Happy Teachers' Day!

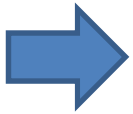
CEng-140

Functions

Function Definition

func-type func-name(*paramter-decl*)

{



var-decl

func-stats

}

Local variables & Parameters

- **Local variables:** can only be accessed by the function where they are declared
  - only accessible in the body of the function, not in any other functions.
- Local variables superseed any identically named variables outside the function
- Paramaters are also treated as if they were declared at the top of the function body

```
int lcm(int m, int n)
{ int i;
...
}
```

Different m ,n, i

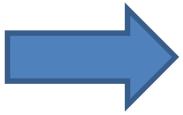
```
int gcd(int m, int n)
{ int i;
...
}
```

Function Definition

func-type func-name(*paramter-decl*)

{

var-decl



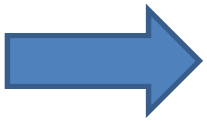
func-stats

}

Function statements & return

- Any valid C stat to be executed when func is called
- The func terminates when it reaches } or when a **return** statement is executed.
  - return (to be used with void type functions)
  - return *expression* (value of the expr is returned to the calling function)

```
int trunc(void)
{
    return 1.5;
}
```



If the type of expr does not match the type of the function, it is **converted** to type of the func!

```
    return (int) 1.5;
```

Function statements & return

- You can have more than 1 return statements in a function!

```
int factorial(int n)
  {
    int i, result;

    if (n < 0) return -1;

    if (n == 0) return 1;

    for (i = 1, result = 1; i <= n; i++) result *= i;

    return result;
  }
```

Function Declaration

- If a func is used (called) before it is defined, it must be declared with a **prototype**:

  func-type func-name(*paramter-decl*) **;**

- The prototype must agree with the func definition
  - but parameter names can be different, or even omitted!

Function Call

- Function call is an expression of the form

  func-name(*argument-list*)

  – when followed a semi-colon, it becomes an expression statement, like:

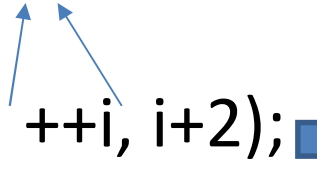
  printf("hello \n");

- Argument-list is a comma separated list of expressions

square(i*3);

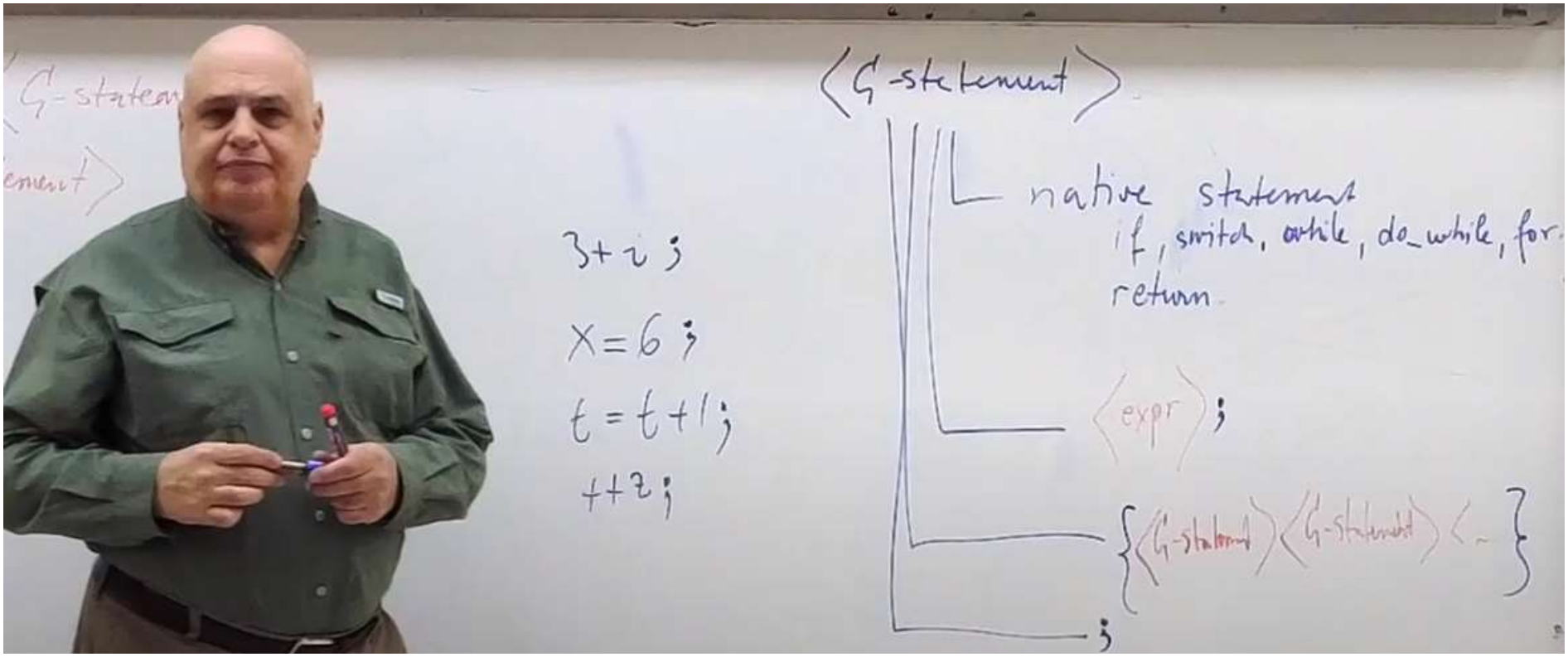
myMathFunc(i*3, ++i, i+2);

Comma operator???



Exp eval order is **unspecified** also for arguments of a function, so **AVOID** exp with side-effects on the same variable!

Statements in C



Function Call

- A func call can occur in any place where an expr can occur:

```
if (i == square(j))
```

- Function calls can be embedded:

```
j = square(square(i));
printf("%d ", square(i));
```

- () must be there even if the arg list is empty!

```
init();
```

Function Call

func-name(*argument-list*)

- When a function is called, its **parameters are initialized** to the **value of the args** in the func call
- Thus, if the **args are** not correct (declared) type, they are **converted** (as-if **type-cast by assignment**) to the declared type of params
  - if no. of args is different than the declared no. of params → **ERROR!!!**
  - if arg and params are not the same (or, convertible) types, → **ERROR!!!**

Function Call

When a function is called, its **parameters are initialized** to the **value of the args** in the func call

```
int lcm(int m, int n)
{ int i=0;
…
}

int main(void)
{…
  lcm(20, 30); }
```

int m=20;

int n=30;

type-cast by assignment

Function Call

- Upon function call:
- the program control passes from the calling funtion to the called function
  - calling function is suspended
- execution continues in the called function
- when the called function terminates,
  - the control passes back to calling function (to the point right after the func call)
  - the value returned by the func is **substituted for the func call** in the calling function

Parameter Passing

func-name(*argument-list*)

- C provides **pass-by-value** parameter passing
  - Called function is provided with the **current values** of the arguments
  - Changes in the value of the parameter does not change the value of the argument!

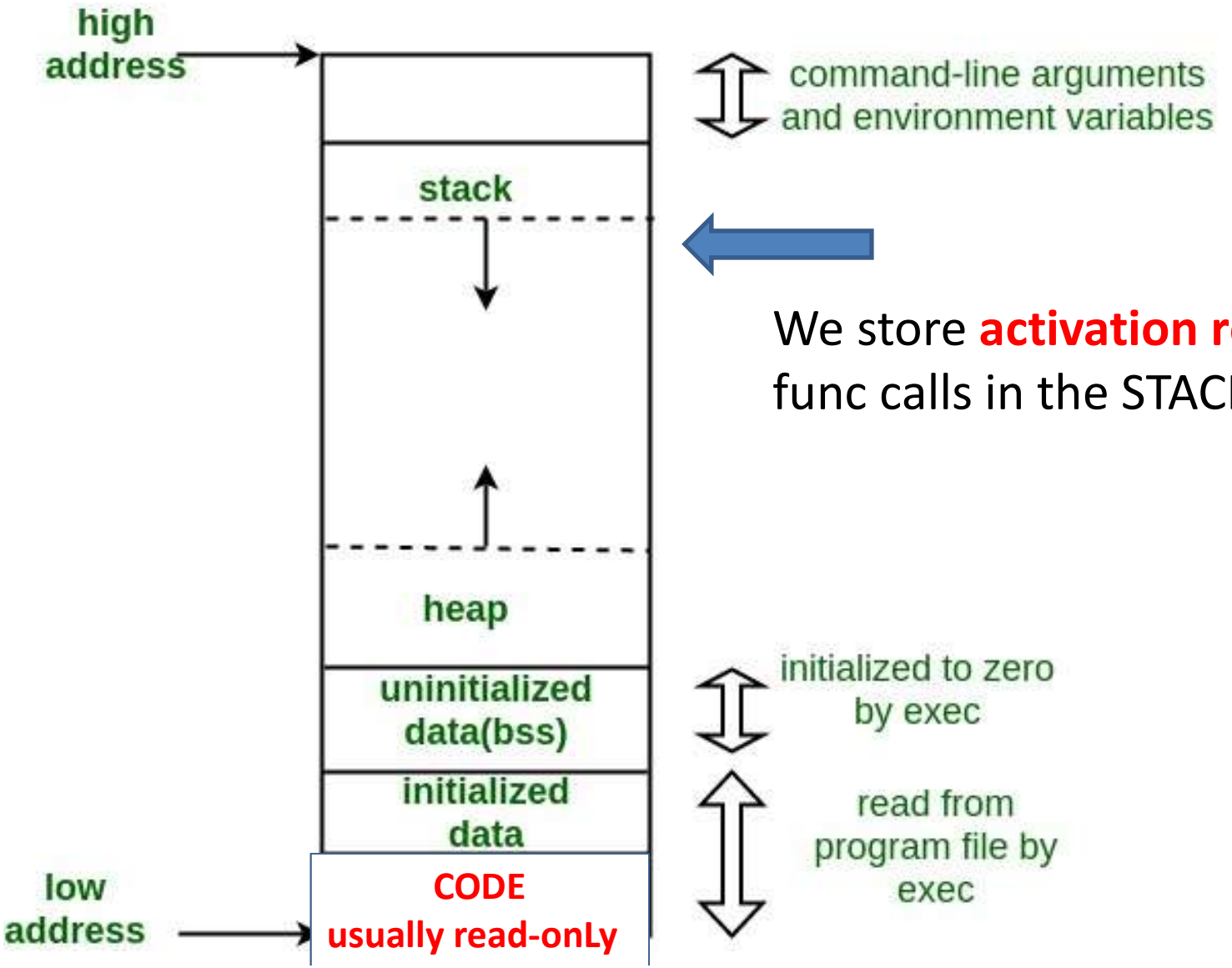
Parameter Passing

```
void exchange (int i, int j);
int main(void)
{ int i= 1,  j= 2;
  printf("%d %d", i, j);
  exchange(i, j);
  printf("%d %d", i, j);
}
```

```
void exchange (int i, int j)
{ int t;
   t =i, i=j, j=t;
   printf("%d %d", i, j);
}
```

To get the value of the **args** changed:
a) use return, b) use global vars, c) use pointers!

Memory Layout of C Programs



We store **activation records** for the func calls in the STACK

Activation Record

- Stores the **state** with one **instance (i.e., call)** of a function
  - its form can be known during compile time
  - AR is allocated in the stack (in run-time) and remains there as long as the function executes

Upon function call:

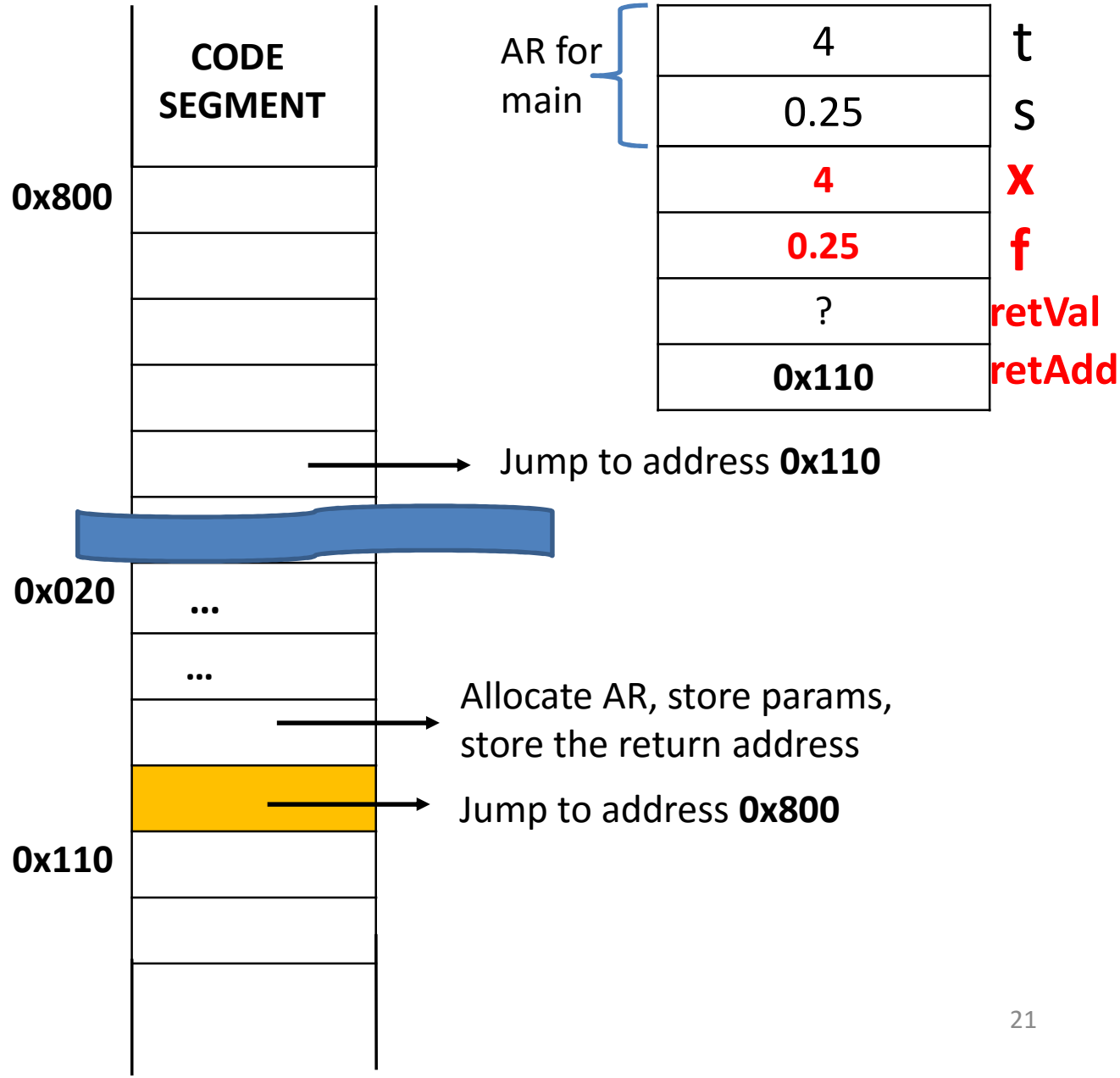
- Caller func allocates the AR (for the func to be called) into stack
  - stores values of parameters to AR
  - stores the return address (the address of code to come back when the called func terminates) to AR
  - caller suspends (i.e, execution jumps to the code for the called funtion)

Upon function call:

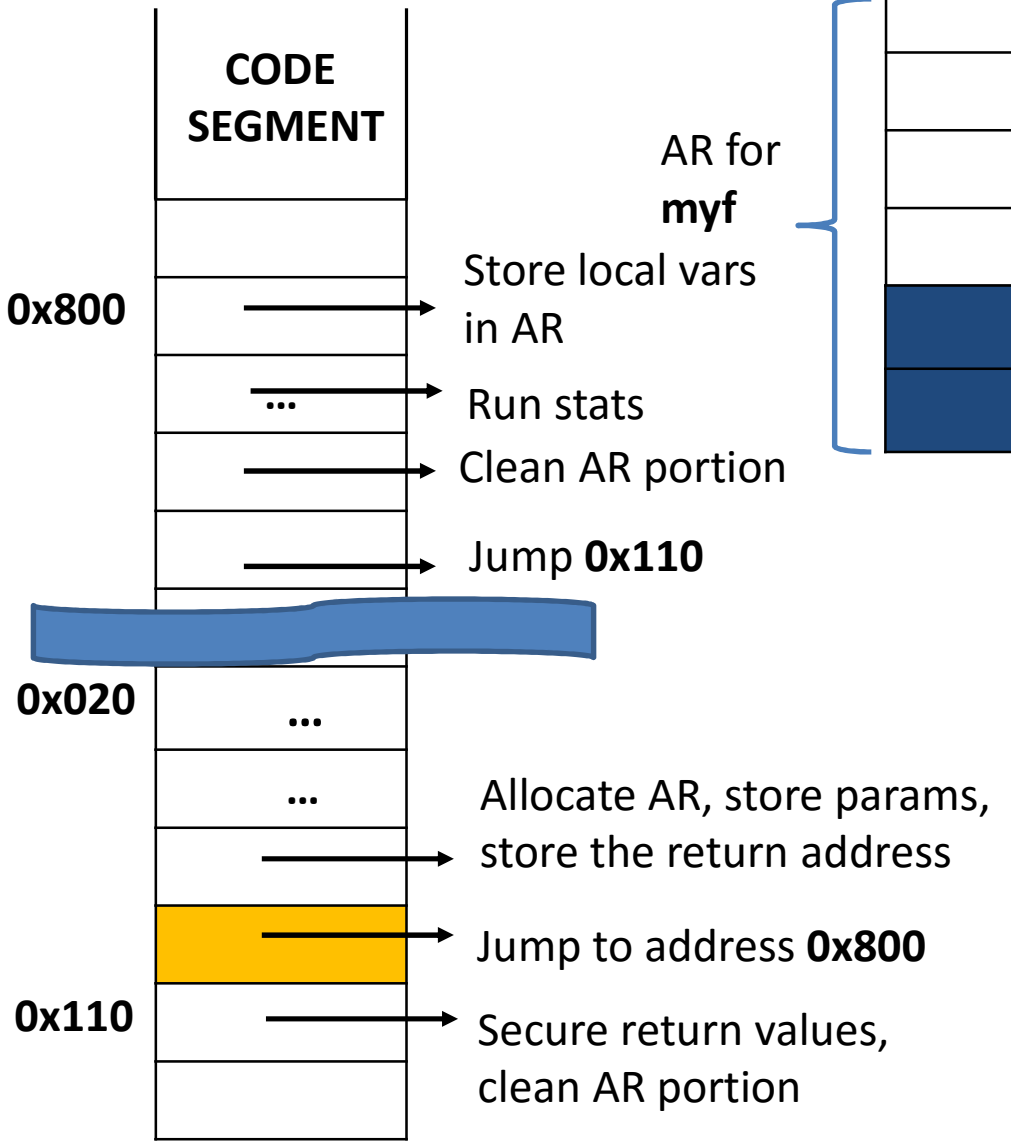
- Called func:
  - Stores its local variables to AR
  - Executes its code
  - If there is a return value, stores to AR
- When called func terminates
  - Cleans its portion of AR (local vars)
  - Control goes back to caller (jump to return address)
- Caller takes control
  - Secures the return value
  - Cleans its portion of AR (params, return address, value)

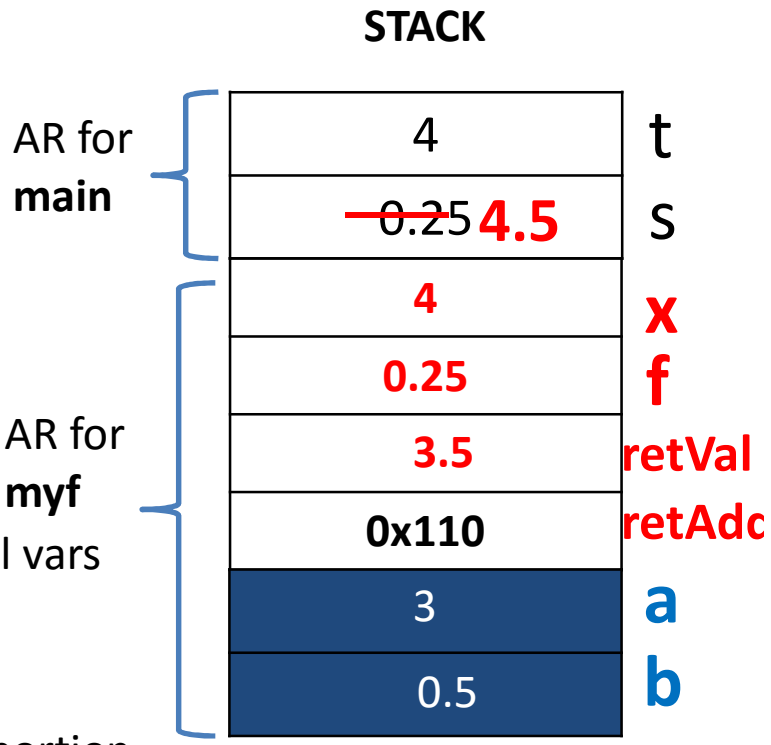
```
float myf(float f, int x)
{ int a; float b;
   a  = x-1;
   b = f *2;
   return a+b; }
```

```
int main()
{ int t = 4;
   float s = 0.25;
   ....
   s = myf(s, t)+1; }
```



```
float myf(float f, int x)
{ int a; float b;
   a  = x-1;
   b = f *2;
   return a+b; }

int main()
{ int t = 4;
   float s = 0.25;
   ....
   s = myf(s, t)+1;
 }
```

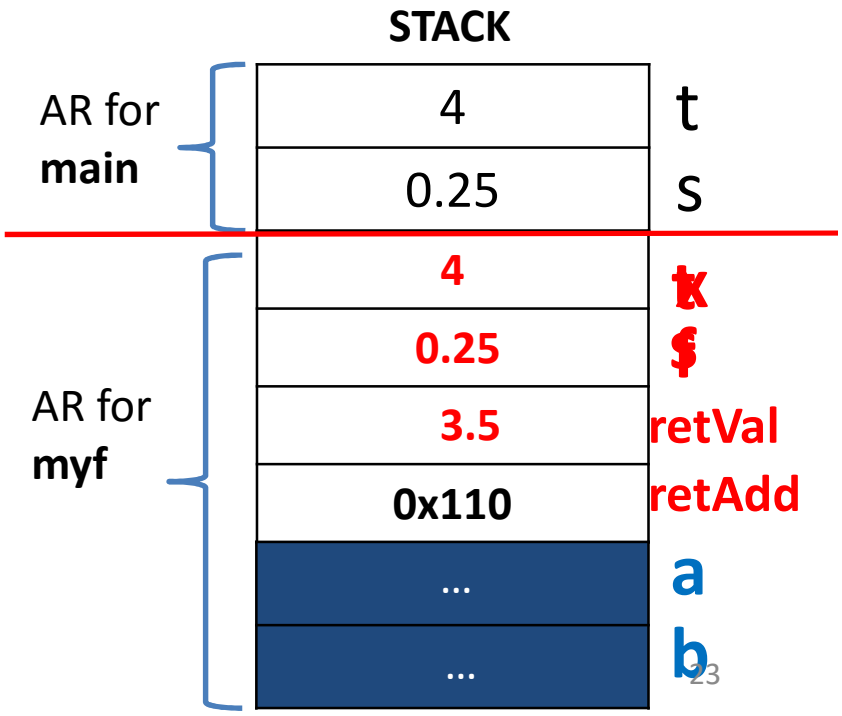




Remember previous claims:

- Parameters are like locals
  - initialized by arg values
- C provides pass-by-value parameter passing
  - Called function provided with the **current values** of the arguments
  - Changes in the value of the parameter does not change the value of the argument!

```
float myf(float f, int x)
{ int a, b;
  ... }
/* could name f, x as s, t*/
s = myf(s, t)+1;
```

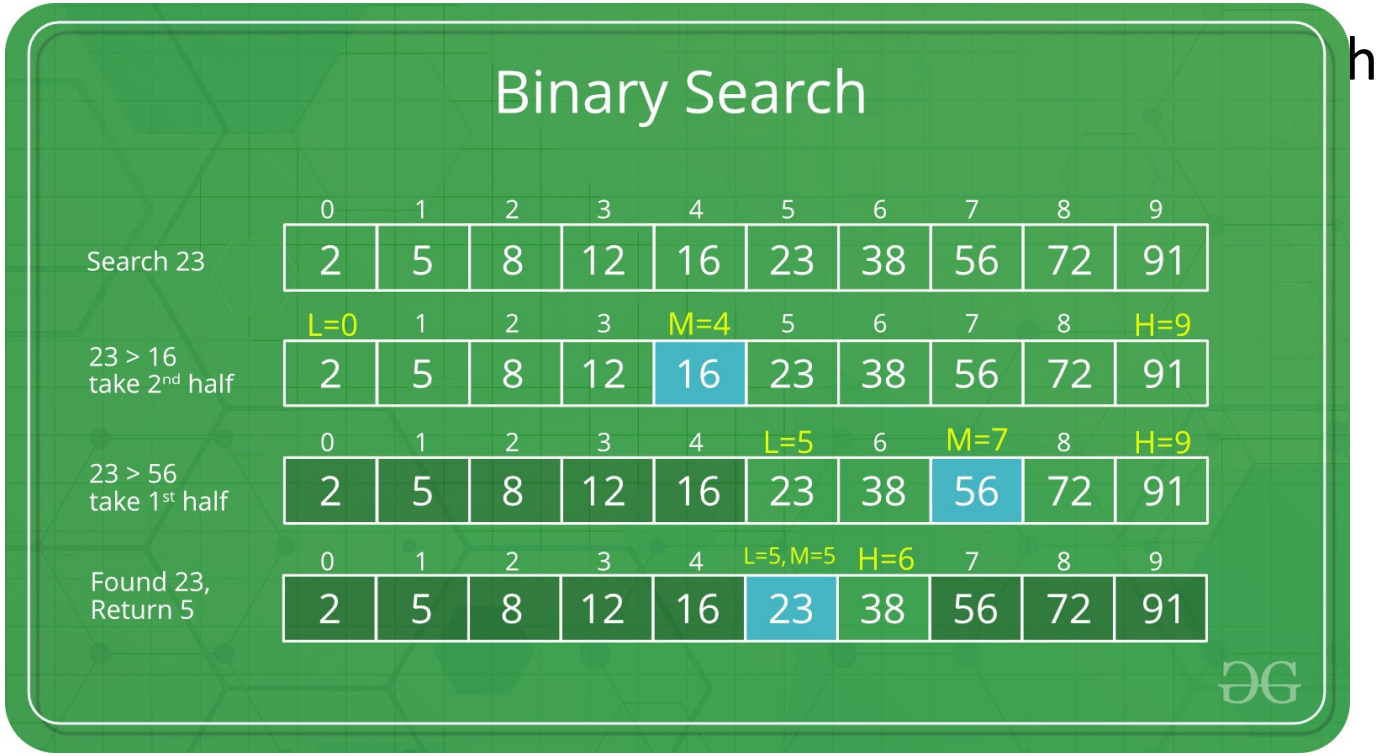


Recursion

- Recursion is an extremely powerful problem-solving technique
  - Breaks a problem in smaller identical problems
  - An alternative to iteration, which involves loops
- A sequential search is iterative
  - Starts at the beginning of the collection
  - Looks at every item in the collection in order
- A binary search is recursive
  - Repeatedly halves the collection and determines which half could contain the item
  - Uses a divide and conquer strategy

Recursion

- A sequential search is iterative
  - Starts at the beginning of the collection
  - Looks at every item in the collection in order
- A binary search is recursive



half

Recursive Solutions

- Facts about a recursive solution
  - A recursive method calls itself
  - Each recursive call solves an identical, but smaller problem
  - A test for the base case enables the recursive calls to stop
    - Base case: a known case in a recursive definition
  - Eventually, one of the smaller problems must be the base case

Recursive Solutions

- Four questions for construction of recursive solutions
  - How can you define the problem in terms of a smaller problem of the same type?
  - How does each recursive call diminish the size of the problem?
  - What instance of the problem can serve as the base case?
  - As the problem size diminishes, will you reach this base case?

Recursive Functions

- A function may **directly** or **indirectly** call itself.

```
int factorial(int n)
{ if (n == 0)
     return 1;
  else
     return n*factorial(n -1); }
```

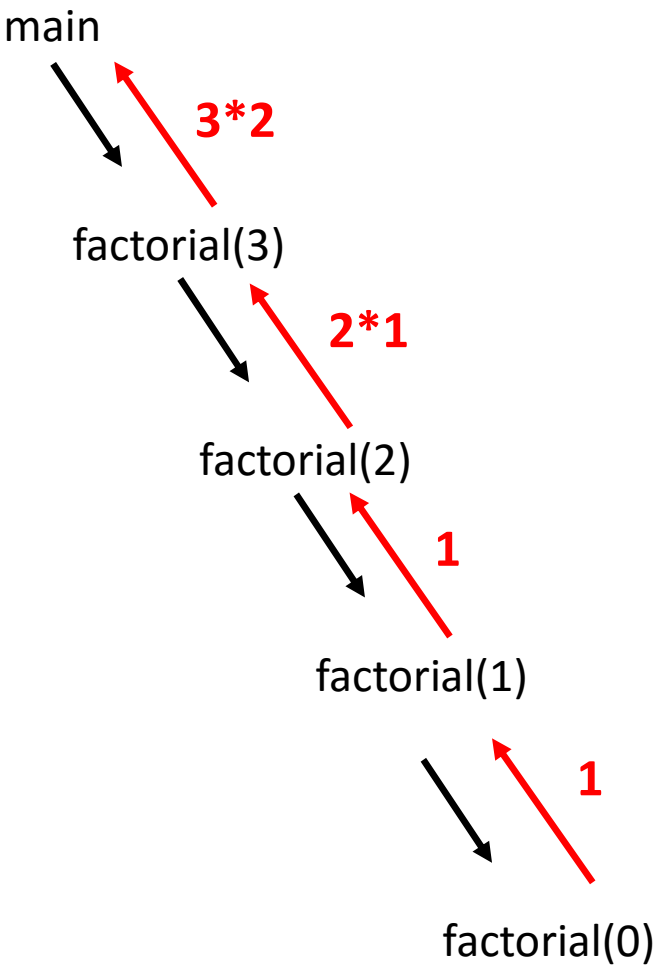
```
int main(void)
{ int result;
  result = factorial(3);
}
```

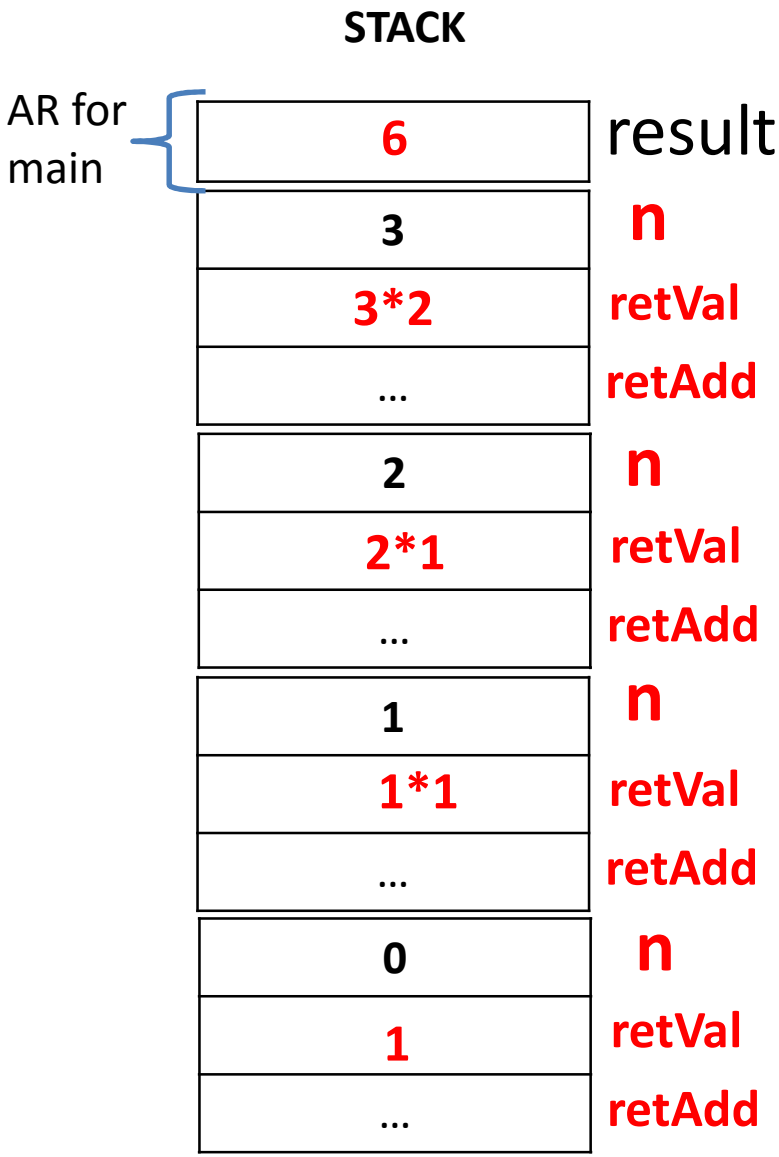
What is 16!, roughly?

What if I tell you 15! İs 1,307,674,368,000

```
int main(void)
{ int result;
  result = factorial(3);
}
```

```
int factorial(int n)
{ if (n == 0)
    return 1;
  else
    return n*factorial(n -1); }
```





Looks expensive!

Blocks

- In Ansi-C you cannot nest functions
  - But, blocks can be nested
- A block is a sequence of variable declarations and statements enclosed within **braces**
  - You can declare and initialize variables at the beginning of any block

Blocks

```
int factorial(int n)
{ if (n<0) return -1;
  else if (n == 0) return 1;
  else
  {  int i, result = 1;
     for (i=1; i <= n; i++)
          result *= i;
     return result;
  }
}
```

Variables are declared near their usage (good for readability)

Like function's local vars, These vars local to block may not get memory (i.e., allocated) if the block is not entered

Scope

- Part of the program within which a name can be used is called its **scope**.
  - The range of statements where the variable is visible
- Scope of a local variable: the function in which it is defined
  - A local var defn supersedes (or, hides) that of an external (global) variable with the same name.
- Scope of a variable in a block: the block in which it is defined
  - A block var defn supersedes (or, hides) that of an outer block variable with the same name.

Static Scoping

```
f(   )
{ int a, b, c;
   ...
   {  int   d;
      ...
   }
   ....
   {  int    e;
   }
}
```

Where can I use the variable d (or, e)?

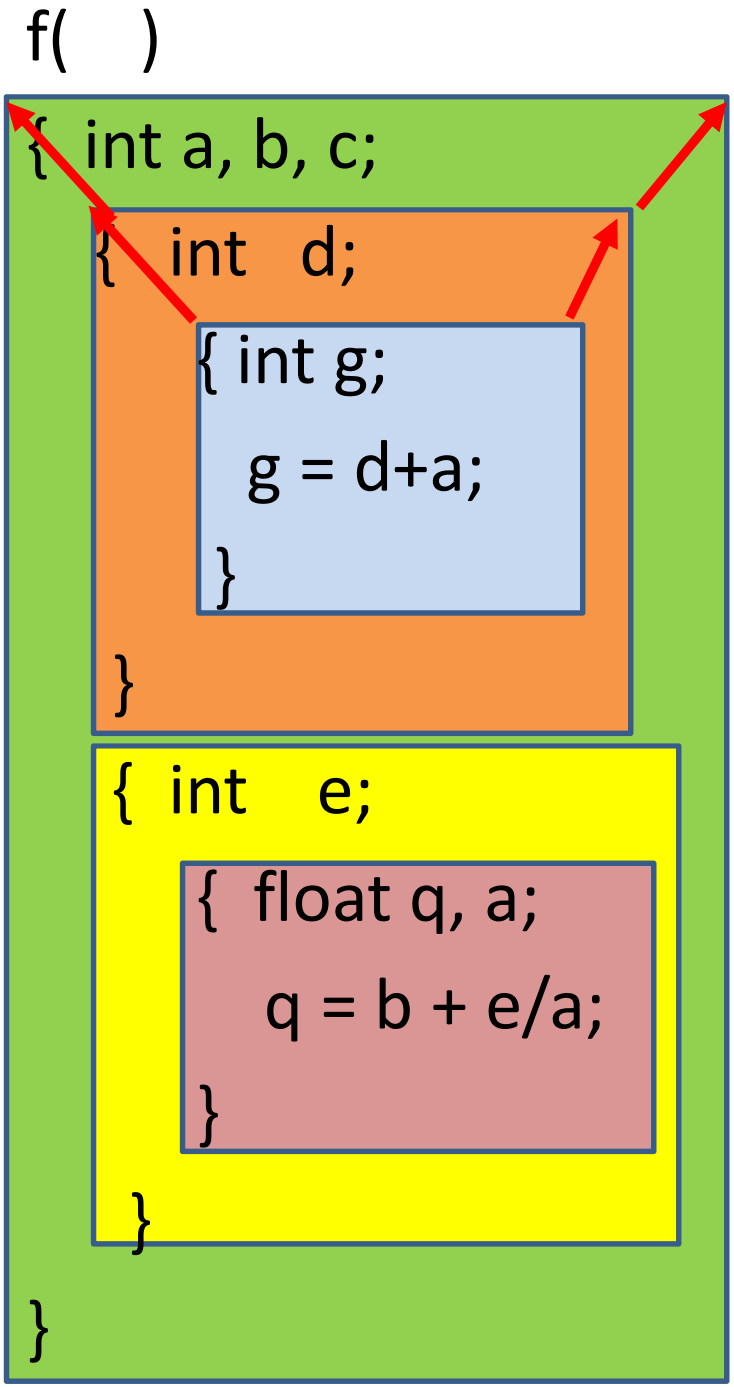
Where can I use the variables a, b, c?

What happens if I declare int b in first block?

What happens if I declare float a in first block?

What happens if first block has z = a + d;

Static scoping: determined at compile time,
from the smallest enclosing unit to the largest!



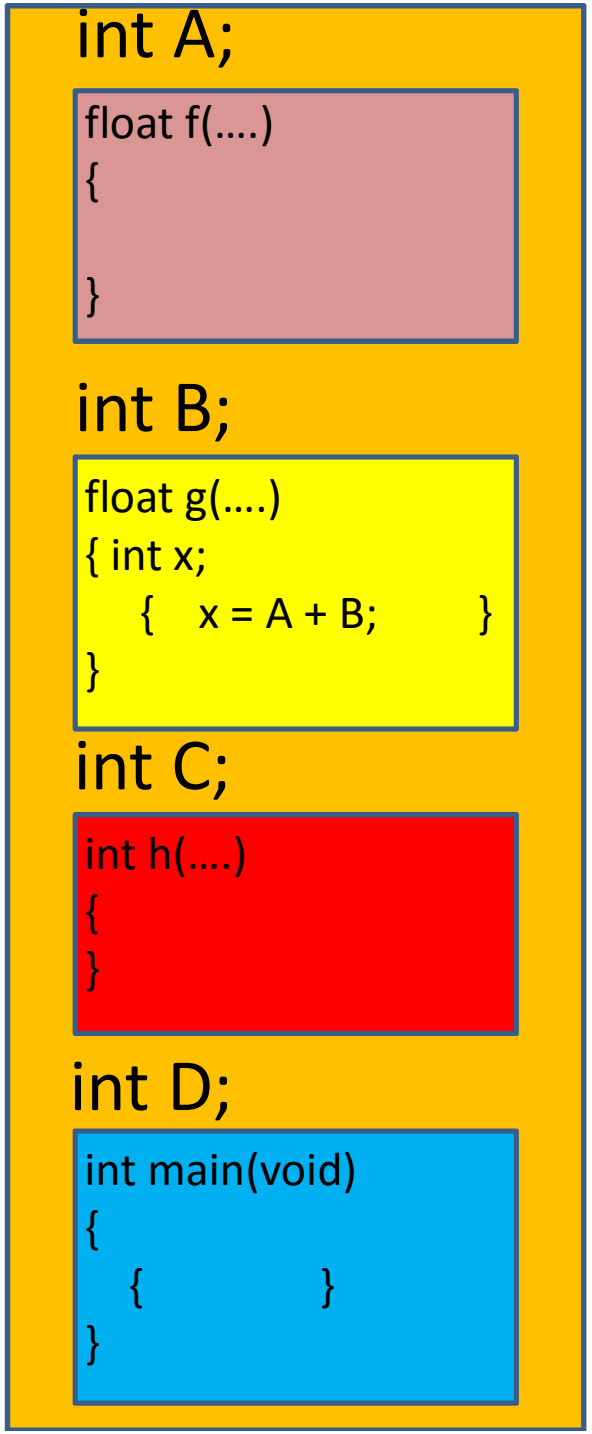
Static scoping: determined at compile time,
from the smallest enclosing unit to the largest!

What happens if the statement is:
q= d +e/a;

What happens if the statement is:
g= b +e/a;

Global (external) variables

- Variables thay are available accross function boundaries
- A variable defined outside of any function (i.e. at the same level as function definitions) is called a global variable
- Scope of a global variable: The rest of the source file starting from its definition.
- Global vars are good for returning more than one values from a function
  - Yet, use carefully! (may not be best way to return values)



Scope of a global variable: The rest of the source file starting from its definition.

```c
#include<stdio.h>
int a = 5;

void f(int a)
{ printf("a in f() = %d\n", a); }

void g()
{ int a = 30; printf("a in g() = %d\n", a); }

void h()
{ printf("a in h() = %d\n", a); }

int main()
{
int a = 10;

        { int a = 20; printf("a in block structure = %d\n", a); }

        printf("a in main() = %d\n", a);

        f(a);
        g();
        h();

return 0;
}
```

Storage Classes of Variables

- Storage class determines the lifetime of the storage associated with the variables
  - A var is in one of the two storage classes:

    automatic or **static**

Automatic Variables

- A var is automatic if it is **allocated storage** upon entry to a code segment & **deallocated** upon exit

  auto type *var_name*

- Variables can be declared auto **only** within a block. If the storage class is **not** specified explicitly, a var decl. within a block is auto.

Automatic Variables

- An auto var may be explicitly initialized with an expression. The expression is evaluated (and its value is assigned to the auto variable **each time** the block is entered).

```
int foo(int n)
{  int i, j = 10, result = n-1, k = j+1;
   …}
```

What is the value of var i if I print it?

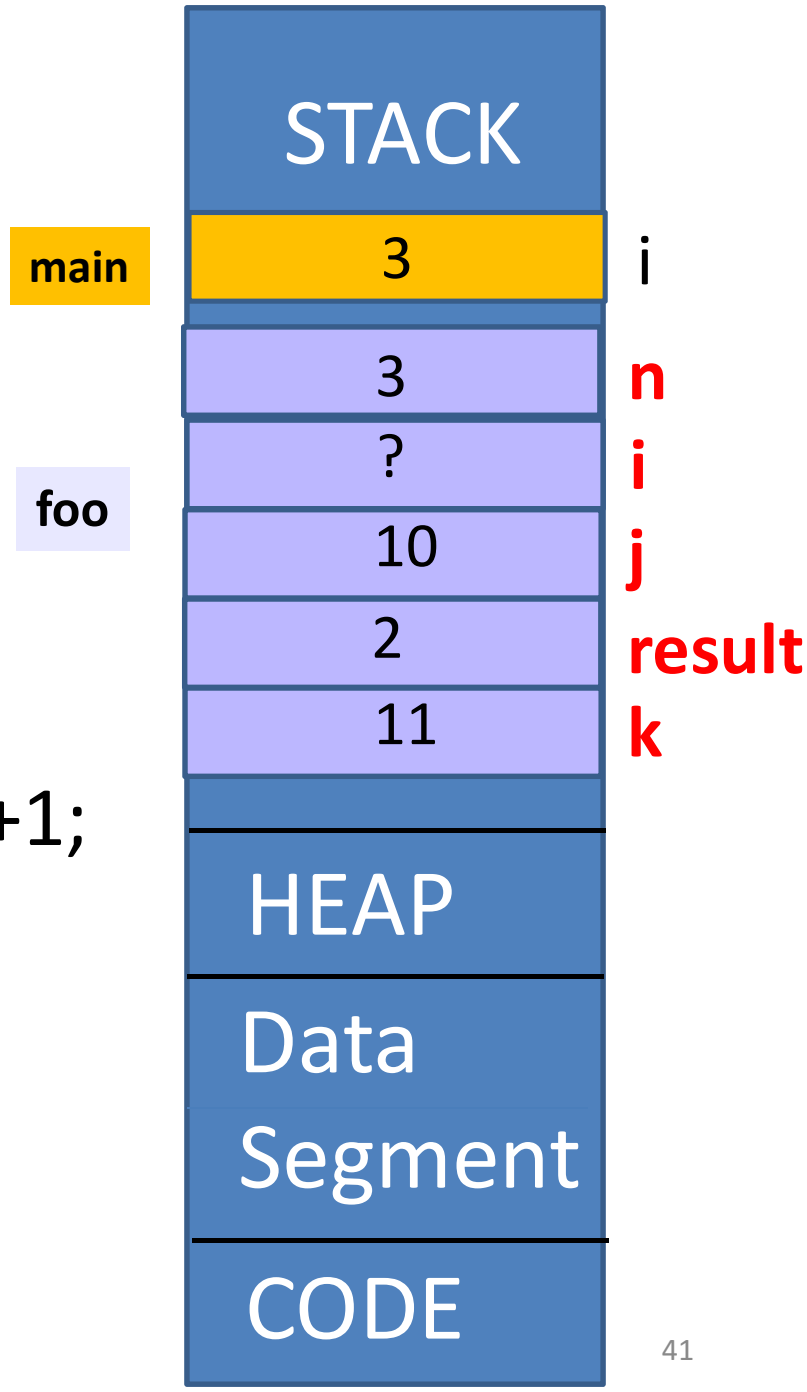
Are these initializations allowed?

Do I need to say **auto**?

No: Since in a block, all are by definiton auto variables
Equivalent to
auto int i, j…;
(notice all will be type auto int)

Example

```
int foo (int n)
{  int i, j = 10, result = n-1, k = j+1;
    …}
int main(void)
{ int i=3;
  foo(i); }
```



Static Variables

- A var is static if it is **allocated storage** at the beginning of the program execution & storage remains unallocated until execution terminates

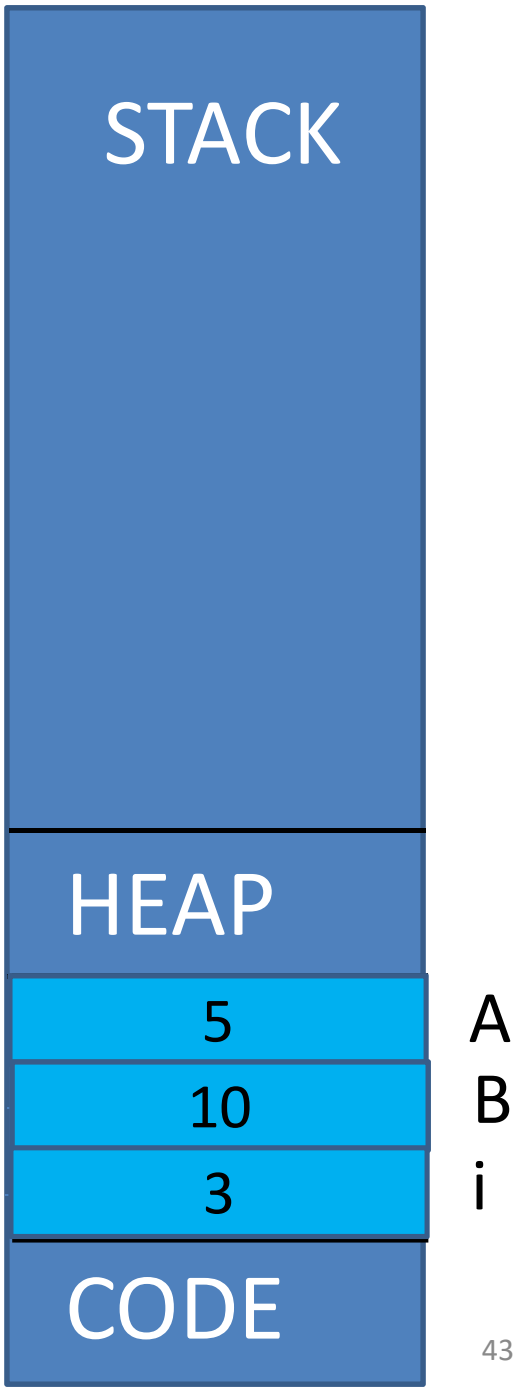
  static type *var_name*

- Vars declared outside of all blocks (i.e., at the same level as func definitions) are ALWAYS static
  - Don't use static with globals, it means smt else (later)
- Within a block, a var can be specified to be static
  → REMARK: It is still local, but storage & lifetime properties are different!

```
int  f()
{ static int x; }
```

Example

```
int A = 5, B = 10;

int main(void)
{ static int i=3;
}
```



Static Variables

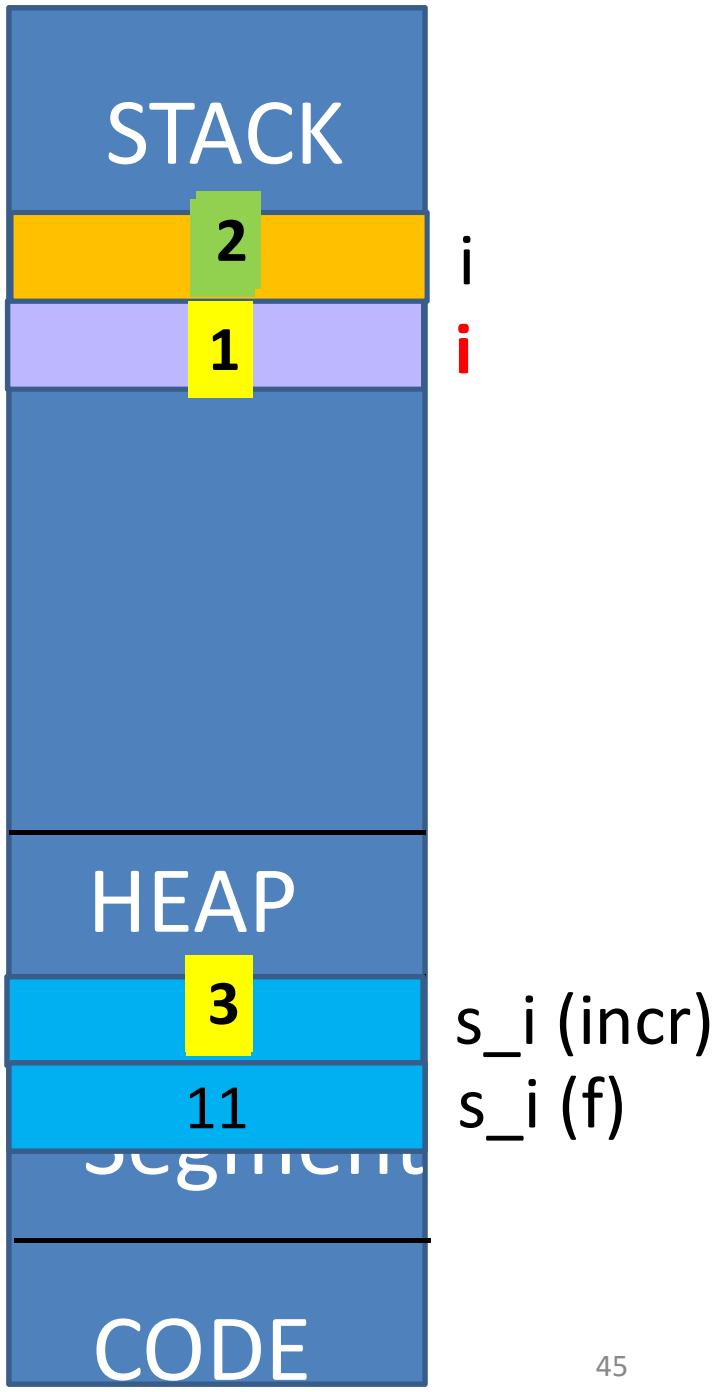
- A static variable is initialized only with a **constant expr.** and **only once** [when the block is entered for the first time…[ANSI C book]]
  - If not initialized, default value of a static variable is **0**.
  - Thus, the values assigned to static vars (in a function) are **retained** accross function calls

Examples: (on the board)

Example

```
void incr()
{ int i = 0;
   static int s_i = 0;
   printf("%d %d", i++, s_i++); }
void f()
{static int s_i  = 11;
}
 int main(void)
{ int i;
  for (i=0; i<3; i++)
      incr(); }
```

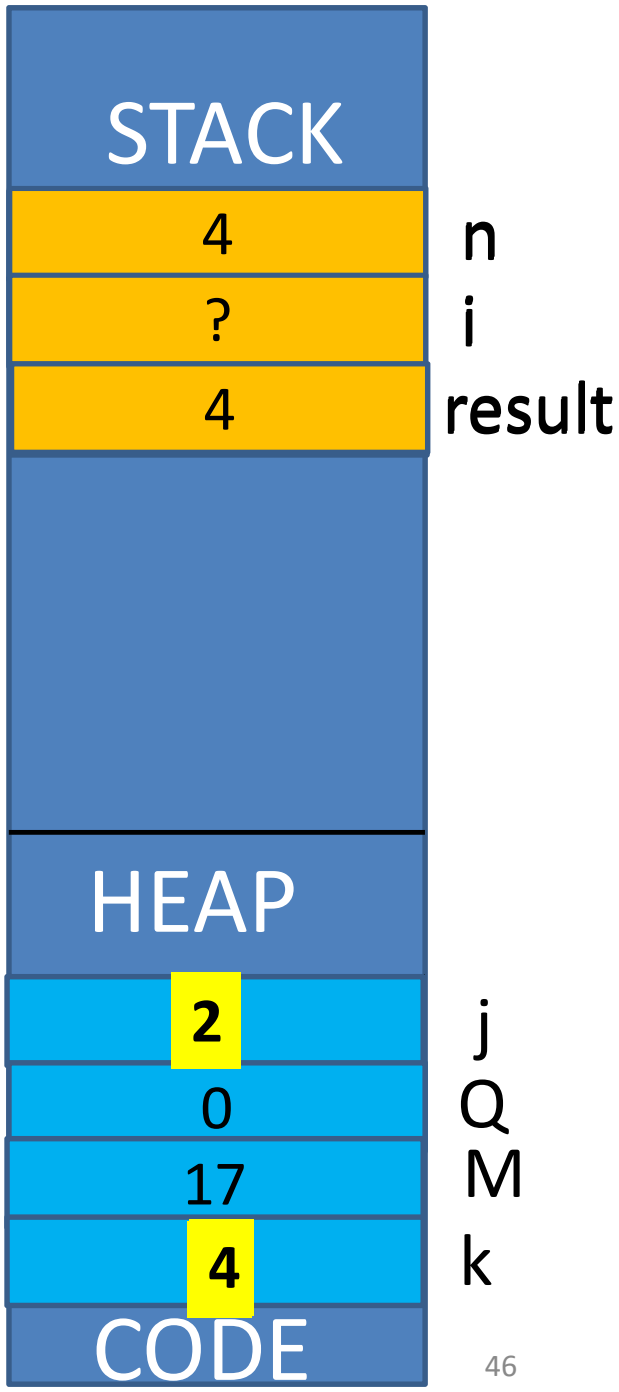
0 0
0 1
0 2



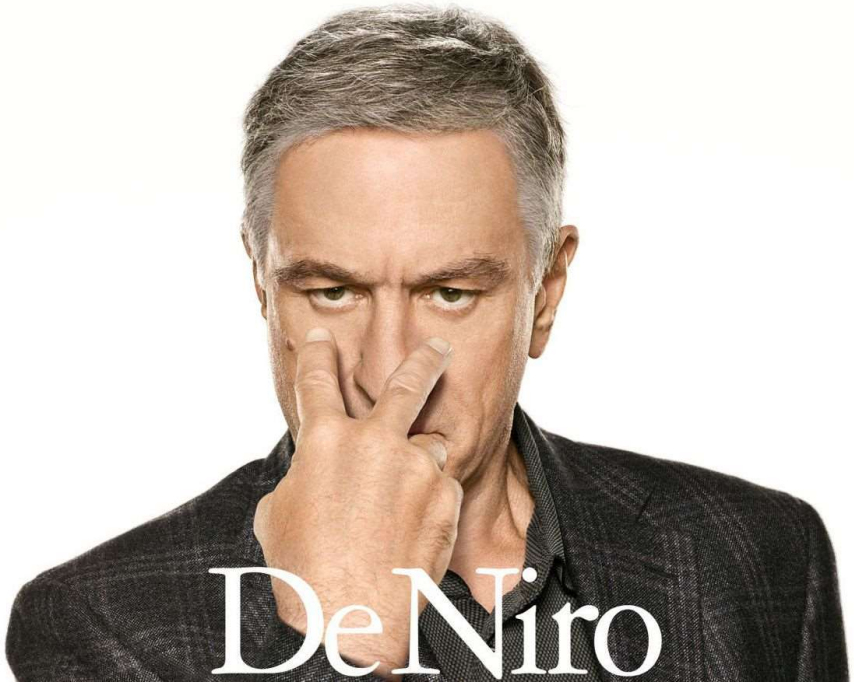
```
int M=17;

int f(int n)
{ int i, result = n;
  static int j, k = 2;
  j++; k++;
  printf(..j, k); }

int Q;
int main()
{ f(7);
  f(4);}
```

What if we printf M, Q in main?

What if we printf j, k in main?



Quiz (Really!!!)



Choose one:

a) **LOCAL**
b) **GLOBAL**
c) **STATIC**
d) **AUTO**

An auto variable is always **LOCAL**

A local variable is not always **AUTO**

A global variable is always **STATIC**

A static variable is not always **GLOBAL**

Extern

- If an **external (global) var** is needed in **another file**
  - (or, in the same file but at a point earlier than that at which it is defined)
- it must be declared with the keyword **extern** before it can be used.

  extern type *identifier*

Decl vs definition of an extern var

- The **declaration** of an **external variable**, specified using keyword extern, declares for the rest of the source file the type of the variable,
  - But, does not allocate any storage!
- The **definition** of an external variable, **w/o** keyword extern, causes the storage allocated (and also serves as the declaration for the rest of source that source file).

Initialization

- An **external variable** can be initalized only at the time of its definition (We know how)
- There must be **only one definition** of an external var, all other files that need to access this variable must contain an **extern declaration**

test.c

```
void mod(void);
extern int i;
int main(void)
{  scanf("%d", &i);
   mod();
}
int i;
extern int j;
void output(void)
{  printf(".. ", i, j);
}
```

comp.c

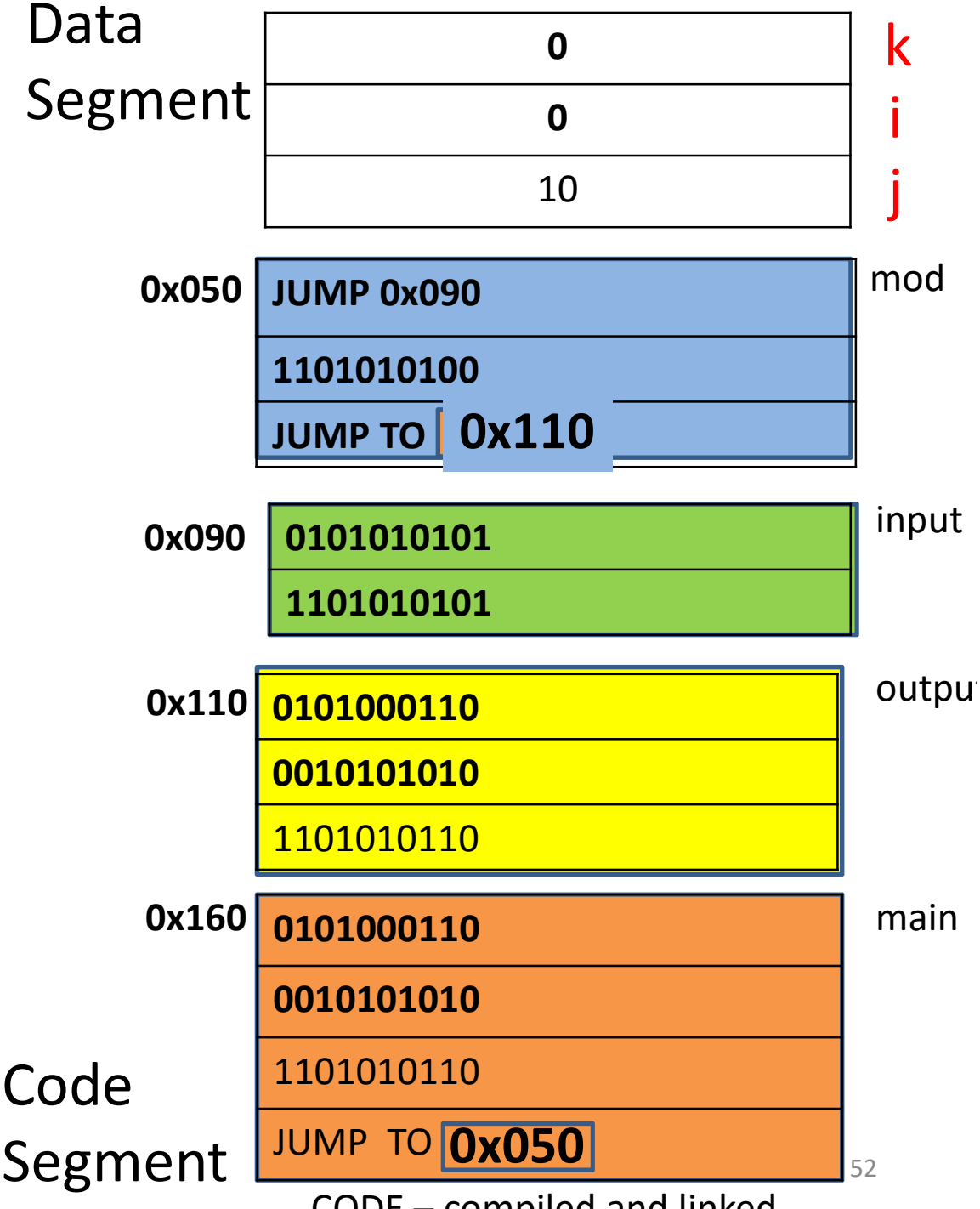
```
extern int i;
void output(void);
int j = 10;
int k;
void input(void)
{ scanf("%d", &k); }
void mod()
{   input();
    i % = j + k;
    output(); }
```

```
void mod(void);
extern int i;
int main(void)
{  scanf("%d", &i);
   mod();
}
int i;
extern int j;
void output(void)
{  printf(".. ", i, j);
}
```

test.c

comp.c

```
extern int i;
void output(void);
int j = 10;
int k;
void input(void)
{ scanf("%d", &k); }

void mod()
{   input();
    i % = j + k;
    output(); }
```



CODE – compiled and linked

Better way

- Create globals.h (a header file to have common declarations)

global.h

```
extern int i, j;
void mod(void);
void output(void);
```

test.c

```
void mod(void);
#include "global.h"
extern int i;

int main(void)
{  scanf("%d", &i);
    mod();
}

int i;
extern int j;

void output(void)
{  printf(".. ", i, j);
}
```

global.h

```
extern int i, j;
void mod(void);
void output(void);
```

comp.c

```
#include "global.h"
void output(void);
int j = 10;
int k;

void input(void)
{ scanf("%d", &k); }

void mod(void)
{   input();
    i % = j + k;
    output(); }
```

Information Hiding

- You can **hide** the global (external) variables and functions
  - Use static again (REMARK:meaning is **DIFFERENT!!!)**
  - The declaration limits the scope to the rest of the source file
  - These become invisible to other files

comp.c

```
#include "global.h"

int j = 10;
static int k;

void input(void)
{ scanf("%d", &k); }

void mod(void)
{   input();
    i % = j + k;
    output(); }
```

- The **global variable k** and **function input(void)** can not be accessed from other files;
- But function mod(void) can be accessed from other files