
Consider this code:

```
void g(void) { … }

void f(void) { … g(); … }

int main(void) { … f(); … }
```

When g() is called, an invocation record for g is pushed to the top of the stack.

g's record is the current/active one.

MEMORY ORGANIZATION

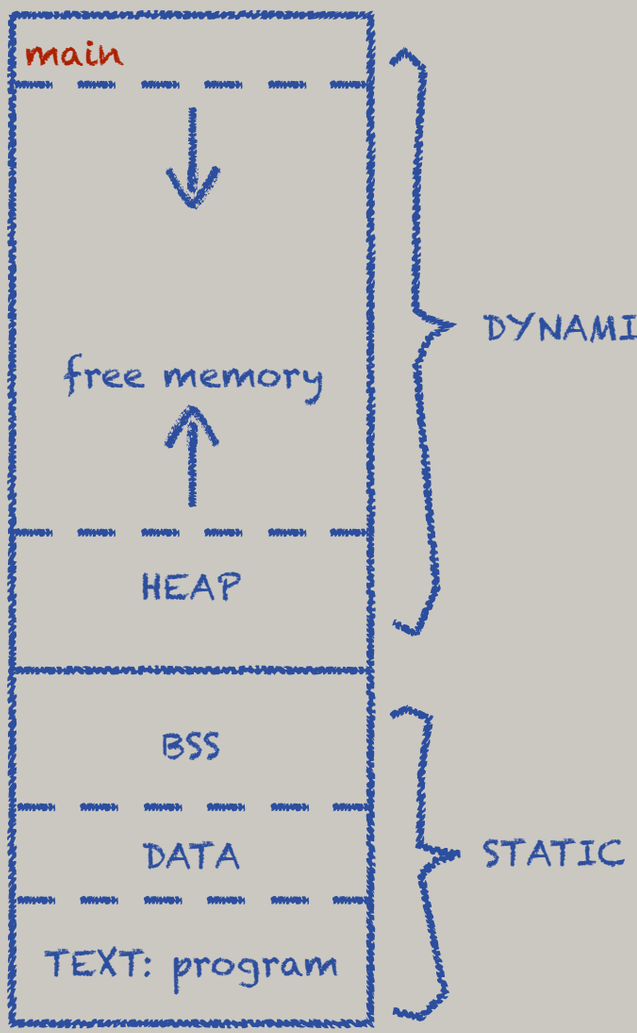


Consider this code:

```
void g(void) { … }

void f(void) { … g(); … }

int main(void) { … f(); … }
```

When g() returns its invocation record is removed from the stack, and f's invocation record becomes the current/active one.

MEMORY ORGANIZATION

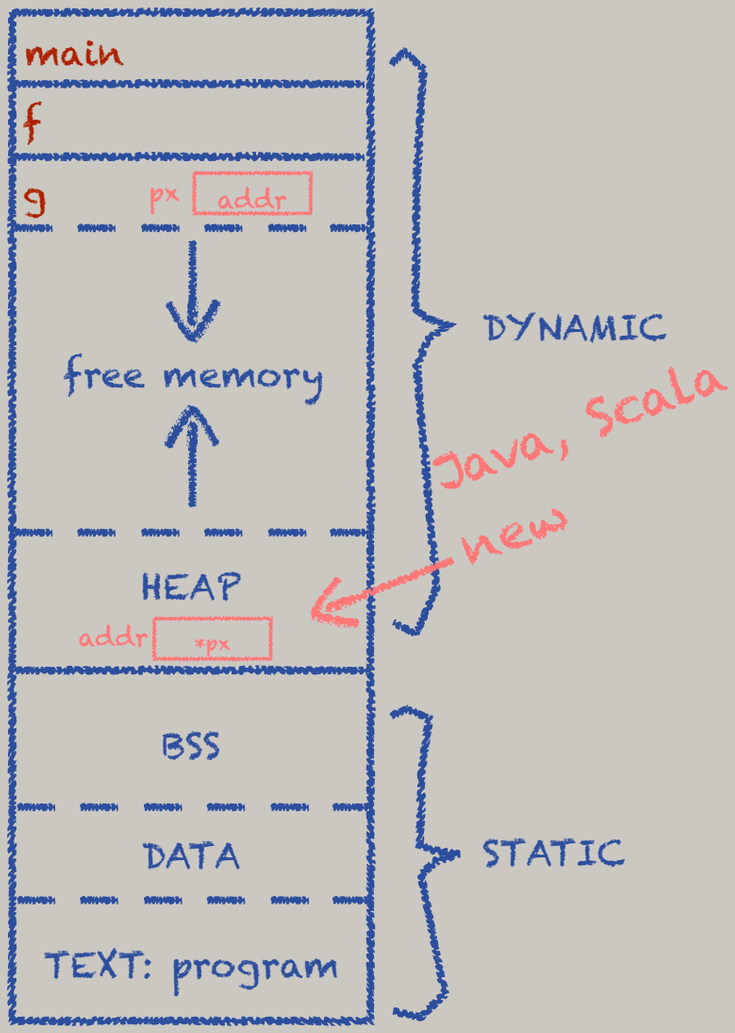


Consider this code:

```
void g(void) { … }

void f(void) { … g(); … }

int main(void) { … f(); … }
```

When f() returns its invocation record is removed from the stack, and main's invocation record becomes the current/active one.

MEMORY ORGANIZATION



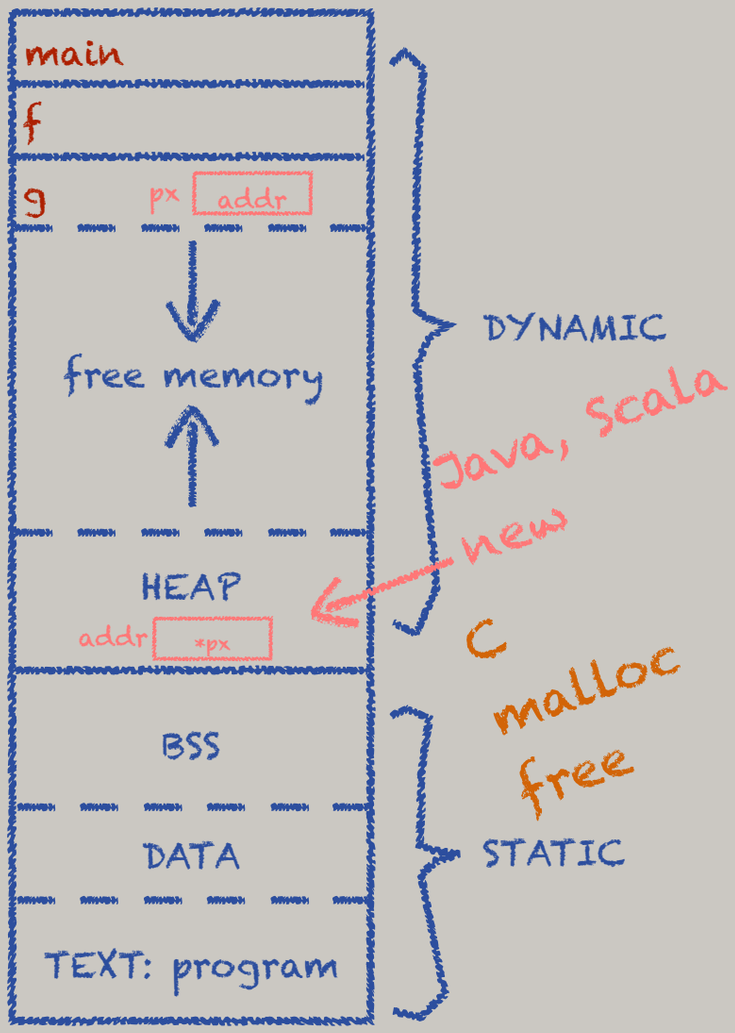
The HEAP is used for dynamic allocation of non-local data.

In Java allocation is done using 'new', as in

```
px = new Foo();
```

Java's garbage collector frees heap-allocated memory when it is no longer in use.

MEMORY ORGANIZATION



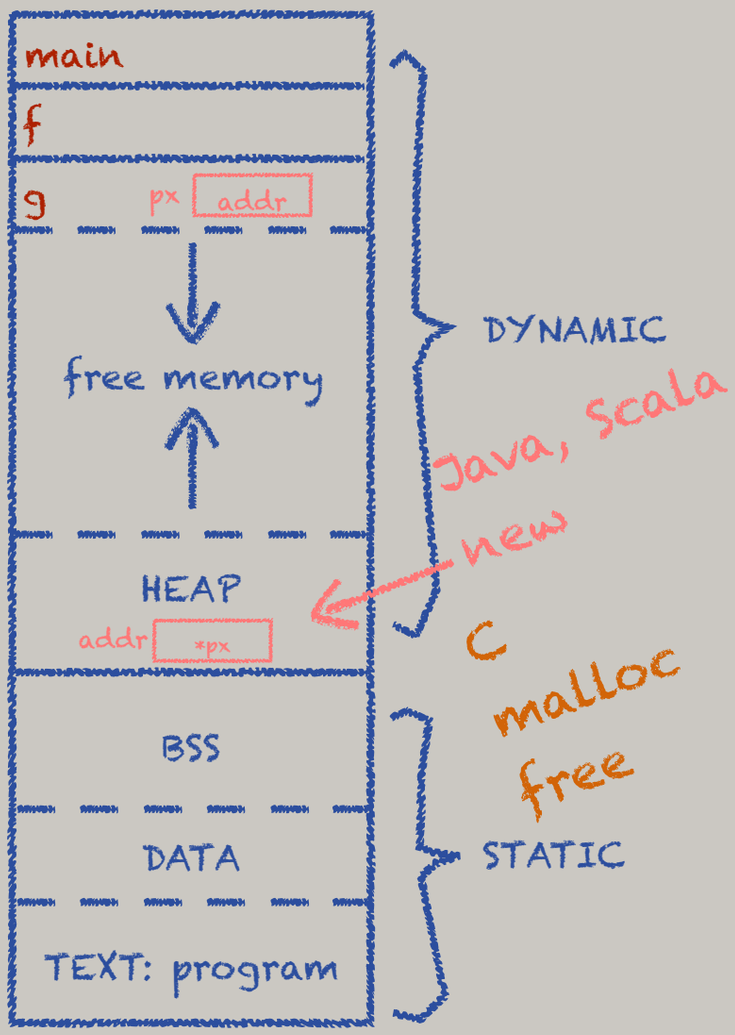
The HEAP is used for dynamic allocation of non-local data.

In Java allocation is done using 'new', as in

```
px = new Foo();
```

Java's garbage collector frees heap-allocated memory when it is no longer in use.

MEMORY ORGANIZATION



In either case the (local) variable px holds the address of the chunk of memory, allocated on the heap, which holds some data.

MEMORY ORGANIZATION

A local variable, like x in the code shown, has memory for its value set aside in the function's invocation record.

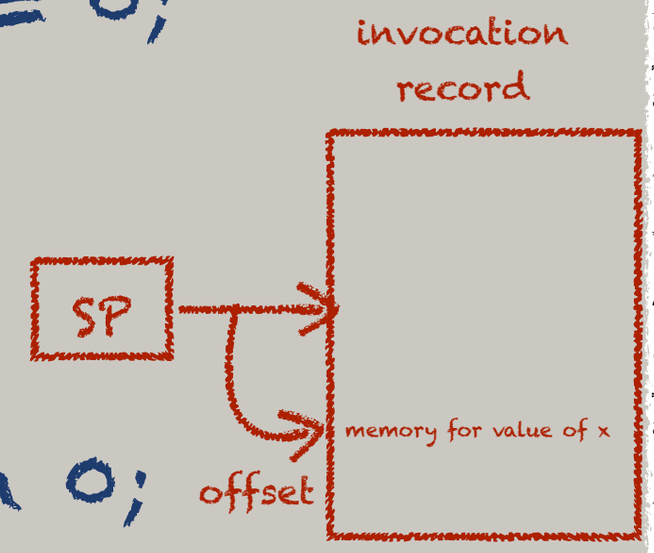
The name of the variable, x in this case, does not exist at runtime.

```
int main() {
    int x = 0;
    .
    .
    .
    return 0;
}
```

MEMORY ORGANIZATION

Any read from x or write to x is translated into a memory access at some offset from the current Stack Pointer (SP). SP refers to a known point within an invocation record.

```
int main() {
    int x = 0;
    .
    .
    .
    return 0;
}
```



COMPILER DOCUMENTATION

https://gcc.gnu.org/onlinedocs/9.4.0/

https://gcc.gnu.org/onlinedocs/gcc-9.4.0/gcc/Standards.html#C-Language

http://releases.llvm.org/10.0.0/tools/clang/docs/UsersManual.html

http://releases.llvm.org/10.0.0/tools/clang/docs/UsersManual.html#c

COMMON OPTIONS

-std	set language standard
-o	set output file name
-g	include debugging information in object file
-c	compile/assemble do not link
-Wall	report "all" warnings
-L	library path
-I	include path

ACTIVITY

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

int main(void) {
      int x = 0;
      while (x < 10) {
            printf("x has value %d\n",x);
            x = x + 1;
      }
      exit(EXIT_SUCCESS);
}
```

Inspect this program and describe as best you can what this program will do when it runs.

Also discuss where in memory space for variable x will be allocated.

ACTIVITY

- Visit the course website: https://cse.buffalo.edu/faculty/alphonce/FA24/CSE306/
- Click on the "In-class activity" button for today.
- Answer the question on the "What does it do?" page.

ACTIVITY

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

int main(void) {
     int x = 0;
     while (x < 10) {
          printf("x has value %d\n",x);
          x = x + 1;
     }
     exit(EXIT_SUCCESS);
}
```

It prints the values 0 through 9 in this format:

```
x has value 0
x has value 1
x has value 2
x has value 3
x has value 4
x has value 5
x has value 6
x has value 7
x has value 8
x has value 9
```

Memory for x will allocated in main's invocation record.

ACTIVITY

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

int main(void) {
     int x = 0;
     while (x < 10) {
          printf("x has value %d\n",x);
          x = x + 1;
     }
     exit(EXIT_SUCCESS);
}
```

It prints the values 0 through 9 in this format:

```
x has value 0
x has value 1
x has value 2
x has value 3
x has value 4
x has value 5
x has value 6
x has value 7
x has value 8
x has value 9
```

In reality the value of x may exist just in a register at runtime, but for our purposes right now we don't need to be concerned about that.

ACTIVITY

- Answer the question on the "Rewrite" page.

ACTIVITY

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

int main(void) {
      int x = 0;
      while (x < 10) {
            printf("x has value %d\n",x);
            x = x + 1;
      }
      exit(EXIT_SUCCESS);
}
```

Transform this program so that the value being printed is stored on the heap rather than on the stack.

First write out your answer on paper WITHOUT compiling the code (compiling at this stage defeats the purpose of the exercise).

This code may have errors in it - that's OK!

- Selected groups - write code on board

- (I will capture what's on boards and insert into slides)