C Programming - Deck 19

Stack vs Heap Memory

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

Memory Layout of C Program

- **Text Segment**: Program code (instructions)
- **Data Segment**: Global and static variables
- **Heap**: Dynamic memory (grows upward)
- **Stack**: Local variables, function calls (grows downward)
- Stack and heap grow towards each other
- Stack overflow: when stack grows too large
- Heap exhaustion: when heap runs out of memory
- Understanding this is crucial for efficient programming

Stack Memory

- Automatic allocation and deallocation
- LIFO (Last In First Out) structure
- Stores local variables and function parameters
- Stores return addresses for function calls
- Fast access (CPU manages it)
- Limited size (typically 1-8 MB)
- Memory freed automatically when function returns
- Cannot access after function returns

Heap Memory

- Manual allocation using malloc/calloc
- Manual deallocation using free
- No specific order (fragmented)
- Slower access than stack
- Much larger than stack (limited by RAM)
- Memory persists until explicitly freed
- Can access across function boundaries
- Requires programmer discipline

Stack vs Heap Comparison

Aspect	Stack	Heap
Allocation	Automatic	Manual (malloc/calloc)
Deallocation	Automatic	Manual (free)
Speed	Fast	Slower
Size	Small (1-8 MB)	Large (RAM limit)
Access	LIFO	Random
Lifetime	Function scope	Until freed
Fragmentation	No	Yes
Overflow	Stack overflow	Heap exhaustion
Management	Compiler	Programmer

Program 1: Stack Variable Scope

```c
#include <stdio.h>
void func() {
  int x = 10;
  printf("Inside func: x = %d\n", x);
  printf("Address: %p\n", (void*)&x);
}
int main() {
  func();
  printf("Back in main\n");
  return 0;
}
```

Output:

```
Inside func: x = 10
Address: 0x7ffeeb3c4a1c
Back in main
```

x is destroyed when func returns

Program 2: Returning Local Variable Address (WRONG)

```c
#include <stdio.h>
int* func() {
  int x = 42;
  printf("Inside func: %d at %p\n",
    x, (void*)&x);
  return &x;
}
int main() {
  int *ptr = func();
  printf("In main: %d (undefined!)\n", *ptr);
  return 0;
}
```

Output:

```
Inside func: 42 at 0x7ffeeb3c4a1c
In main: 42 (undefined!)
```

Dangling pointer - x destroyed after return

Warning: Function returns address of local variable

Program 3: Heap Memory Persists Across Functions

```c
#include <stdio.h>
#include <stdlib.h>
int* func() {
  int *ptr = (int*)malloc(sizeof(int));
  *ptr = 42;
  printf("Inside func: %d at %p\n",
    *ptr, (void*)ptr);
  return ptr;
}
int main() {
  int *p = func();
  printf("In main: %d at %p\n", *p, (void*)p);
  printf("Heap memory persists!\n");
  free(p);
  return 0;
}
```

Output:

```
Inside func: 42 at 0x7f9a8c405820
In main: 42 at 0x7f9a8c405820
Heap memory persists!
```

Heap memory survives function return

Program 4: Stack Frame - Function Call

```c
#include <stdio.h>
void funcC() {
  int c = 30;
  printf("C: c=%d at %p\n", c, (void*)&c);
}
void funcB() {
  int b = 20;
  printf("B: b=%d at %p\n", b, (void*)&b);
  funcC();
  printf("Back in B\n");
}
void funcA() {
  int a = 10;
  printf("A: a=%d at %p\n", a, (void*)&a);
  funcB();
  printf("Back in A\n");
}
int main() {
  funcA();
  return 0;
}
```

Output:

```
A: a=10 at 0x7ffeeb3c4a2c
B: b=20 at 0x7ffeeb3c4a0c
C: c=30 at 0x7ffeeb3c49ec
Back in B
Back in A
```

Stack grows downward with each call

Program 5: Stack Variable Lifetime

```c
#include <stdio.h>
void func() {
  int x = 10;
  printf("Call 1: x = %d at %p\n",
    x, (void*)&x);
}
int main() {
  func();
  func();
  func();
  printf("Each call gets fresh stack space\n");
  return 0;
}
```

Output:

```
Call 1: x = 10 at 0x7ffeeb3c4a1c
Call 1: x = 10 at 0x7ffeeb3c4a1c
Call 1: x = 10 at 0x7ffeeb3c4a1c
Each call gets fresh stack space
```

Same address reused each call

Program 6: Static vs Stack Variables

```c
#include <stdio.h>
void func() {
  int stack_var = 0;
  static int static_var = 0;
  stack_var++;
  static_var++;
  printf("Stack: %d, Static: %d\n",
    stack_var, static_var);
}
int main() {
  func();
  func();
  func();
  return 0;
}
```

Output:

```
Stack: 1, Static: 1
Stack: 1, Static: 2
Stack: 1, Static: 3
```

Static persists, stack resets each call

Program 7: Large Stack Allocation

```c
#include <stdio.h>
void func() {
  int arr[100000];
  arr[0] = 1;
  arr[99999] = 100000;
  printf("Large stack array created\n");
  printf("First: %d, Last: %d\n",
    arr[0], arr[99999]);
  printf("Size: %lu bytes\n",
    sizeof(arr));
}
int main() {
  printf("Creating large stack array\n");
  func();
  printf("Array destroyed on return\n");
  return 0;
}
```

Output:

```
Creating large stack array
Large stack array created
First: 1, Last: 100000
Size: 400000 bytes
Array destroyed on return
```

Large arrays can cause stack overflow

Program 8: Heap Memory Example

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *heap_var;
  int stack_var = 10;
  heap_var = (int*)malloc(sizeof(int));
  *heap_var = 20;
  printf("Stack var: %d at %p\n",
    stack_var, (void*)&stack_var);
  printf("Heap var: %d at %p\n",
    *heap_var, (void*)heap_var);
  printf("Pointer itself at: %p\n",
    (void*)&heap_var);
  free(heap_var);
  return 0;
}
```

Output:

```
Stack var: 10 at 0x7ffeeb3c4a1c
Heap var: 20 at 0x7f9a8c405820
Pointer itself at: 0x7ffeeb3c4a10
```

Different address ranges for stack/heap

Program 9: Stack Overflow Example

```c
#include <stdio.h>
int count = 0;
void recursive() {
  int x;
  count++;
  if (count % 10000 == 0) {
    printf("Depth: %d, addr: %p\n",
      count, (void*)&x);
  }
  recursive();
}
int main() {
  printf("Starting infinite recursion\n");
  recursive();
  return 0;
}
```

Warning: This will crash with stack overflow

Output:

```
Starting infinite recursion
Depth: 10000, addr: 0x7ffee5bc1a0c
Depth: 20000, addr: 0x7ffee5741a0c
...
Segmentation fault (stack overflow)
```

Stack has limited size

Program 10: Heap vs Stack - String Storage

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
  char stack_str[20] = "Stack String";
  char *heap_str;
  heap_str = (char*)malloc(20 * sizeof(char));
  strcpy(heap_str, "Heap String");
  printf("Stack: %s at %p\n",
    stack_str, (void*)stack_str);
  printf("Heap: %s at %p\n",
    heap_str, (void*)heap_str);
  free(heap_str);
  return 0;
}
```

Output:

```
Stack: Stack String at 0x7ffeeb3c4a00
Heap: Heap String at 0x7f9a8c405820
```

Strings can be on stack or heap

Program 11: Multiple Heap Allocations

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *p1, *p2, *p3;
  p1 = (int*)malloc(sizeof(int));
  p2 = (int*)malloc(sizeof(int));
  p3 = (int*)malloc(sizeof(int));
  *p1 = 10; *p2 = 20; *p3 = 30;
  printf("p1: %d at %p\n", *p1, (void*)p1);
  printf("p2: %d at %p\n", *p2, (void*)p2);
  printf("p3: %d at %p\n", *p3, (void*)p3);
  printf("Heap not contiguous\n");
  free(p1); free(p2); free(p3);
  return 0;
}
```

Output:

```
p1: 10 at 0x7f9a8c405820
p2: 20 at 0x7f9a8c405840
p3: 30 at 0x7f9a8c405860
Heap not contiguous
```

Heap allocations may be scattered

Program 12: Stack Array vs Heap Array

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int stack_arr[5] = {1, 2, 3, 4, 5};
  int *heap_arr;
  int i;
  heap_arr = (int*)malloc(5 * sizeof(int));
  for (i = 0; i < 5; i++) {
    heap_arr[i] = (i + 1) * 10;
  }
  printf("Stack array at: %p\n",
    (void*)stack_arr);
  printf("Heap array at: %p\n",
    (void*)heap_arr);
  printf("Stack: ");
  for (i=0; i<5; i++) printf("%d ", stack_arr[i]);
  printf("\nHeap: ");
  for (i=0; i<5; i++) printf("%d ", heap_arr[i]);
  printf("\n");
  free(heap_arr);
  return 0;
}
```

Output:

```
Stack array at: 0x7ffeeb3c4a00
Heap array at: 0x7f9a8c405820
Stack: 1 2 3 4 5
Heap: 10 20 30 40 50
```

Arrays can be on stack or heap

Program 13: Structure on Stack vs Heap

```c
#include <stdio.h>
#include <stdlib.h>
struct Point {
  int x;
  int y;
};
int main() {
  struct Point stack_pt = {10, 20};
  struct Point *heap_pt;
  heap_pt = (struct Point*)malloc(
    sizeof(struct Point));
  heap_pt->x = 30;
  heap_pt->y = 40;
  printf("Stack: (%d,%d) at %p\n",
    stack_pt.x, stack_pt.y, (void*)&stack_pt);
  printf("Heap: (%d,%d) at %p\n",
    heap_pt->x, heap_pt->y, (void*)heap_pt);
  free(heap_pt);
  return 0;
}
```

Output:

```
Stack: (10,20) at 0x7ffeeb3c4a10
Heap: (30,40) at 0x7f9a8c405820
```

Structures can be on stack or heap

Program 14: Memory Address Comparison

```c
#include <stdio.h>
#include <stdlib.h>
int global_var = 100;
int main() {
  int stack_var = 10;
  int *heap_var = (int*)malloc(sizeof(int));
  static int static_var = 50;
  *heap_var = 20;
  printf("Global: %p\n", (void*)&global_var);
  printf("Static: %p\n", (void*)&static_var);
  printf("Stack: %p\n", (void*)&stack_var);
  printf("Heap: %p\n", (void*)heap_var);
  printf("\nDifferent memory regions\n");
  free(heap_var);
  return 0;
}
```

Output:

```
Global: 0x10a8e4020
Static: 0x10a8e4024
Stack: 0x7ffeeb3c4a1c
Heap: 0x7f9a8c405820

Different memory regions
```

Each segment has distinct address range

Program 15: Function Parameter Passing

```c
#include <stdio.h>
void func(int x, int *p) {
  printf("In func:\n");
  printf("  x (value): %d at %p\n",
    x, (void*)&x);
  printf("
p (pointer): %p\n", (void*)p);
  printf("  *p (value): %d\n", *p);
}
int main() {
  int a = 10;
  int b = 20;
  printf("In main:\n");
  printf("
a: %d at %p\n", a, (void*)&a);
  printf("
b: %d at %p\n", b, (void*)&b);
  func(a, &b);
  return 0;
}
```

Output:

```
In main:
  a: 10 at 0x7ffeeb3c4a1c
  b: 20 at 0x7ffeeb3c4a18
In func:
  x (value): 10 at 0x7ffeeb3c49fc
  p (pointer): 0x7ffeeb3c4a18
  *p (value): 20
```

Parameters copied to function's stack frame

Program 16: Heap Fragmentation

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *p1, *p2, *p3, *p4;
  p1 = (int*)malloc(sizeof(int));
  p2 = (int*)malloc(sizeof(int));
  p3 = (int*)malloc(sizeof(int));
  printf("Initial: %p %p %p\n",
    (void*)p1, (void*)p2, (void*)p3);
  free(p2);
  printf("After freeing p2\n");
  p4 = (int*)malloc(sizeof(int));
  printf("New p4: %p\n", (void*)p4);
  printf("p4 may reuse p2's space\n");
  free(p1); free(p3); free(p4);
  return 0;
}
```

Output:

```
Initial: 0x7f9a8c405820
  0x7f9a8c405840
  0x7f9a8c405860
After freeing p2
New p4: 0x7f9a8c405840
p4 may reuse p2's space
```

Heap memory can be fragmented and reused

Program 17: Dangling Pointer After Function Return

```c
#include <stdio.h>
int* getStackAddress() {
  int local = 42;
  return &local;
}
int* getHeapAddress() {
  int *ptr = (int*)malloc(sizeof(int));
  *ptr = 42;
  return ptr;
}
int main() {
  int *bad = getStackAddress();
  int *good = getHeapAddress();
  printf("Stack (BAD): %d (undefined)\n", *bad);
  printf("Heap (GOOD): %d\n", *good);
  free(good);
  return 0;
}
```

Output:

```
Stack (BAD): 42 (undefined)
Heap (GOOD): 42
```

Heap persists, stack doesn't

Program 18: Stack Size Demonstration

```c
#include <stdio.h>
void printStackUsage(int depth) {
  int x;
  static int *first = NULL;
  if (first == NULL) {
    first = &x;
  }
  if (depth % 1000 == 0) {
    printf("Depth %d: %ld bytes from start\n",
      depth, (char*)first - (char*)&x);
  }
  if (depth < 10000) {
    printStackUsage(depth + 1);
  }
}
int main() {
  printf("Measuring stack growth\n");
  printStackUsage(0);
  return 0;
}
```

Output:

```
Measuring stack growth
Depth 0: 0 bytes from start
Depth 1000: 32000 bytes from start
Depth 2000: 64000 bytes from start
...
Depth 10000: 320000 bytes from start
```

Stack grows with recursion depth

Program 19: Choosing Stack vs Heap

```c
#include <stdio.h>
#include <stdlib.h>
void processSmallData() {
  int data[10];
  int i;
  for (i = 0; i < 10; i++) data[i] = i;
  printf("Small: Use stack\n");
}
void processLargeData() {
  int *data = (int*)malloc(1000000*sizeof(int));
  int i;
  for (i = 0; i < 1000000; i++) data[i] = i;
  printf("Large: Use heap\n");
  free(data);
}
int main() {
  processSmallData();
  processLargeData();
  printf("Choose based on size and lifetime\n");
  return 0;
}
```

Output:

```
Small: Use stack
Large: Use heap
Choose based on size and lifetime
```

Stack for small/short, heap for large/long

Program 20: Complete Stack vs Heap Example

```c
#include <stdio.h>
#include <stdlib.h>
struct Data {
  int value;
  struct Data *next;
};
void stackList() {
  struct Data n1 = {1, NULL};
  struct Data n2 = {2, &n1};
  struct Data n3 = {3, &n2};
  printf("Stack list: destroyed on return\n");
}
struct Data* heapList() {
  struct Data *n1, *n2, *n3;
  n1=(struct Data*)malloc(sizeof(struct Data));
  n2=(struct Data*)malloc(sizeof(struct Data));
  n3=(struct Data*)malloc(sizeof(struct Data));
  n1->value=1; n1->next=NULL;
  n2->value=2; n2->next=n1;
  n3->value=3; n3->next=n2;
  return n3;
}
int main() {
  stackList();
  struct Data *list = heapList();
  printf("Heap list: persists, must free\n");
  while(list){struct Data *t=list;list=list->next;free(t);}
  return 0;
}
```

Output:

```
Stack list: destroyed on return
Heap list: persists, must free
```

Stack for temporary, heap for persistent data

When to Use Stack vs Heap

- **Use Stack when:**
  - Data size is small and known at compile time
  - Data lifetime matches function scope
  - Need fast allocation/deallocation
  - Don't need to return data from function
- **Use Heap when:**
  - Data size is large or unknown at compile time
  - Data must persist beyond function scope
  - Need to return data from function
  - Building dynamic data structures (linked lists, trees)
  - Size determined at runtime

Key Takeaways

- Stack: automatic, LIFO, fast, limited size
- Heap: manual, flexible, slower, large size
- Stack variables destroyed when function returns
- Heap memory persists until explicitly freed
- Never return address of local (stack) variable
- Stack overflow from deep recursion or large locals
- Heap exhaustion from too many allocations
- Choose based on size, lifetime, and scope needs
- Understanding this prevents common bugs
- Critical for efficient memory management