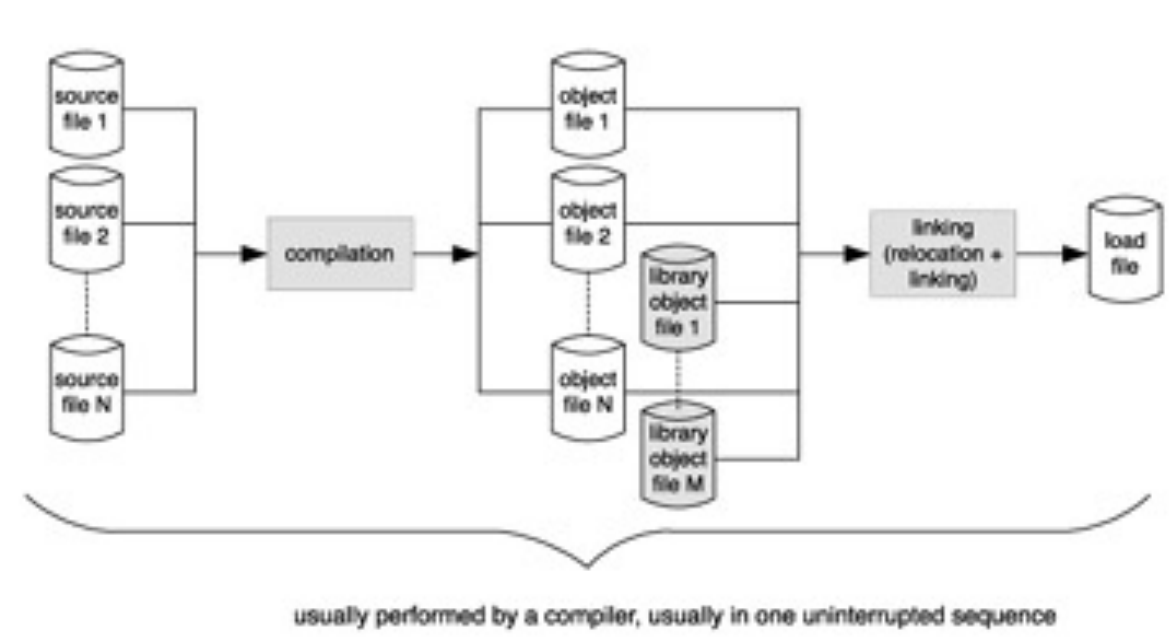
Memory Management and pointers in C

Content

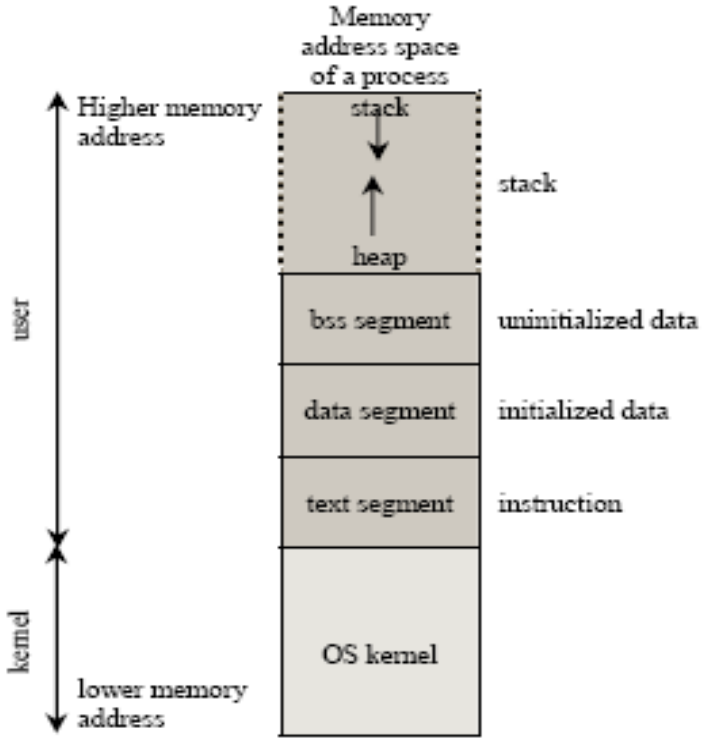
- Part 1: Memory Management
  - From Source file to executable file
  - Memory layout for a process
  - Stack
  - Heap
- Part 2: Pointers in C
  - What is a pointer?
  - Pointers & Arrays
  - Pointer arithmetic
  - Pointer to structure
  - Pointer & parameter passing

Part 1: Memory Management

- From source file to executable file



- Memory layout for a process



- **Code segment or text segment: C**ontains the code executable or code binary
- **Data segment:** sub divided into two parts
  - Initialized data segment: global, static and constant data
  - Uninitialized data segment: All the uninitialized data are stored in BSS.
- **Heap:** Memory allocated using calloc and malloc function at runtime. Heap grows upward.
- **Stack:** Used to store your local variables and also when making function calls for the activation frame. Stack grows downward.

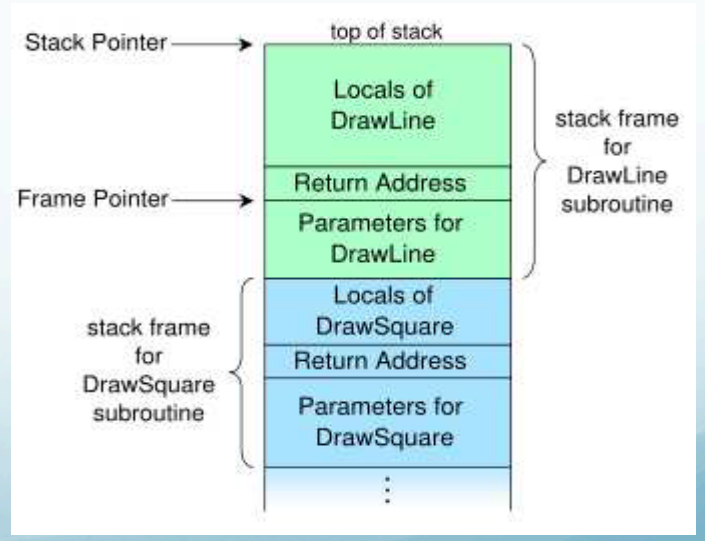
```
int a[10];

 void main()
{
   int b[2];
   int *k;
   static int m=3;
   k=(int *)malloc(sizeof(int));

  /*other codes*/

}
```

- Stack :
  - Regions of memory space where data is added and removed in a last-in-first-out manner
  - The stack is attached to a thread, so when the thread exits the stack is automatically and efficiently reclaimed. The size of a stack can be set when a thread is created: e.g., in pthread library, pthread_attr_setstacksize().
  - Faster to allocate stack than heap.

- Call Stack
  - A stack data structure that stores information about the active subroutines of a program
  - To keep track of the point to which each active subroutine should return control when it finishes.
  - A call stack is composed of stack frames. Each stack frame corresponds to a call which has not yet terminated with a return

Call stack for code:

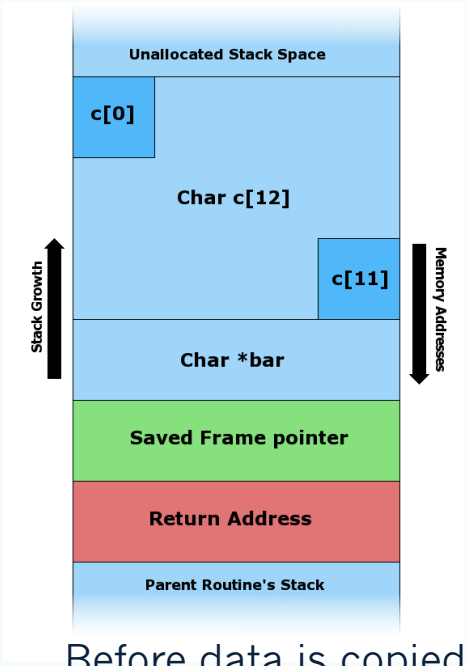
```
DrawSquare(Point topleft, int len)
{
    DrawLine(topleft, topleft+len);
}
```



- Stack overflow
  - Too much of the stack is used (mostly from inifinite loops or too much recursion, very large allocations)

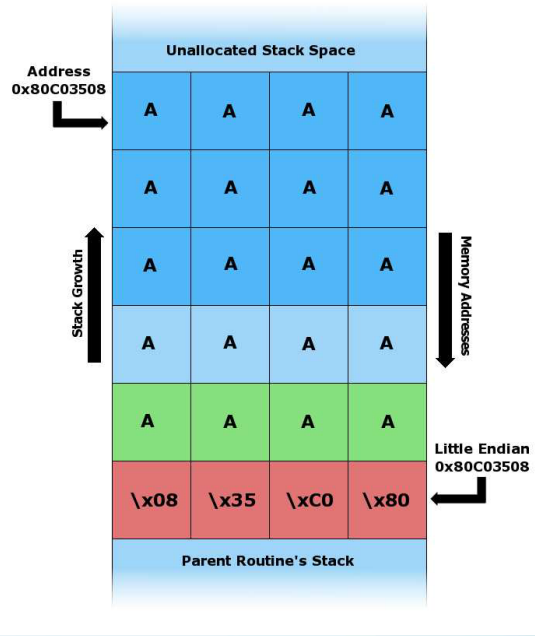
```
int main()
{
    main();
}
```

```
int add(int n)
{
    return n + add(n + 1);
}
```



Before data is copied

```
#include <string.h>
void foo (char *bar)
{
        char c[12];
        strcpy(c, bar); // no bounds
        checking...
}

int main (int argc, char **argv)
{
        foo(argv[1]);
}
```

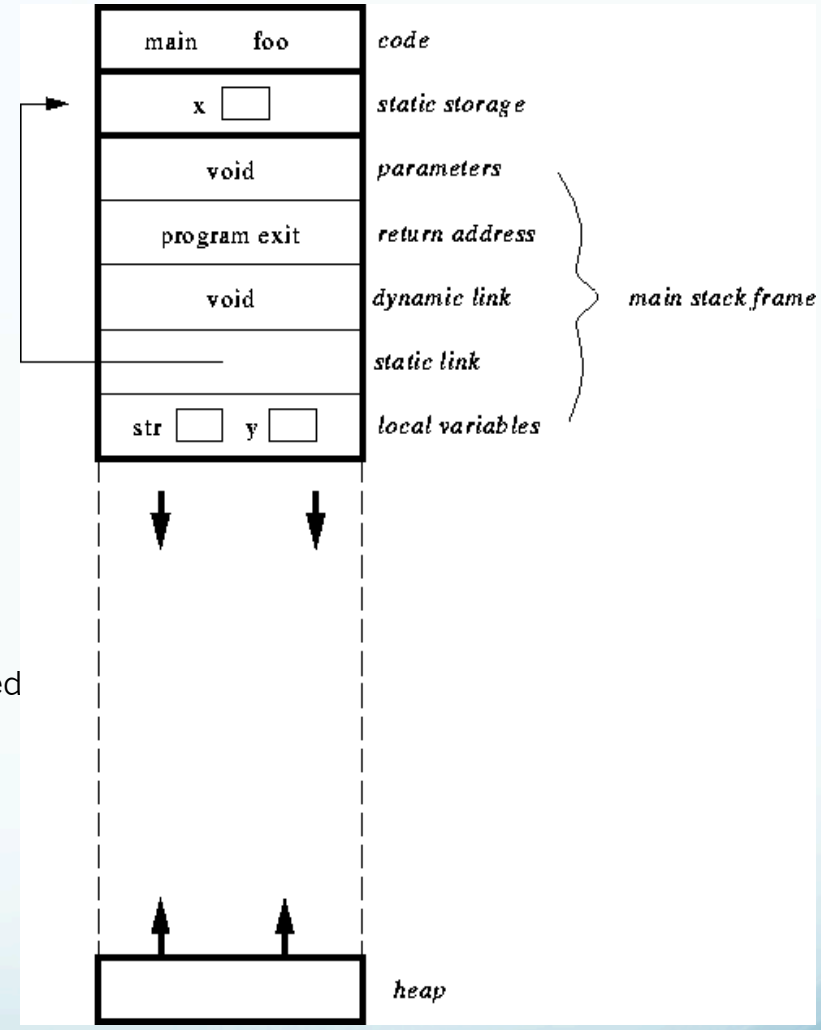


After data >12 chars is copied

- Heap
  - The heap contains a linked list of used and free blocks
  - Memory allocation using malloc, calloc and realloc or new
  - The size of the heap is set on application startup, but can grow as space is needed (the allocator requests more memory from the operating system)
  - Variables on the heap must be destroyed manually and never fall out of scope. The data is freed with delete, delete[] or free
  - Slower to allocate in comparison to variables on the stack.

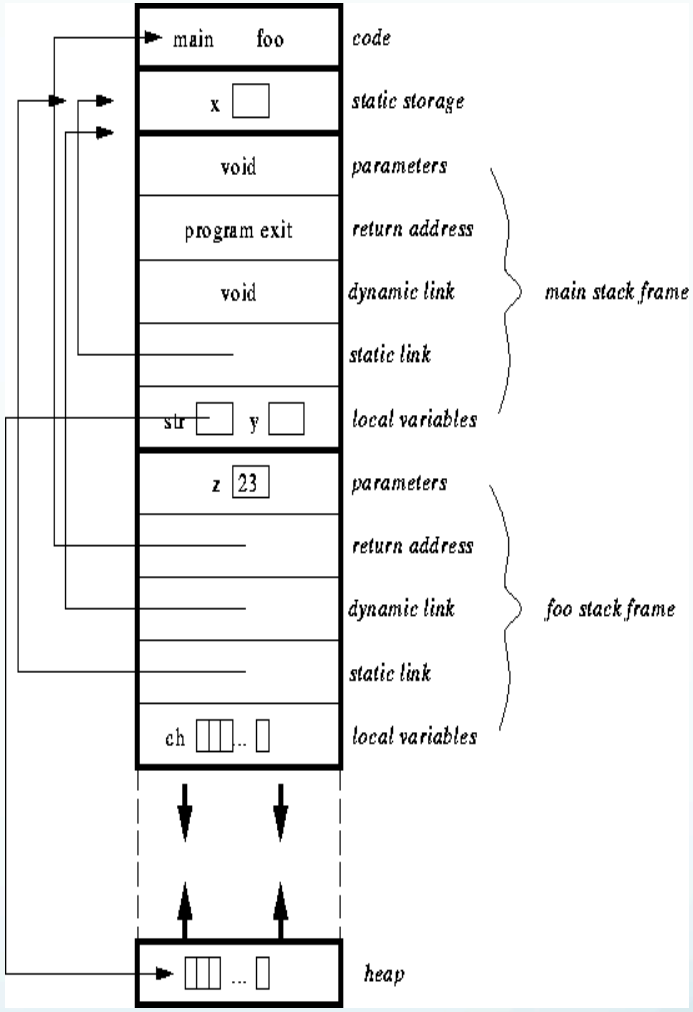
- Note:
  - Can have fragmentation when there are a lot of allocations and deallocations
  - Can have allocation failures if too big of a buffer is requested to be allocated.
  - You would use the heap if you don't know exactly how much memory you will need at runtime or if you need to allocate a lot of data.

```
int x;                    /* static storage */

void main()
{
      int y;              /* dynamic stack storage */
      char *str;          /* dynamic stack storage */
      str = malloc(100);      /* allocates 100 bytes of dynamic heap
                                 storage */
      y = foo(23);
      free(str);              /* deallocates 100 bytes of dynamic heap
                                 storage */
}                             /* y and str deallocated as stack frame is
                                 popped */

int foo(int z)    /* z is dynamic stack storage */
{
       char ch[100];      /* ch is dynamic stack storage */
      if (z == 23) foo(7);
      return 3;     /* z and ch are deallocated as stack frame is popped
                       3 put on top of stack */
}
```



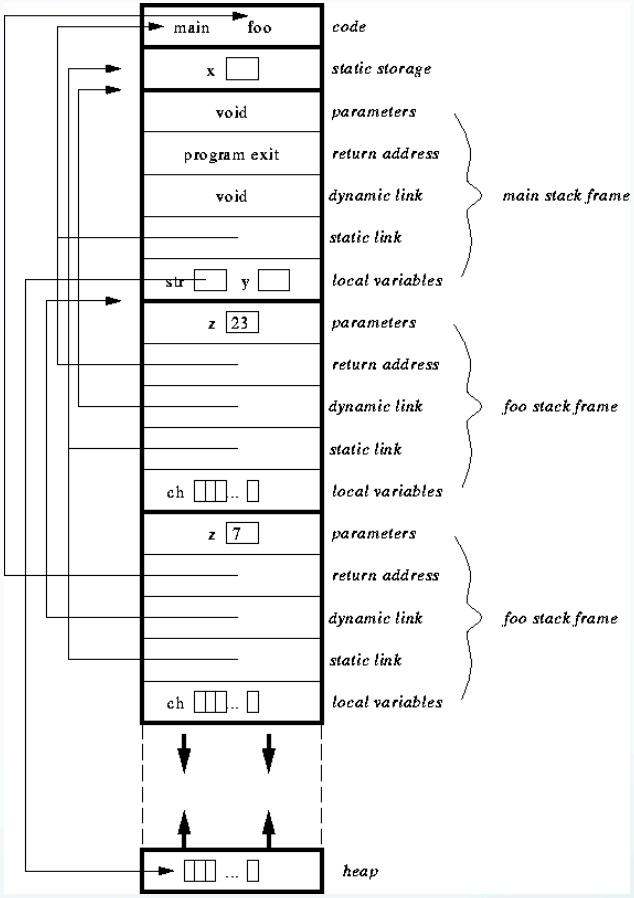
At the start of the program

```
int x; /* static storage */

void main()
{
      int y; /* dynamic stack storage */
      char *str; /* dynamic stack storage */
      str = malloc(100); /* allocates 100 bytes of dynamic
      heap storage */
      y = foo(23);
      free(str); /* deallocates 100 bytes of dynamic heap
      storage */
} /* y and str deallocated as stack frame is popped */

int foo(int z) /* z is dynamic stack storage */
{
       char ch[100]; /* ch is dynamic stack storage */
      if (z == 23) foo(7);
      return 3; /* z and ch are deallocated as stack frame
                        is popped, 3 put on top of stack */
}
```



At the first call of foo

```
int x; /* static storage */

void main()
{
      int y; /* dynamic stack storage */
      char *str; /* dynamic stack storage */
      str = malloc(100); /* allocates 100 bytes of dynamic
      heap storage */
      y = foo(23);
      free(str); /* deallocates 100 bytes of dynamic heap
      storage */
} /* y and str deallocated as stack frame is popped */

int foo(int z) /* z is dynamic stack storage */
{
       char ch[100]; /* ch is dynamic stack storage */
      if (z == 23) foo(7);
      return 3; /* z and ch are deallocated as stack frame is
      popped,
      3 put on top of stack */
}
```



At the second call of foo

Part 2: Pointers

- What is a pointer?
  - Pointer: a special type of variable which holds the address or location of another variable
  - Pointee: the variable which a pointer points to

  - Reference: x =&a;
  - Dereference: a=*x;
  - Pointer assignment : two pointer x,y , x=y

- Memory allocation and deallocation
  - void * malloc (numofBytes)
  - void free(p)
  - void * calloc(numItems, itemSize)
  - void * realloc(* oldspace, sizeNewSpace)

Memory Leaks

- A memory leak, in computer science (in such context, it's also known as leakage), occurs when a computer program consumes memory but is unable to release it back to the operating system.

```
#include <stdlib.h>
int main(void)
{
        while (malloc(50));
         /* malloc will return NULL sooner or later,
                due to lack of memory */

        return 0;
         /* free the allocated memory by operating
        system itself after program exits */
}
```

```
#include <stdlib.h>
void f(void)
{
        int* x = malloc(10 * sizeof(int));
        x[10] = 0;
        // problem 1: heap block overrun
} // problem 2: memory leak -- x not freed

int main(void)
{
        f();
        return 0;
}
```

- Pointers & Arrays
  - int a[10] ;

    a is a const pointer to the first element of the array.

  - a[1]:

    *(a+1)

- Pointer arithmetic

```
Int main()

{
    char str[] = "ABCDEFG";
    char *PC = str, *PC2 = PC + 1;
    short X = 33; short *PX = &X;
    printf("%c ", *PC ) ;

    /* Pointer comparison (==, !=) */
    if (PC != PC2) printf ("PC and PC2 are different") ;

    /*pointer arithmetic */
    /*pointer + number -> pointer */
    PC += 4; printf("%c ", *PC ) ;
    PC--; printf("%c ", *PC ) ;
    /* pointer - pointer -> number */
    printf("%d ", (PC2 - PC) ) ;

}
```

- Pointer to structure

```
struct date
{int month;int  day ; int year;};
int main()
{
    struct date *my_date;
    my_date = (struct date *)malloc(sizeof(struct date));
    (*my_date).year = 1776;
    my_date->month = 7;
    my_date->day = 4;
    /* code using my_date */
    free(my_date);
}
```

Exercise: Compare (*s).name vs. (s.name) vs. *s.name

Hint: priority(.) > priority(*))

- Pointer & parameter passing

```
#include <stdio.h>
void swap(int *i, int *j)
{
    int t;
    t = *i;
    *i = *j;
    *j = t;
}
Int main()
{
    int a,b;
    a=5;
    b=10;
    printf("%d %d”,a,b);
    swap(&a,&b);
    printf("%d %d”,a,b);
}
```

- Note:
  - Avoid Memory allocation in infinite loop
  - Use dereferencing operator (*, ->) whenever needed.
  - Never reference a variable after it is deallocated.
  - Use malloc(), free() with non-pointer arguments

- Reference:
  - Memory as a Programming Concept in C & C++
  - Cs352 Blackboard: mem_in_c.pdf
  - Cs352 Blackboard: memorymgmt.pdf
  - http://en.wikipedia.org/wiki/Loader_(computing)
  - http://en.wikipedia.org/wiki/Call_stack

Questions