C Programming Language: Overview

UCSB CS140, T. Yang

Some of slides are modified from UCB cs61c and Stephen Edwards' lectures.

TIOBE Index of Language Popularity

https://www.tiobe.com/tiobe-index/

Nov 2023	Nov 2022	Change	Programming Language	Ratings	Change
1	1		Python	14.16%	-3.02%
2	2		C	11.77%	-3.31%
3	4	↑	C++	10.36%	-0.39%
4	3	↓	Java	8.35%	-3.63%
5	5		C#	7.65%	+3.40%
6	7	↑	JavaScript	3.21%	+0.47%
7	10	↑	PHP	2.30%	+0.61%
8	6	↓	Visual Basic	2.10%	-2.01%
9	9		SQL	1.88%	+0.07%
10	8	↓	Assembly language	1.35%	-0.83%

The ratings are based on the number of skilled engineers world-wide, courses and third party vendors.

Table of Content:
Focus on what C differs from others

- **Hello world example**
- **C vs. Java**
- **Addresses, Pointers**
- **Use of heap space with malloc/free**
- **Arrays**
- **Structures**
- **Strings**
- **Global/local variables**
- **Discussion section this Thursday**
  - Macro preprocessor, Makefile utility, Valgrind, GDB, Homework issues.

Hello World in C

```
#include <stdio.h>

void main()
{
  printf("Hello, world!\n");
}
```

Preprocessor used to share information among source files

Program mostly a collection of functions
"main" function special: the entry point
"void" qualifier indicates function does not return anything

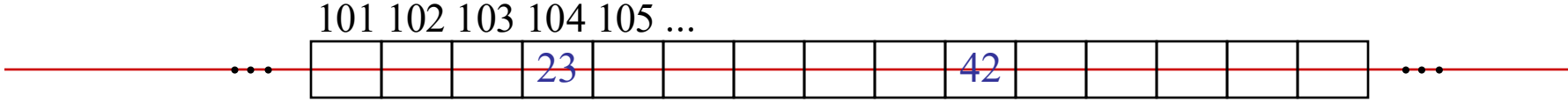
I/O performed by a library function: not included in the language

Similarities and Difference of C and Java

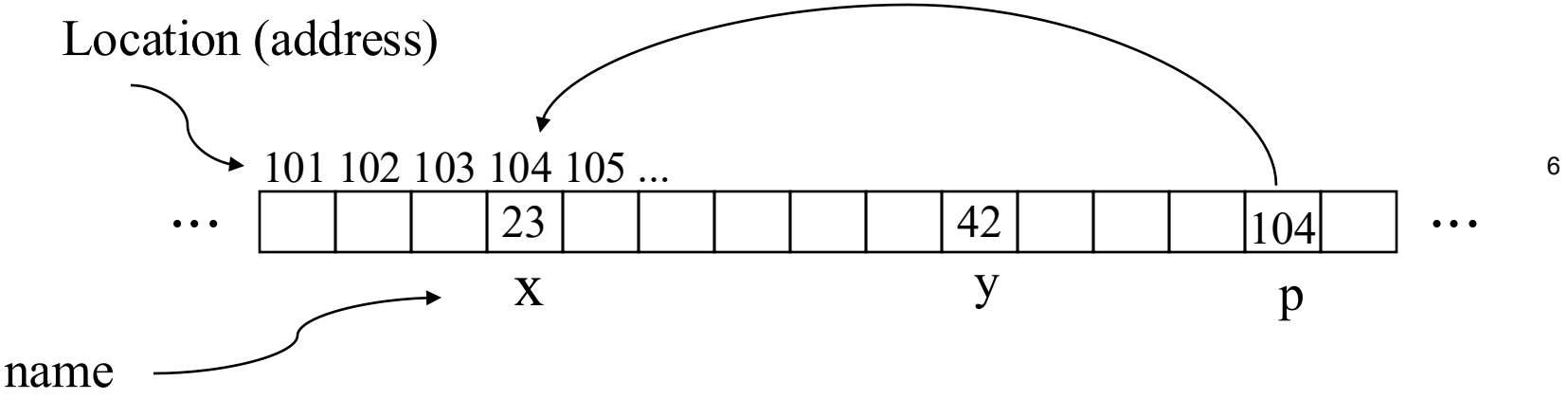
- **These structures are identical in Java and C**
  - *if* statements, *switch/case* statements
  - *while*, *do/while* loops, *for* loops
  - standard operators
    - arithmetic: +, -, *, /, %, ++, --, +=, etc.
    - logical: || , && , ! , == , != , >= , <=
    - bitwise: | , & , ^ , ~
  - First function to execute is *main*

- **Difference**
  - C has no classes
  - All work in C is done in functions
    - Variables may exist outside of any functions
    - Global variables seen by all functions declared after variable declaration

Address vs. Value in C



- **Consider memory to be a single huge array**
  - Each cell of the array has an address associated with it
  - Each cell also stores some value
  - Don't confuse the address referring to a memory location with the value stored there
- **An *address*** refers to a particular memory location; e.g., it points to a memory location
- ***Pointer***: A variable that contains the address of a variable



Pointer Syntax

- `int *x;`
  - Tells compiler that variable x is address of an `int`
- `x = &y;`
  - Tells compiler to assign address of `y` to `x`
  - `&` called the “address operator” in this context
- `z = *x;`
  - Tells compiler to assign value at address in `x` to `z`
  - `*` called the “dereference operator” in this context

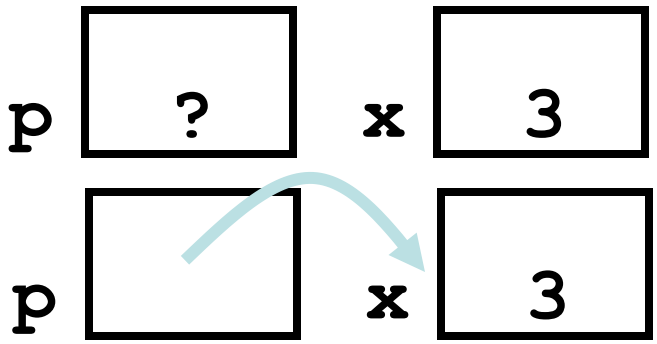
Creating and Using Pointers

- **How to create a pointer:**

  `&` operator: get address of a variable

```
int *p, x;
  x = 3;

  p = &x;
```



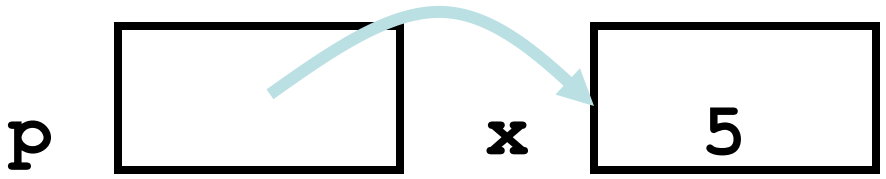
Note the "`*`" gets used 2 different ways in this example. In the declaration to indicate that `p` is going to be a pointer, and in the `printf` to get the value pointed to by `p`.

- How get a value pointed to?

  "`*`" (dereference operator): get the value that the pointer points to

```
printf("p points to %d\n",*p);
```

- **How to change a variable pointed to?** `*p = 5;`



Pointers and Parameter Passing

- **Java and C pass parameters "by value"**
  - Procedure/function/method gets a copy of the parameter, *so changing the copy cannot change the original*

```
void add_one (int x) {
    x = x + 1;
  }
int y = 3;
add_one(y);
```

y remains equal to 3

```
void add_one (int *p) {
   *p = *p + 1;
  }
int y = 3;
add_one(&y);
```

y is now equal to 4

Types of Pointers

- **Pointers are used to point to any kind of data (`int`, `char`, a `struct`, a function etc.)**
- **Normally a pointer only points to one type**
  - **`void *`** is a type that can point to anything (generic pointer)
- **Function pointer example:**

```
int add1(x){
      return x+1;
}
int use_add1() {
    int (*fun_ptr)(int) = add1;
    return (*fun_ptr)(10);
}
```

Calling use_add1() returns 11

Passing a function pointer

- **Function pointer can be used as an argument**

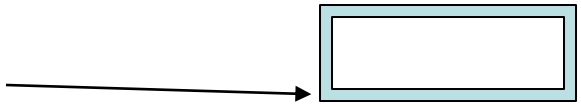
```
int add1(x){
      return x+1;
}
int  run( int (*func_prt)(int), int x ){
      func_prt(x);
}
int use_add1() {
    return  run(add1, 10);
}
```

Calling use_add1() returns 11

More C Pointer Dangers

- ***Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to!***
- **Local variables in C are not initialized, they may contain anything (aka "garbage")**
- **Is the following code legal?**

```
void f()
{
     int *ptr;
     *ptr = 5;
}
```



Error: Unallocated space

```
void g()
{
     int *ptr=NULL;
     *ptr = 5;
}
```

```
NULL means 0 in C
defined in <stdlib.h>
```

Error: illegal address.
0 is often allocated for OS

Arrays

- **Array: sequence of identical objects in memory**
  - `int a[10];` means space for ten integers
  - **By itself, a is the address of the first integer**
  - **`*a` and `a[0]` mean the same thing**
- **Legal array declarations**

```
int scores[20];
#define   MAX_LINE   80
char line[MAX_LINE];   // place 80 inside [ ] at compile time
```

- **Illegal array declaration**

```
int x = 10;
float nums[x];   // using variable for array size
```

More on C Arrays

- **C arrays (diff from Java array)**
  - Declared size of a C array must be a constant
    - Cannot use variables for the size. int a[5];
      - May use a variable with malloc() to get heap space.
    - no *.length* parameter in array
    - **Dynamic array length is allowed after C99** standard, but we do not recommend this feature for large arrays

```
int func(int size){
  int a[size];
}
```

More on C Arrays

- **No bounds checking. You may access an index outside of the array in C, but it is dangerous**
  - int a[5]; a[6]=1;
- **Arrays can be passed as parameters to functions**
  - arrays are always passed-by-reference
    - the address of the first element is passed
    - Changes made to array in the called function are seen in the calling function

Dynamic Storage Allocation with malloc/free()

- **Library routines for managing the heap**
  - Each segment allocated is contiguous in memory (no holes)
  - Segments do not move once allocated
- **malloc(size)**
  - Find a memory area large enough for segment
  - Mark that memory is allocated
- **free(pointer)**
  - Mark the segment as unallocated

```
int *a, k;
k=10;
a = (int *) malloc(sizeof(int) * k);
a[5] = 3;
free(a);
```

sizeof() returns # bytes used by this data type.

Must free allocated memory space, otherwise there is memory leak during execution

Pointers for arrays

- **Example**

```
int *p, *q, z;
p = (int *) malloc(sizeof(int)*10);
q=p+5;  /*&p[5] */
z= *(p+5); /* equivalent to p[5] */
z = (int) q-p;
add1(p); /* p[0]++ */
inc1(q); /* p[5]++ */
```

As p and q point into same array, q-p is 5

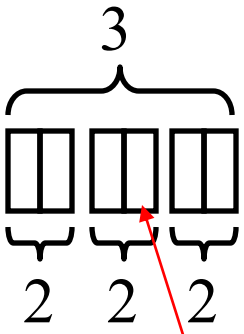
- the number of integer elements between p and q.
- q-p is NOT the number of bytes between them.

```
void add1 (int a[]){
    a[0]++;
}
void inc1 (int *a){
    a[0]++;
}
```

Multidimensional Arrays in C

- **Array declarations read right-to-left**
- **Linearized representation in memory**

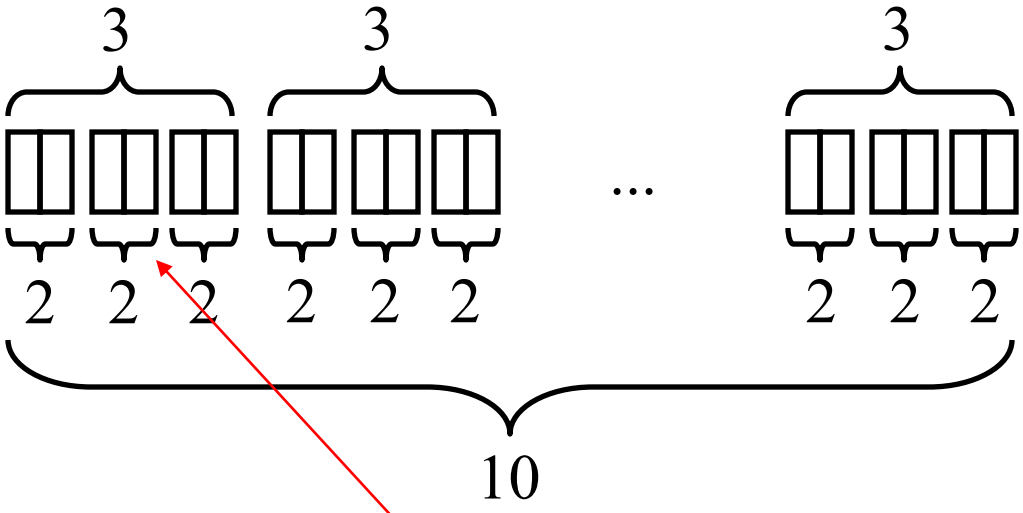
int a[3][2];



Where is a[1][1]?

Compiler converts its address as a+ 1*2 +1 which is a+3

int b[10][3][2];



Where is b[0][1][0]?

Compiler converts it as b+ 0*3*2 +1*2+0

Which is b+2

Arrays passed as arguments

- **Pass 1D array**

```
void examine( int c[] ) {
      c[5]=11;
 }
```

- **Pass 2D arrays**

```
void examine( int a[][2] ) {
      a[1][1]=11;
 }
```

- **Compiled C code computes address of `a[1][1]` as `a + 2*1+1`**
  - That requires constant 2 (2nd dimension) to be specified in the function argument.

A programmer uses a one-dimensional array to represent a 2D matrix

$A = (a_{ij})$ is an $m \times n$ matrix

allocated as

*malloc(sizeof(int)*12)*

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

stored as

0 1 2 3 4 5 6 7 8 9 10 11

Access A[i][j] with A[i*n+j]

```
void addAll(int *A, int m, int n, int x){
   for (int  i=0; i<m; i++)
         for (int j=0; j<n; j++)
               A[i*n+j] +=x;
}
```

```
void main(){
    int m=3, n=4;
    int *A= malloc(sizeof(int)*m*n);
    addAll(A, m, n, 1);
}
```

Strings

- In C, strings are just an array of characters
- C provides a standard library for copying strings, counting characters in string, concatenate strings, compare strings, etc.
- By convention, all strings are terminated by the null character (\0)

Common String Mistakes

```
char *str1="ab", *str2="cd";
if (str1 ==str2) {
        …
}
if(str1 <str2){
        …
}
```

Common String Functions

- ***int strlen(char str[]);***
  - counts the number of characters up to (but not counting) the null character and returns this number
- ***int strcpy(char strTo[], char strFrom[]);***
  - copies the string in strFrom to the string in strTo
  - make sure strTo is at least as big as strFrom
- ***int strcat(char strTo[], char strFrom);***
  - copies the string in strFrom to the end of strTo
  - again, make sure strTo is large enough to hold additional chars
- ***int strcmp(char str1[], char str2[]);***
  - compares string 1 to string 2
  - return values are as follows
    - less than 0 if str1 is lexicographically less than str2
    - 0 if str1 is identical to str2
    - greater than 0 if str1 is lexicographically greater than str2

C Structures

- **C does not have classes**
- **However, C programmers can create their own data types, called *structures***
  - Structures allow a programmer to place a group of related variables into one place. Example:

```
struct person {
    char name[30];
    int  id;
};
```

- **Variables can now be created by type *struct person***

```
struct person  bob;
bob.id=1234;
strcpy(bob.name. “Bob K”);
```

- When passed to a function, a structure is passed by value

typedef

- **It can be hassle to always type *struct person***
- **C provides a way for you to give "nicknames"**
  - it is the keyword *typedef*
- **Example**
- Using *typedef* with a standard data type

```
typdef unsigned long ulong_t
```

- Using *typedef* with a structure declaration

```
typdef struct person {
   char name[30];
   int id;
} person_t;
```

- Whenever a *struct person* is needed, just type *person_t*

Pointers and Structures: Example

```
typedef struct {
    int x;
    int y;
} Point;

Point p1;
Point p2;
Point *paddr;
```

```
/* dot notation */
int h = p1.x;
p2.y = p1.y;

/* arrow notation */
int h = paddr->x;
int h = (*paddr).x;

/* This works too to copy */
p1 = p2;
```

Pointers and Structures: Example in Exercise 1

```
struct key_action {
    char *cmd;
    int (*func)();
};
```

```
Int set_key_action(
  struct key_action *rec,
  char *cmd, int (*f)()){
       if(rec!=NULL) {
              rec->cmd=cmd;
              rec->func=f;
               return 1;
            }
       return 0;
}
```

```
int del1(int x){
        return x-1;
}

char * test(void){
   struct key_action rec;
   char *key="del1";
   int ret=set_key_action(&rec, key, del1);

   mu_assert("Error in set_key",
                    strcmp(key, rec.cmd)==0);
   mu_assert("Error in set_action",
                    rec.func == del1);

   /*All comparisons  are valid so far*/
   return NULL;
}
```

Macro mu_assert () is used for testing in the rest of quarter. Read code to learn implementation

Pointers and Structures: Example in Exercise 1

```
struct key_action {
    char *cmd;
    int (*func)();
};
```

```
#define
mu_assert(msg,cond)
do {
  if (!(cond))
            return msg;
} while (0)
```

```
int del1(int x){
            return x-1;
}

char * test(void){
   struct key_action rec;
   char *key="del1";
   int ret=set_key_action(&rec, key, del1);

   mu_assert("Error in set_key",
                  strcmp(key, rec.cmd)==0);
   mu_assert("Error in set_action",
                  rec.func == del1);

   /*All comparisons  are valid so far*/
   return NULL;
}
```

Macro mu_assert () is used for testing in the rest of quarter. Read code to learn implementation

Testing Example

```
int add(int x, int y){
        return  x+y+1;
}
```

```
#define
mu_assert(msg,cond)
do {
 if (!(cond))
        return msg;
} while (0)
```

```
char * test(void){
  int ret= add(4,3);
  mu_assert("Error in func add", rec == 7);
   int ret= add(10,3);
  mu_assert("Error in func add", rec == 13);
  return NULL;
}
```

```
char * test(void){
  int ret= add(4,3);
  do {
       if(!( ret==7) )
                return "Error in func add";
   } while (0);
   int ret= add(10,3);
   do {
        if(!( ret==13) )
                return "Error in func add";
   } while (0);
   return NULL;
}
```

Global & Local Variables and Constants

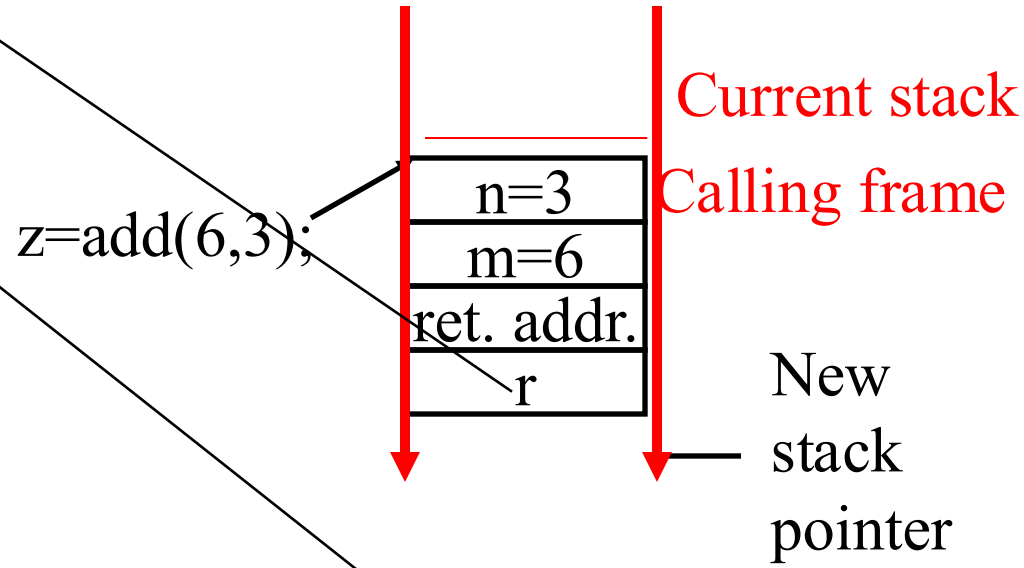
- **Variables declared outside any scope are called global**
  - they can be used by any function declared after them
- **Local variables only exist within their scope**
  - must be declared at the very beginning of the scope
  - stored on the stack
  - destroyed when scope ends
- **Prefer not to use global variables if possible**
  - too many naming conflicts
  - can be confusing to follow in large programs
- **Constants are usually declared globally**
  - use the *const* key word

Important concept for shared memory parallel programming
Learn where global variables are allocated by C compiler/OS

Example

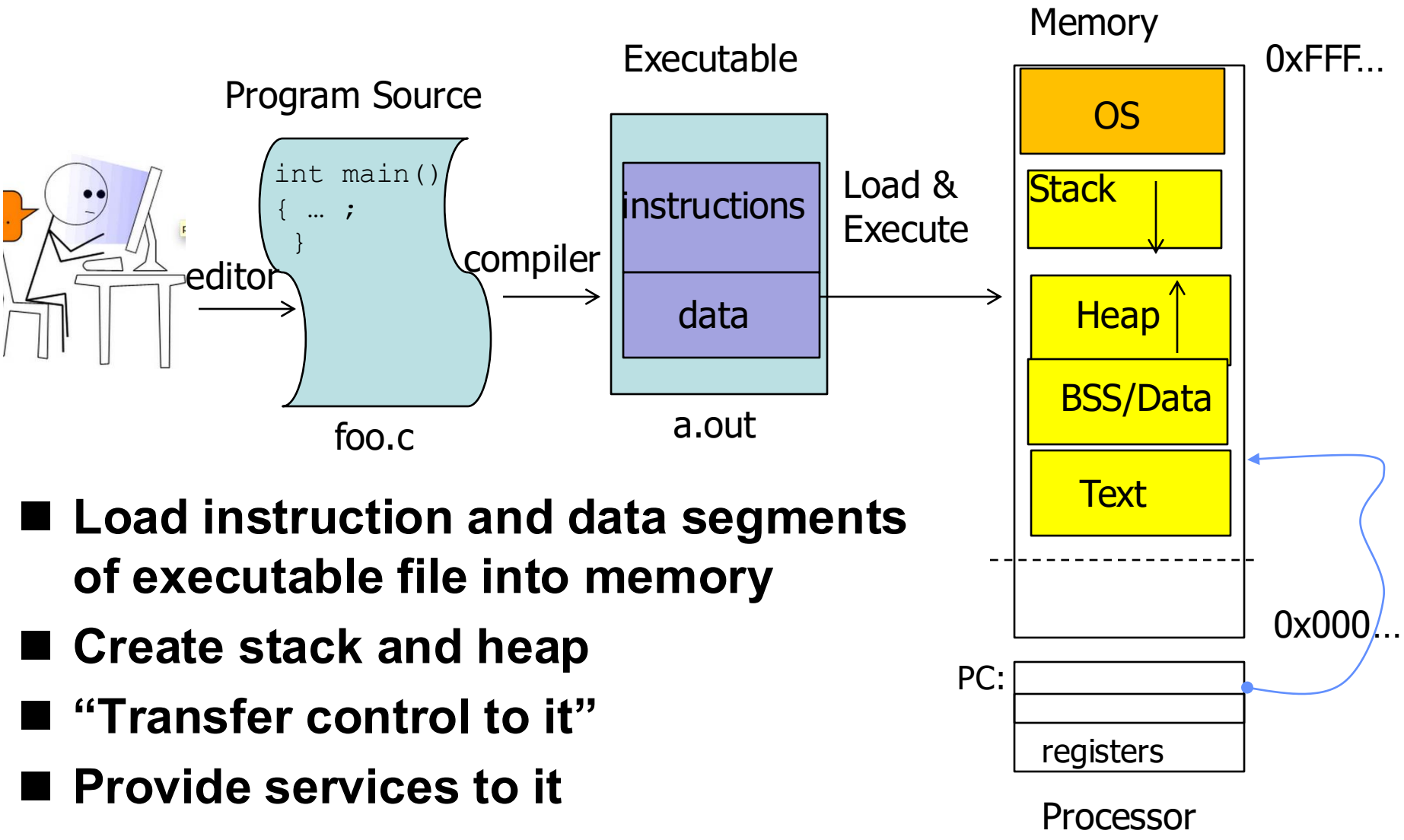
```
int r;  /* a global variable*/
int add(int m, int n)
{
    int r;/*local variable*/
    static int count=0;
    r=m+n;
    count++;
    return r;
}
```

Automatic variable allocated on stack when function called, released when it returns.



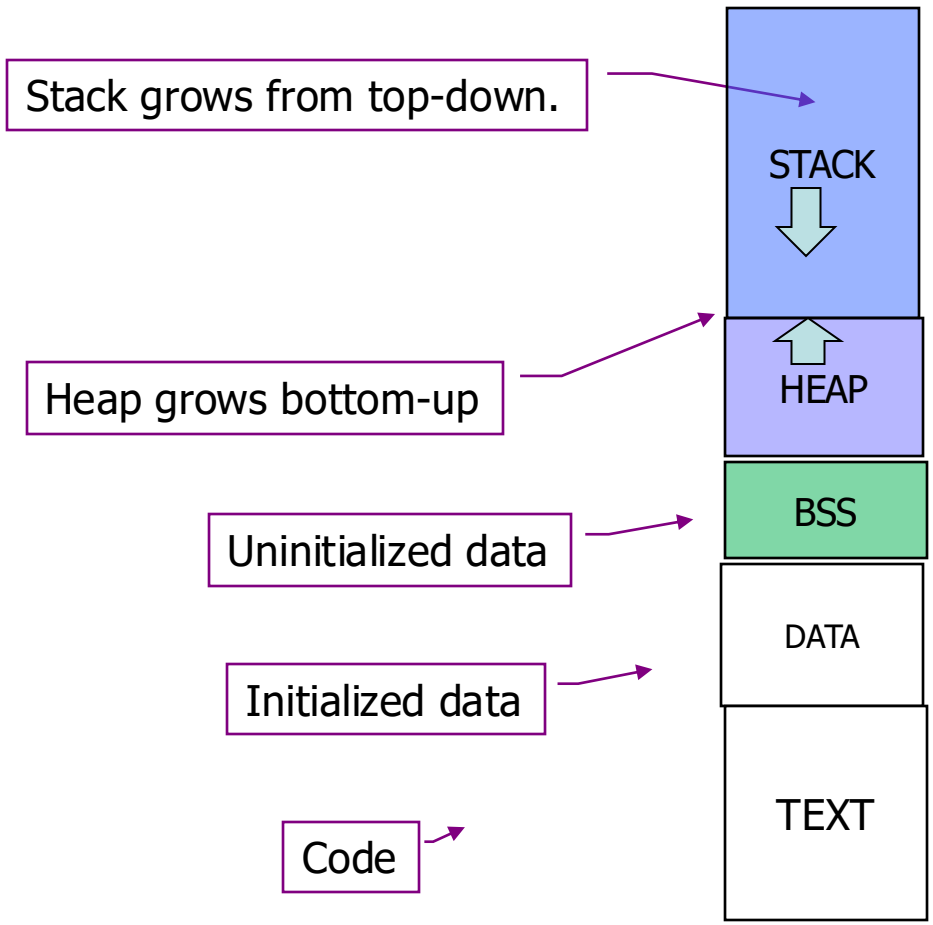
Where are static varilable count and global variable r located?

How does OS run a C program?



- **Load instruction and data segments of executable file into memory**
- **Create stack and heap**
- **"Transfer control to it"**
- **Provide services to it**

Space usage during execution of a C program



STACK for function call frames

HEAP for dynamically allocated space (malloc)

BSS segment contains all uninitialized global variables and static variables or 0 initially

DATA segment contains initialized global or static variables

Text segment contains binary code + constants

TEXT, DATA, BSS, HEAP and STACK in C

```
int f3=3; /*Initialized DATA segment */
int f1; /*Unitialized BSS segment*/
char def[] = "1";
int main(void) {
    static char abc[12]; /* BSS segment */
    static float pi = 3.14159;
    int i = 3; /* Stack*/
    char *cp;
    cp= malloc(10);  /* HEAP for allocated chunk*/
    f1= add1(i); /* code is in TEXT. f1 on STACK*/
    strcpy(abc , "Test" );
}
int add1( int f3){
    return f3+1;
}
```

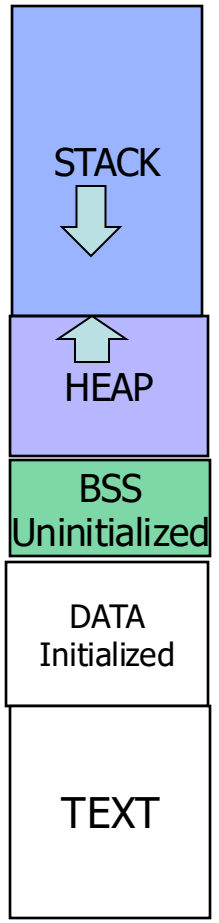
Where is def?

Where is pi?

where is cp?

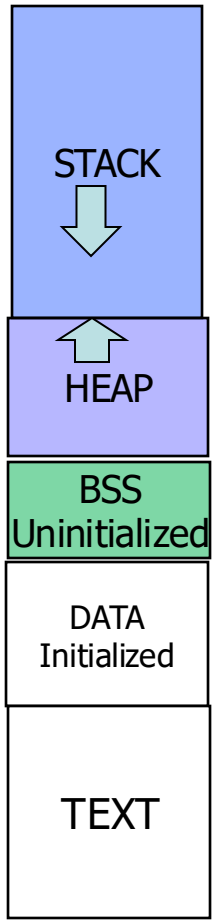
Where is "Test"?

where is f3?

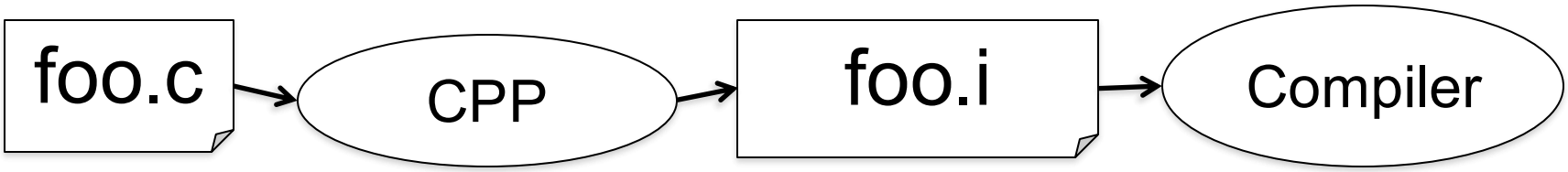


TEXT, DATA, BSS, HEAP, and STACK in C

```
Int f3=3; /* Initialized DATA segment */
Int f1; /*Uninitialized BSS segment*/
char def[] = "1"; /* DATA segment */
int main(void)
    {
    static char abc[12], /* BSS segment */
    static float pi = 3.14159; /* DATA segment */
    int i = 3; /* Stack*/
    char *cp;    /*stack*/
    cp= malloc(10);  /*malloc allocates space from HEAP*/
    f1= add1(i);  /* code is in TEXT*/
    strcpy(abc , "Test" ); /* "Test" is  located in TEXT */
}
int add1( int f3}{/*stack*/
    return f3+1;
}
```



C Pre-Processor (CPP)



- **C source files first pass through macro processor, CPP, before compiler sees code**
- **CPP replaces comments with a single space**
- **CPP commands begin with "#"**
  - #include "file.h" /* Inserts file.h into output */
  - #include <stdio.h> /* Looks for file in standard location */
  - #define M_PI (3.14159) /* Define constant */
  - #if/#endif /* Conditional inclusion of text */
- **Use –save-temps option to gcc to see result of preprocessing**
- **Full documentation at:** `http://gcc.gnu.org/onlinedocs/cpp/`

Concluding remarks

- **Pointer is a C version (abstraction) of a data address**
  - `*` "follows" a pointer to its value
  - `&` gets the address of a value
  - Arrays and strings are implemented as variations on pointers, with linearized memory structure
  - Use pointers with care: they are a common source of bugs in programs
- **Space allocation for global vs local variables.** Important for understanding data location in parallel code
- **C pre-processing**
- Read code of Makefile, minunit.h, and minunit.c in Exercise 1 released soon on how C programs are tested and graded this quarter.