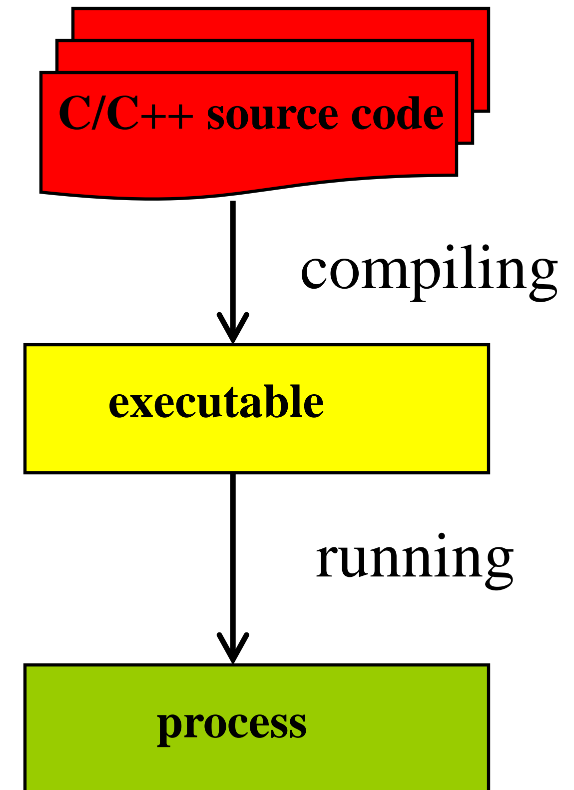
C/C++ Memory Layout

Humayun Kabir
Professor, CS, Vancouver Island University, BC, Canada

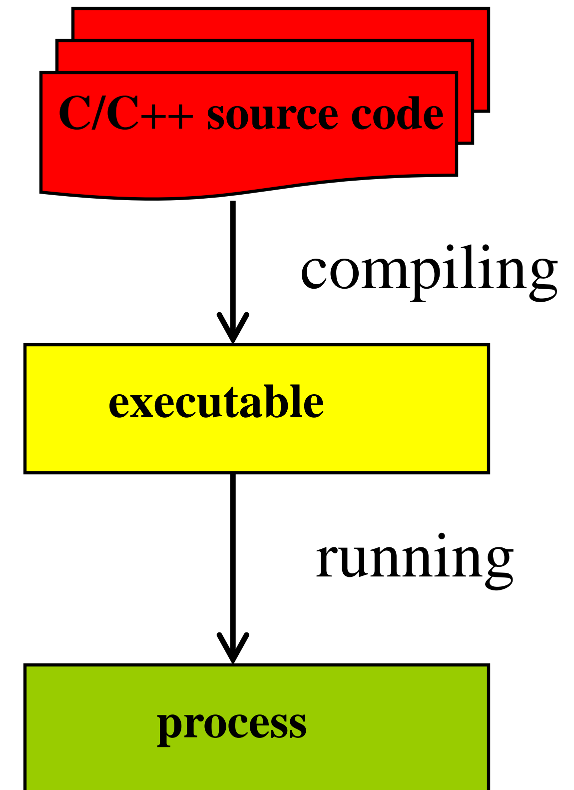
Code vs. Executable vs. Process

- C source code
  - C statements organized into functions
  - Stored as a collection of files (.c and .h)
- C++ source code
  - C++ statements organized into both classes and functions.
  - C++ classes contains member variables and functions.
  - Stored as a collection of files (.cpp and .hpp)



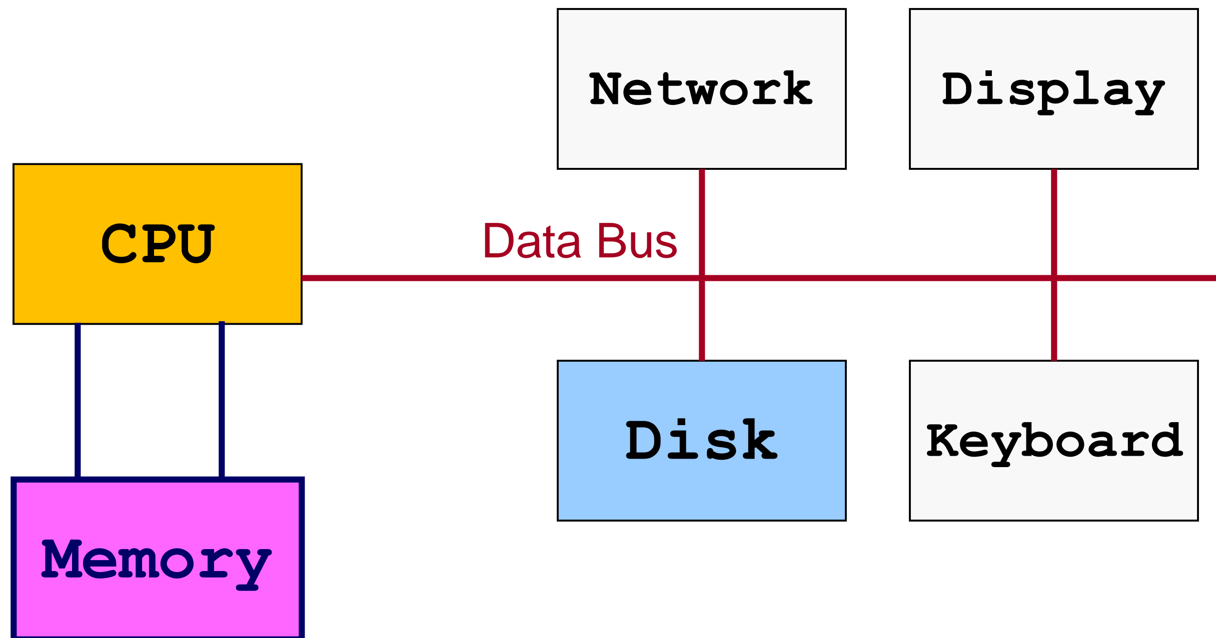
Code vs. Executable vs. Process

- Executable module
  - Binary image generated by compiler
  - Stored as a file (e.g., *a.out*)
- Process
  - Instance of a program that is executing
    - With its own address space in memory
    - With its own id and execution state
  - Managed by the operating system

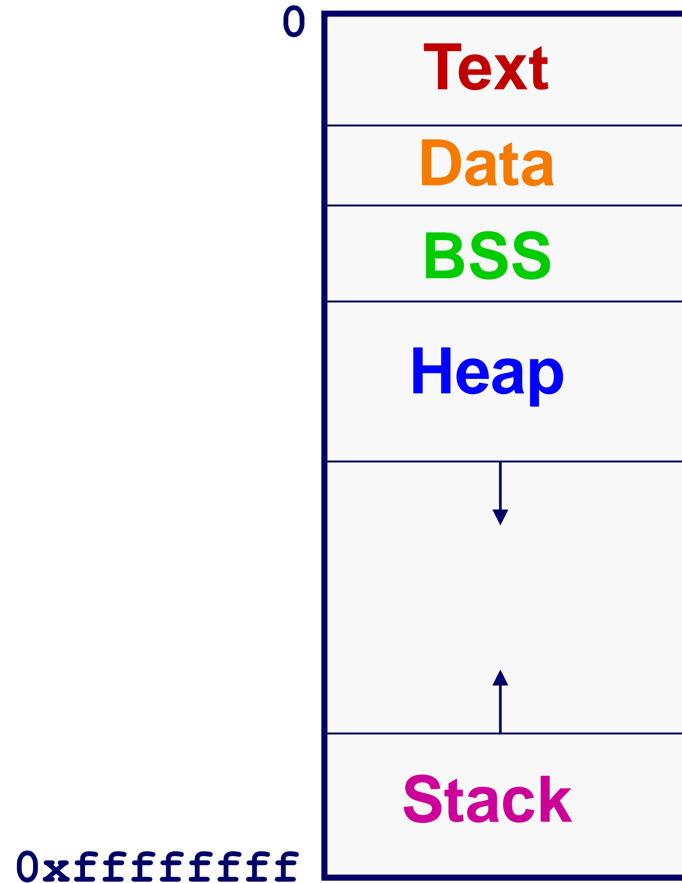


Main Memory on a Computer

- What is main **memory**?
  - Storage for variables, data, code, etc.
  - Shared among many processes

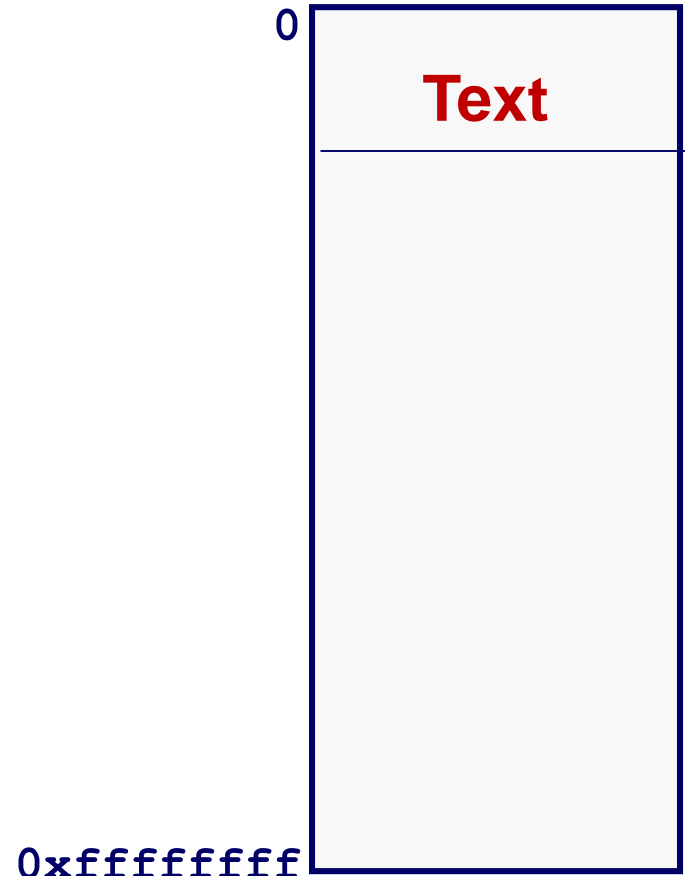


Memory Segments of a Process



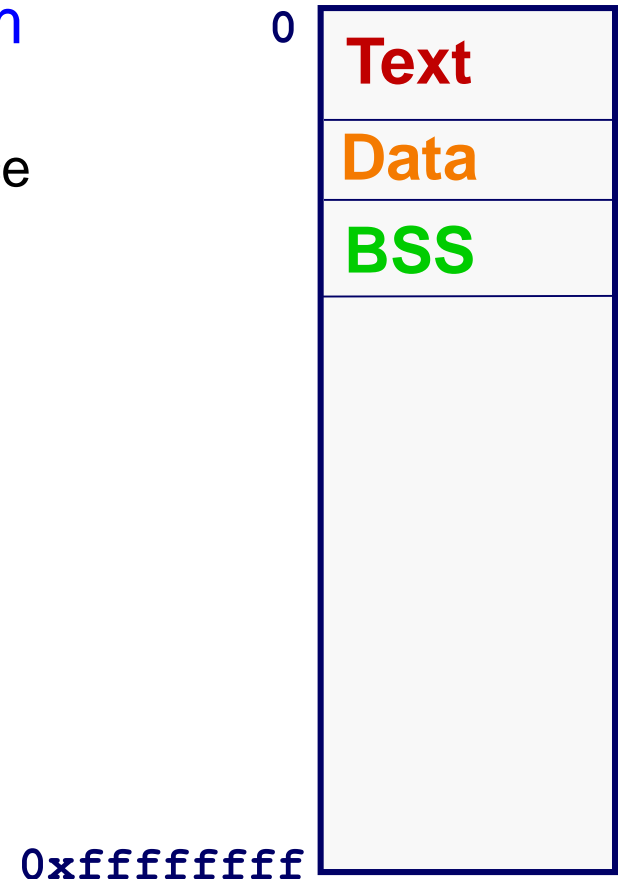
What to Store: Code and Constants

- **Executable code** and **constant data**
  - Program binary, and any shared libraries it loads
  - Necessary for OS to read the commands
- OS knows everything in advance
  - Knows amount of space needed
  - Knows the contents of the memory
- Known as the "code/text" segment



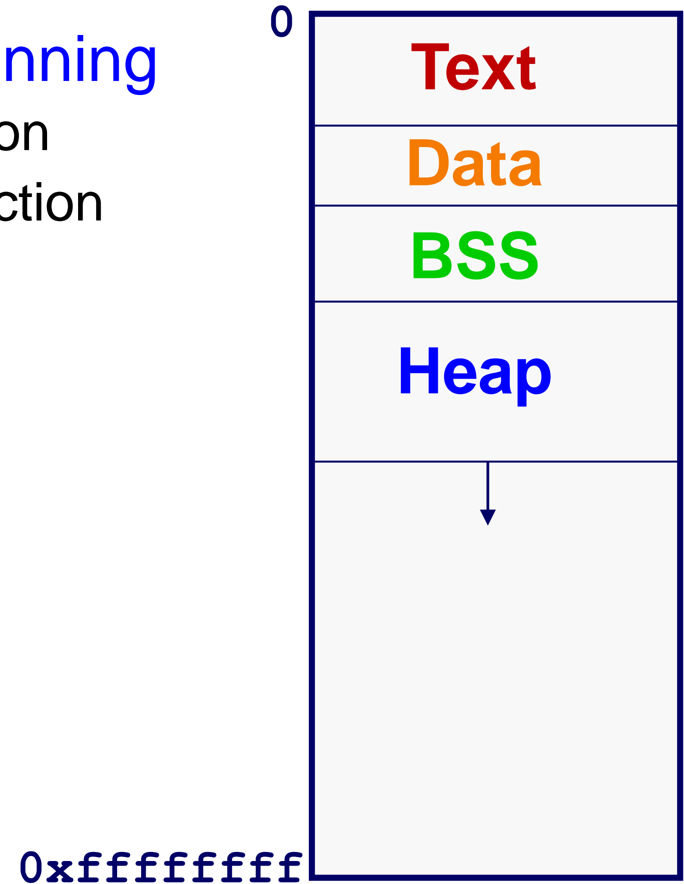
What to Store: Global and "Static" Data

- Variables that exist for the entire program
  - Global variables, and "static" local variables
  - Amount of space required is known in advance
- Data: initialized in the code
  - Initial value specified by the programmer
    - E.g., "**`int x = 97;`**"
  - Memory is initialized with this value
- BSS: not initialized in the code
  - Initial value not specified
    - E.g., "**`int x;`**"
  - All memory initialized to 0 (on most OS's)
  - BSS stands for "Block Started by Symbol"



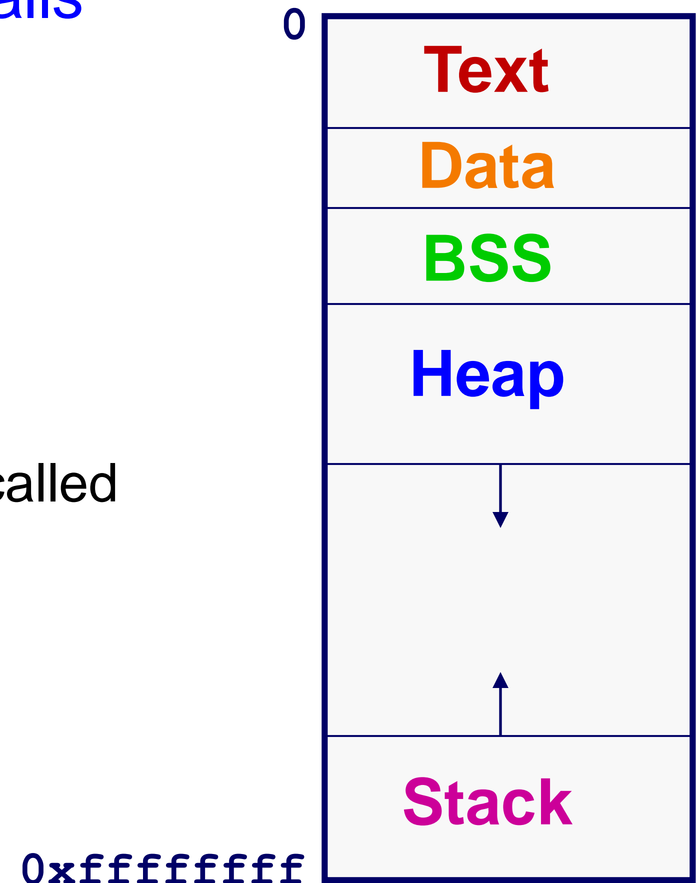
What to Store: Dynamic Memory

- Memory allocated while program is running
  - E.g., allocated using the `malloc()` function
    - And deallocated using the `free()` function
- OS knows nothing in advance
  - Doesn’t know the amount of space
  - Doesn’t know the contents
- So, need to allow room to grow
  - Known as the “heap”



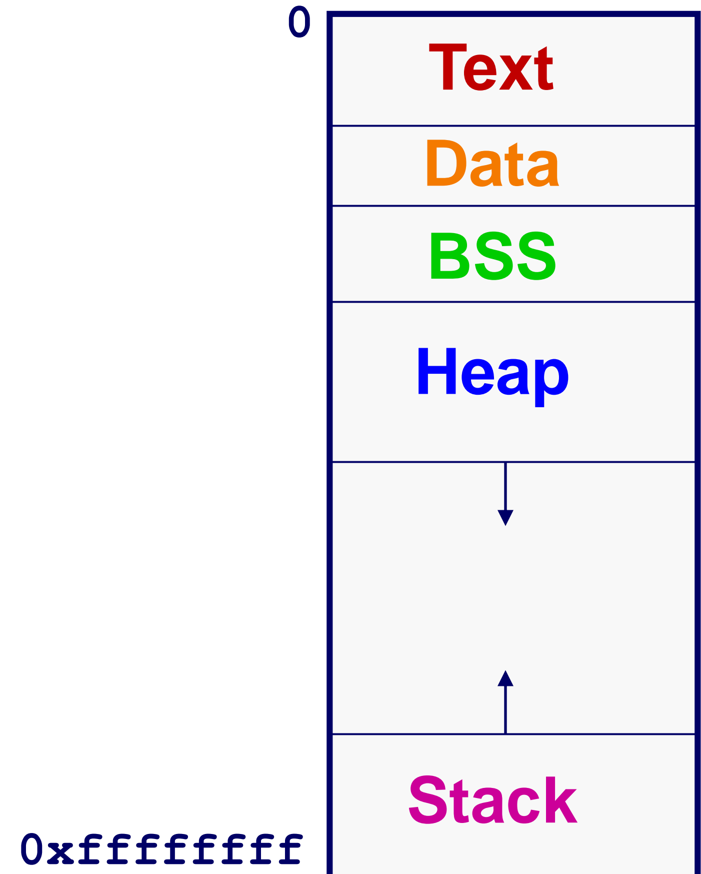
What to Store: Temporary Variables

- Temporary memory during lifetime of a function or block
  - Storage for function parameters and local variables
- Need to support nested function calls
  - One function calls another, and so on
  - Store the variables of calling function
  - Know where to return when done
- So, must allow room to grow
  - Known as the "**Stack**"
  - Push on the stack as new function is called
  - Pop off the stack as the function ends



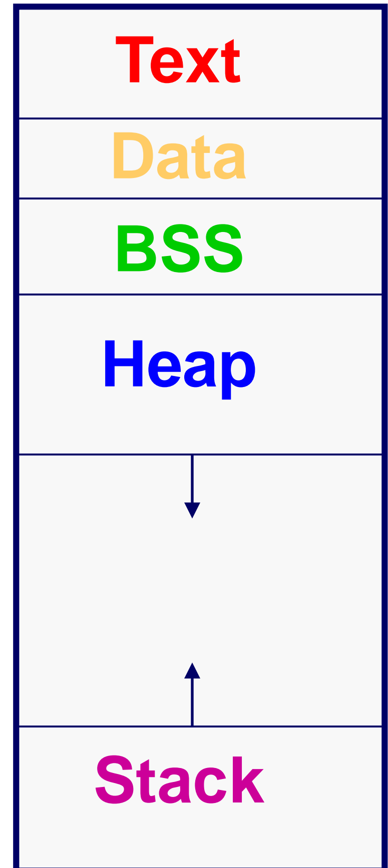
Memory Layout: Summary

- **Text**: code, constant data
- **Data**: initialized global & static variables
- **BSS**: uninitialized global & static variables
- **Heap**: dynamic memory
- **Stack**: local variables



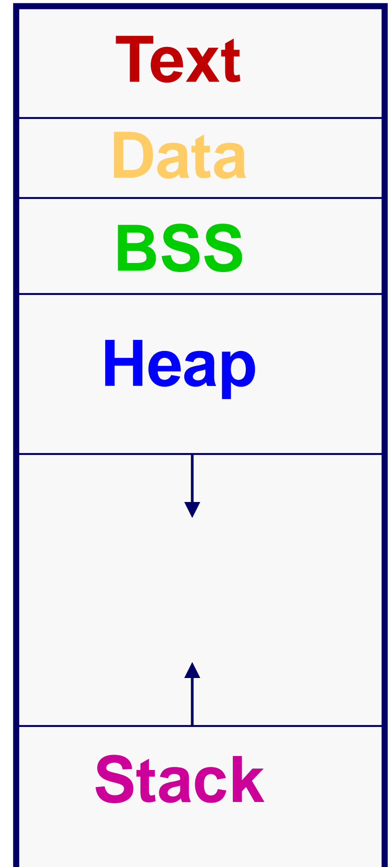
Memory Layout Example

```
char* string = "hello";
int iSize;
static char* st_strg = "world";
static int st_global;
char* f(void)
{
     static char* st_strl = "wow!";
     static int st_local;
     char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```



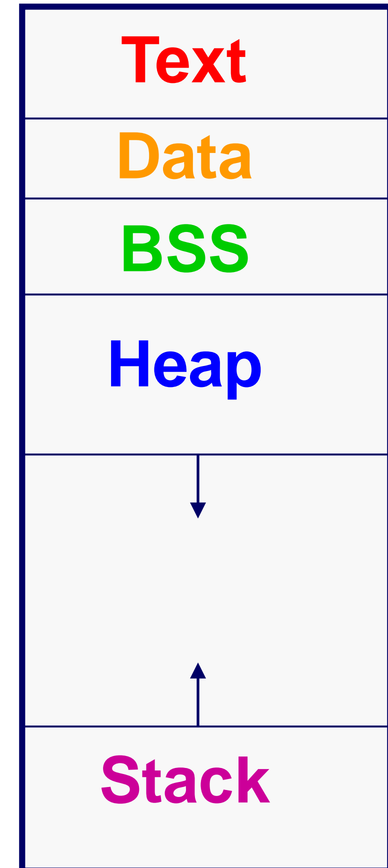
Memory Layout Example: Text

```
char* string = "hello";
int iSize;
static char* st_strg = "world";
static int st_global;
char* f(void)
{
    static char* st_strl = "wow!";
    static int st_local;
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```



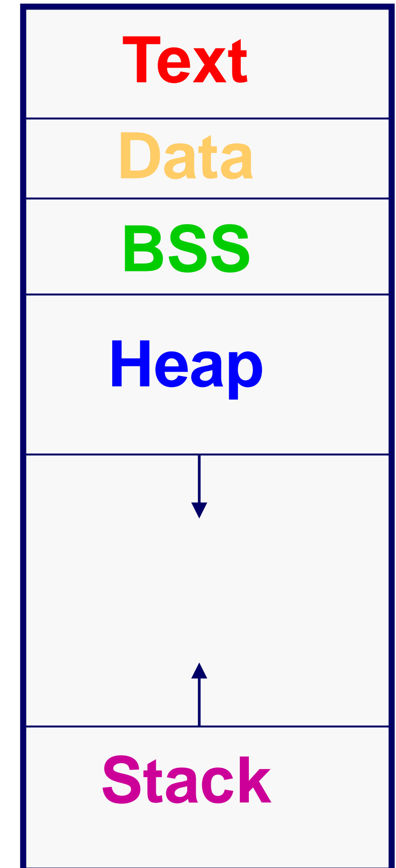
Memory Layout Example: Data

```
char* string = "hello";
int iSize;
static char* st_strg = "world";
static int st_global;
char* f(void)
{
    static char* st_strl = "wow!";
    static int st_local;
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```



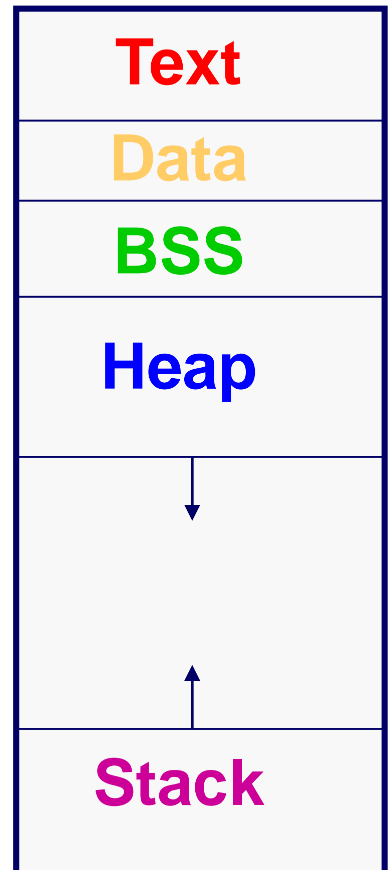
Memory Layout Example: BSS

```
char* string = "hello";
int iSize;
static char* st_strg = "world";
static int st_global;
char* f(void)
{
    static char* st_strl = "wow!";
    static int st_local;
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```



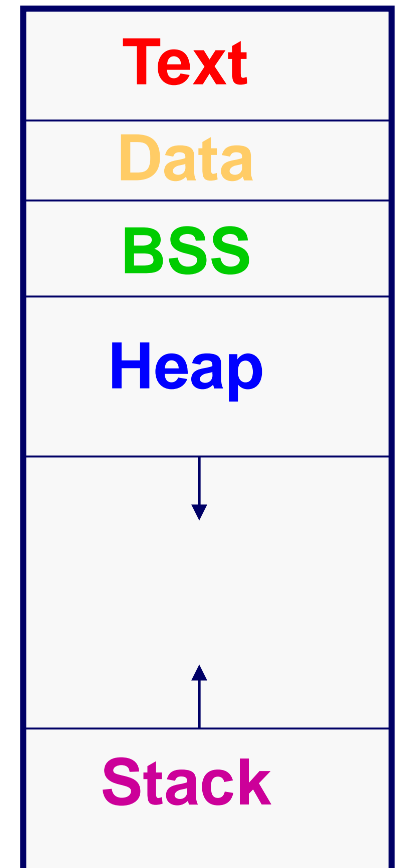
Memory Layout Example: **Heap**

```
char* string = "hello";
int iSize;
static char* st_strg = "world";
static int st_global;
char* f(void)
{
     static char* st_strl = "wow!";
     static int st_local;
     char* p;
     iSize = 8;
     p = malloc(iSize);
     return p;
}
```



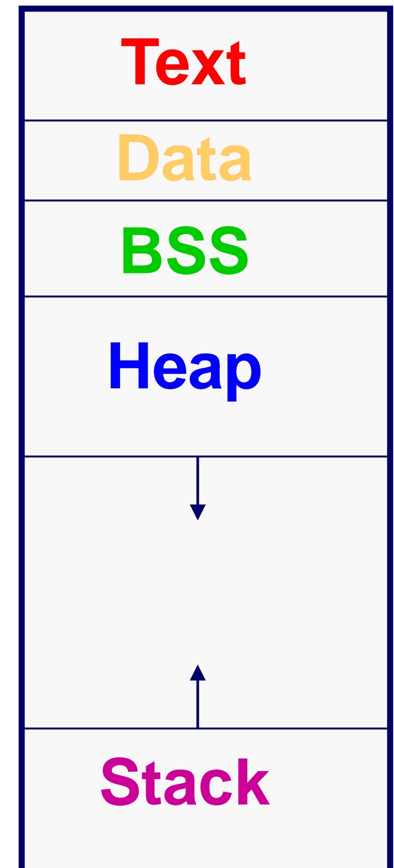
Memory Layout Example: Stack

```
char* string = “hello”;
int iSize;
static char* st_strg = “world”;
static int st_global;
char* f(void)
{
    static char* st_strl = “wow!”;
    static int st_local;
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```



Memory Allocation & Deallocation

- How, and when, is memory allocated?
  - Global and static variables: program startup
  - Local variables: function call
  - Dynamic memory: **`malloc()`**
- How is memory deallocated?
  - Global and static variables: program finish
  - Local variables: function return
  - Dynamic memory: **`free()`**
- All memory deallocated when program ends
  - It is good style to free allocated memory anyway



Memory Allocation Example

```
char* string = "hello";     ← Data: "hello" at startup
int iSize;                  ← BSS: 0 at startup

char* f(void)
{
    char* p;                ← Stack: at function call
    iSize = 8;
    p = malloc(iSize);      ← Heap: 8 bytes at malloc
    return p;
}
```

Memory Deallocation Example

```
char* string = “hello”;
int iSize;

char* f(void)
{
     char* p;
     iSize = 8;
     p = malloc(iSize);
     return p;
}
```

- `char* string = “hello”;` ← Available till termination
- `int iSize;` ← Available till termination
- `char* p;` ← Deallocate on return from function
- `p = malloc(iSize);` ← Deallocate on free()

Memory Initialization

- Local variables have undefined values

```
int count;
```

- Memory allocated by **`malloc()`** has undefined values

```
char* p = (char *) malloc(8);
```

- Must explicitly initialize if you want a particular initial value

```
int count = 0;
p[0] = '\0';
```

- Global and static variables are initialized to 0 by default

```
static int count = 0;
```

is the same as

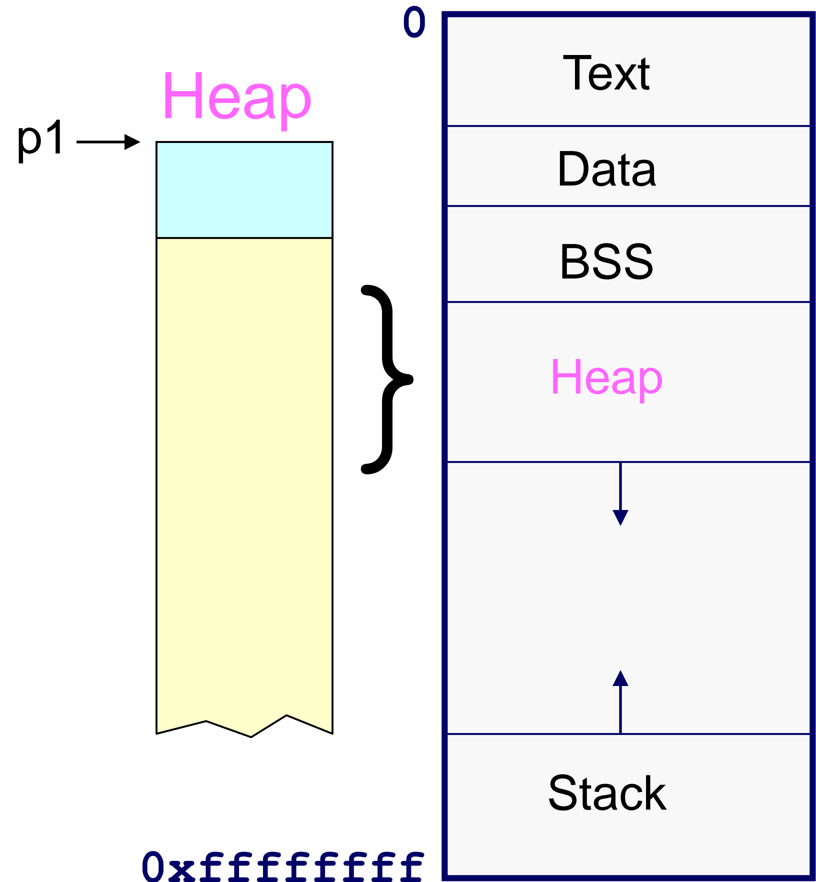
```
static int count;
```

It is bad style to depend on this

Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

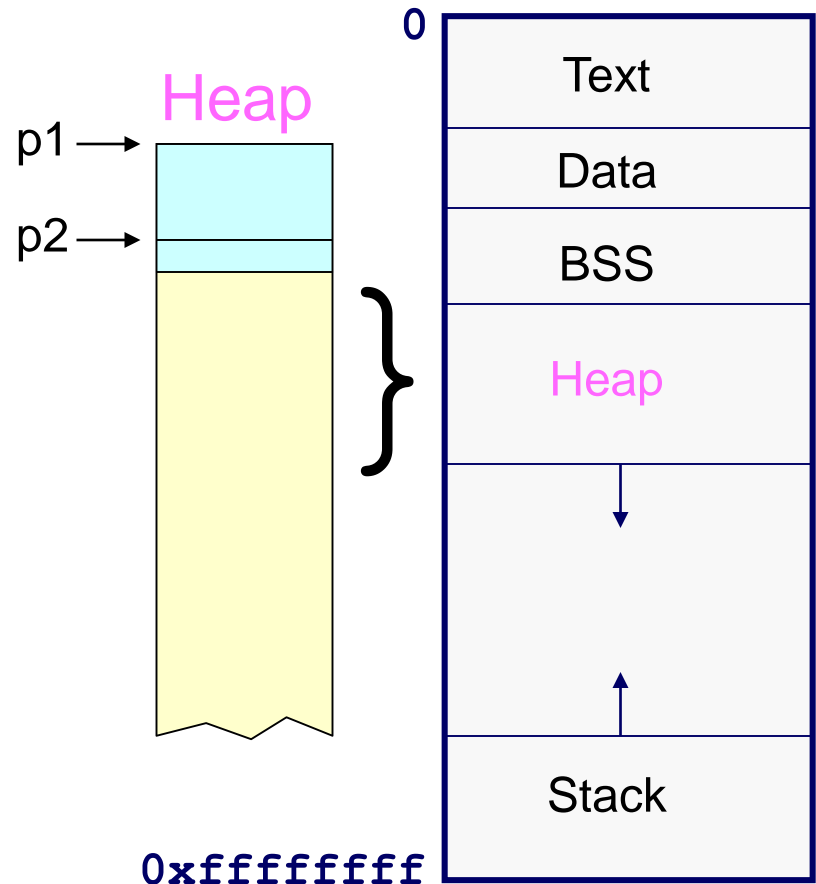
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

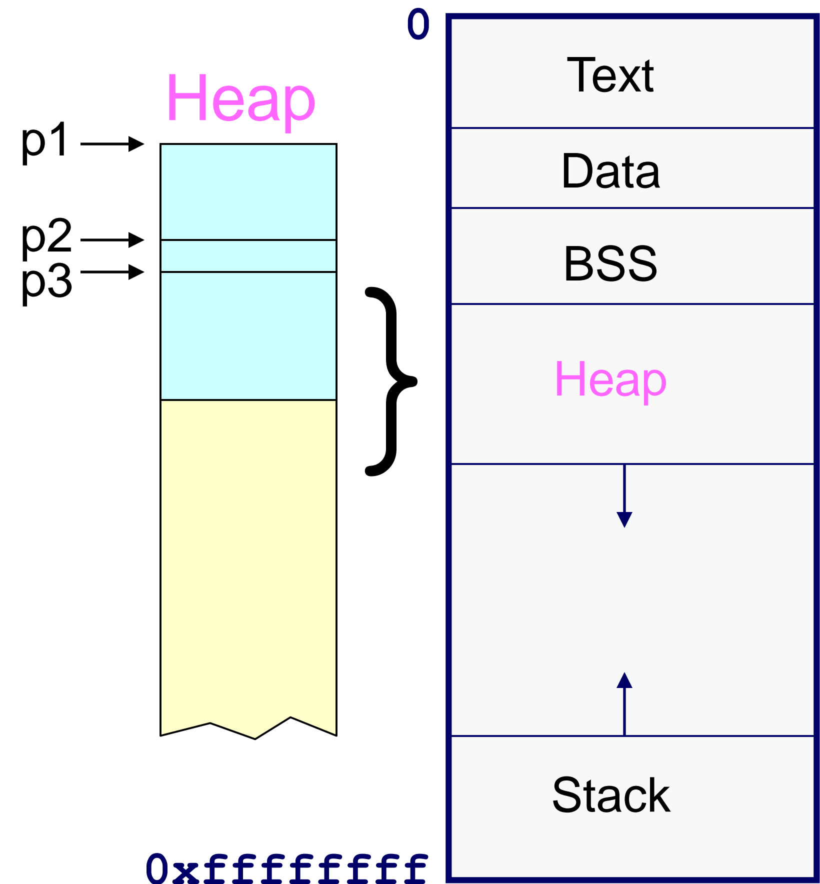
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

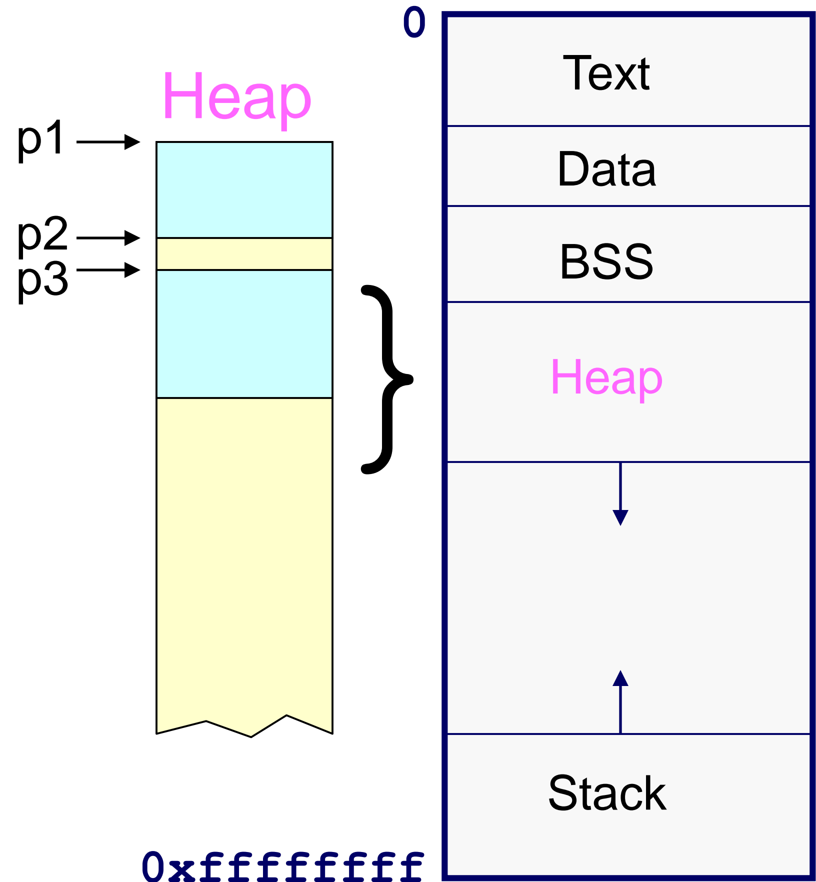
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

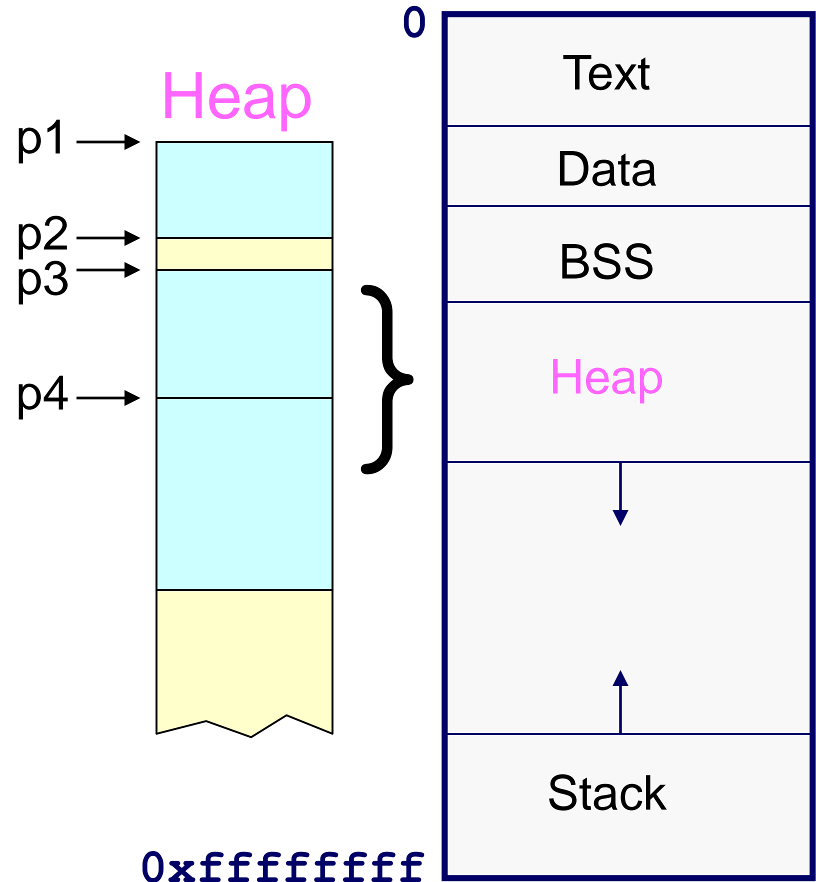
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

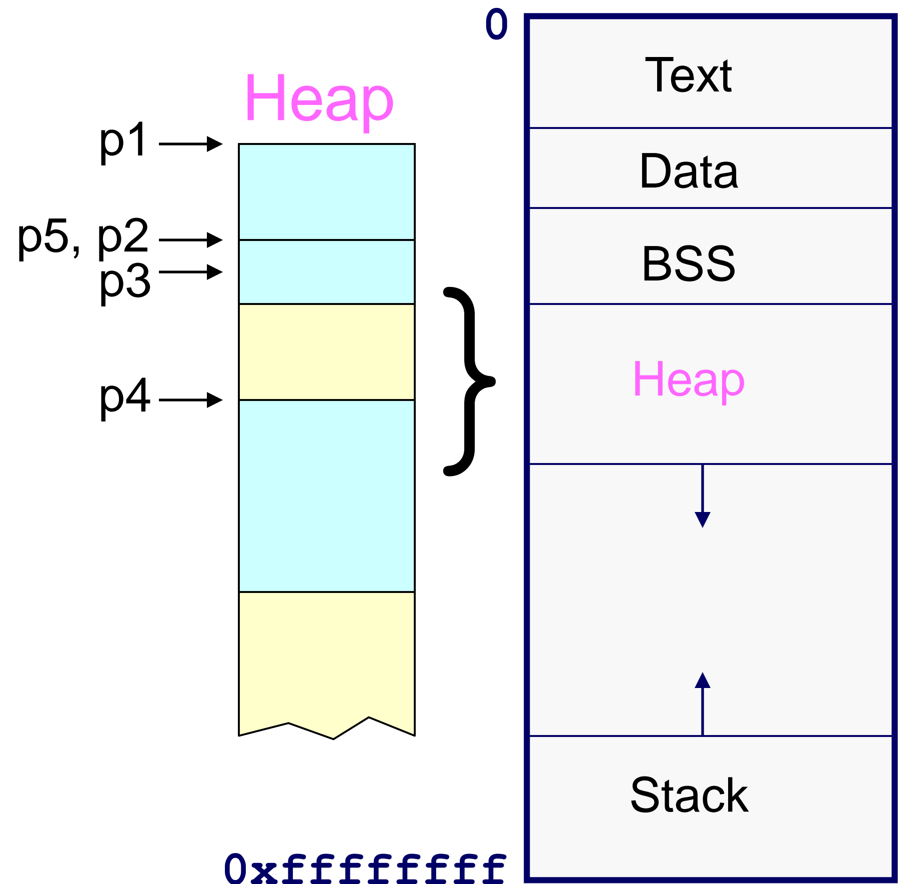
```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

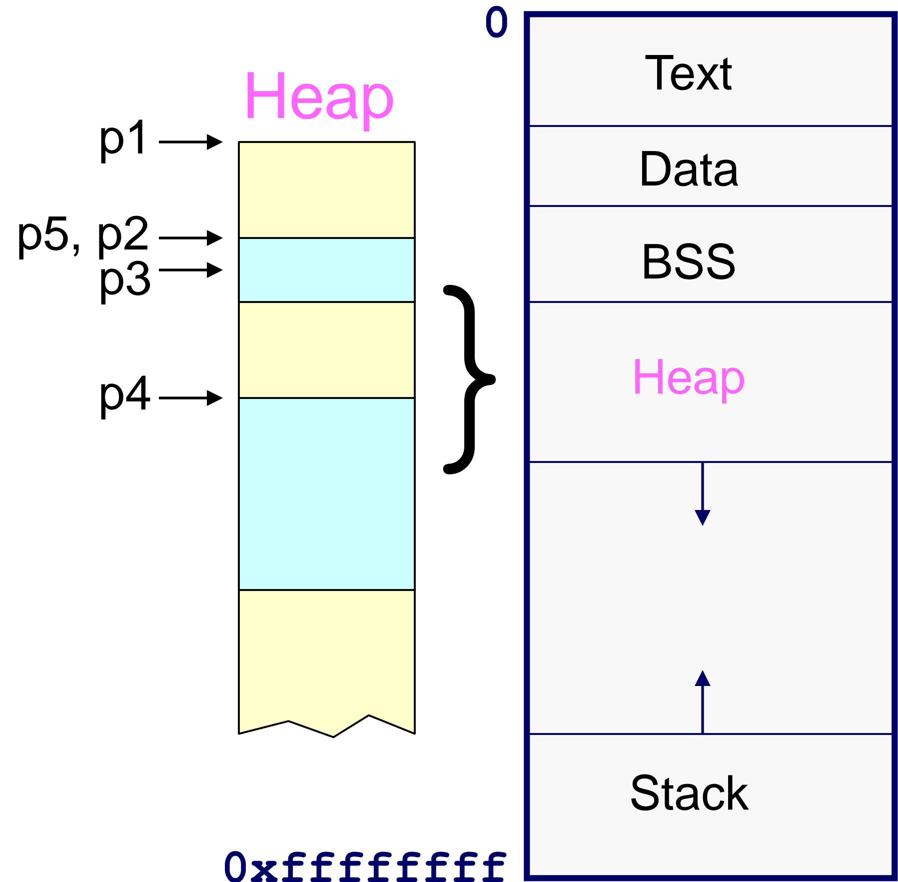
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

Heap

p1

p5, p2

p3

p4

0

Text

Data

BSS

Heap

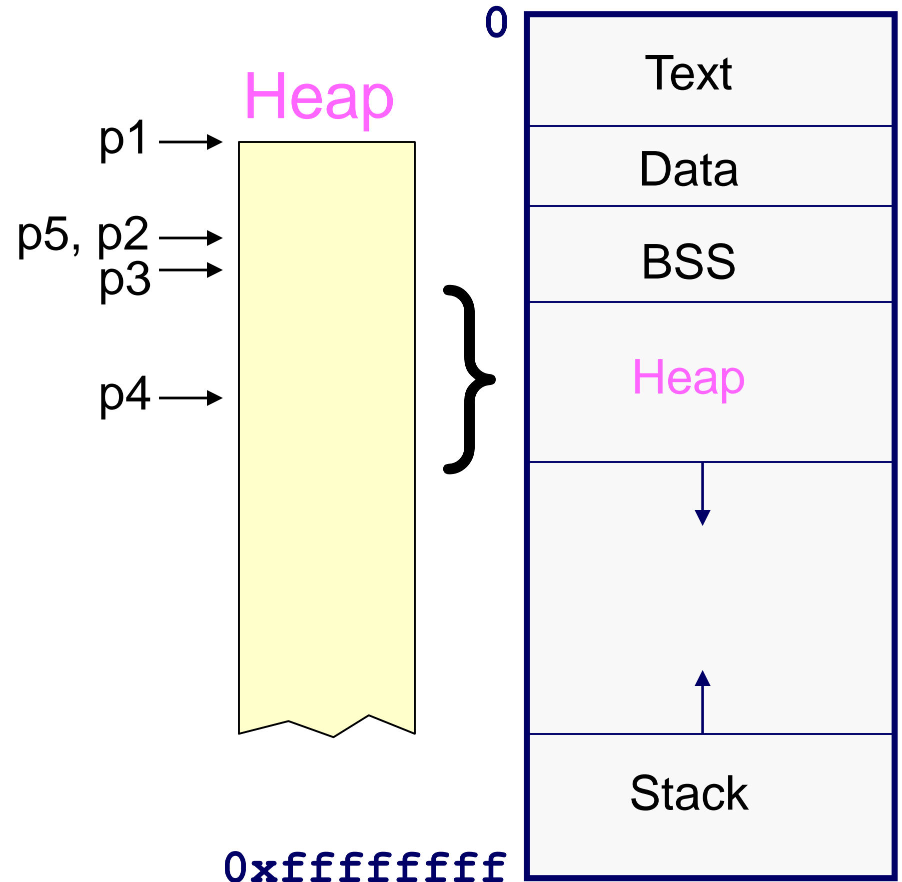
Stack

0xffffffff

Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Summary

- Five types of memory for variables
  - Text: code, constant data (constant data in rodata on hats)
  - Data: initialized global & static variables
  - BSS: uninitialized global & static variables
  - Heap: dynamic memory
  - Stack: local variables
- Important to understand differences between
  - Allocation: space allocated
  - Initialization: initial value, if any
  - Deallocation: space reclaimed
- Understanding memory allocation is important
  - Make efficient use of memory
  - Avoid “memory leaks” from dangling pointers