Lecture 16

Dynamic Memory Allocation

Shibo Li

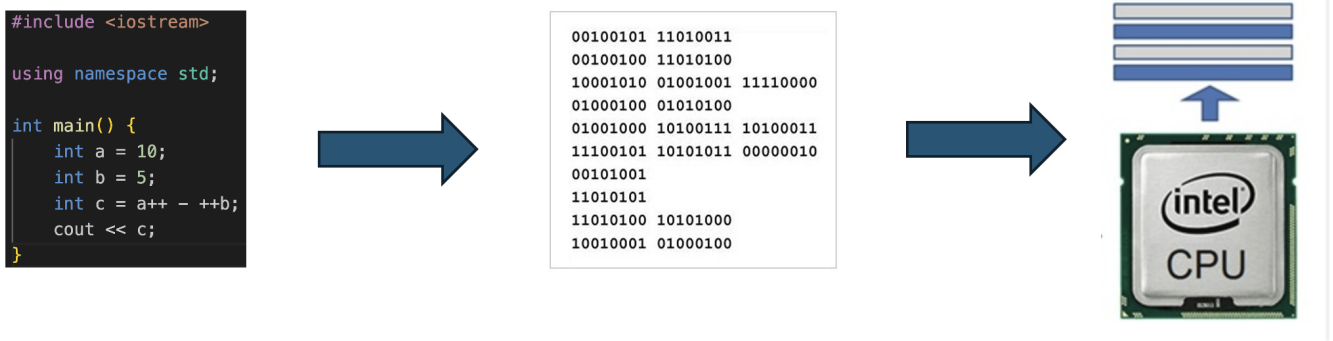
shiboli@cs.fsu.edu

Department of Computer Science
Florida State University

The slides are mainly from Sharanya Jayaraman

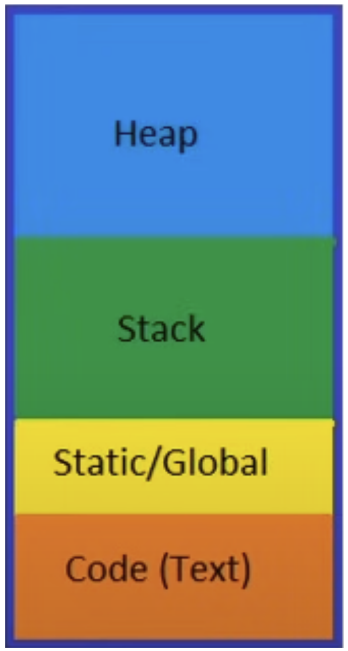
Program vs. Prcoess

- **Program:** This is a static set of instructions written in a language like C++ that is stored on disk.
- **Process:** Once a program is loaded and executed, it becomes a process. A process is an active instance of a program that runs in memory



Memory Allocation

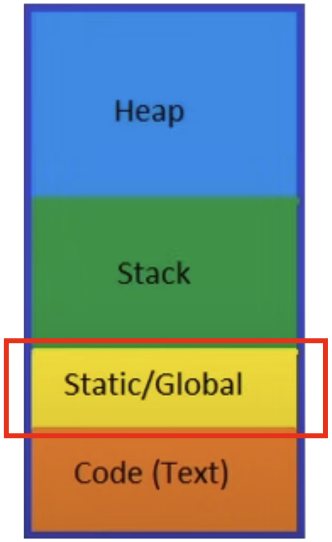
- crucial for **process** execution
- assigning specific memory areas
- data, variables, and structures



Allocating Memory

Static/Compile-Time Allocation: variables or constants before the program runs

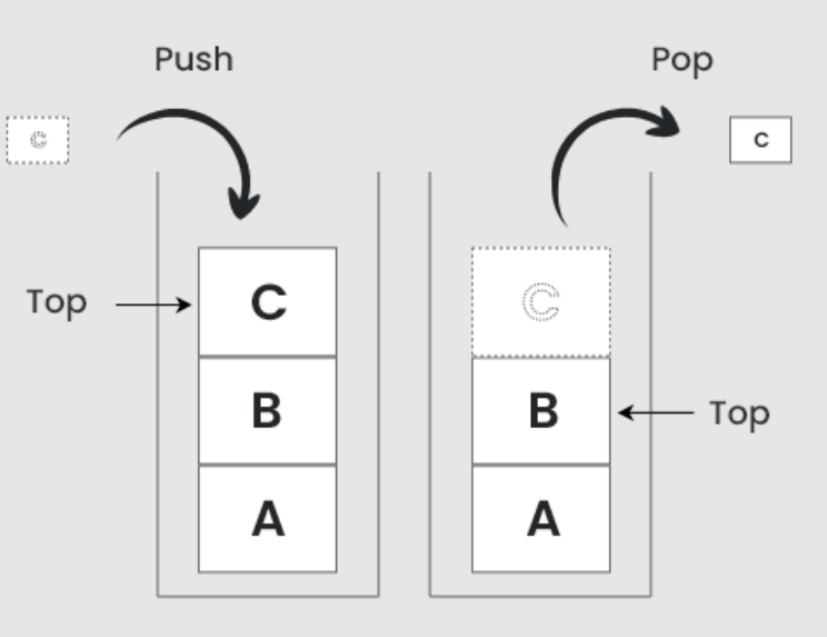
- **Global Variables**: Variables declared outside of any function are stored in the global memory segmen
- **Static Variables**: `static`
- **Constants**: `#define`, `const`

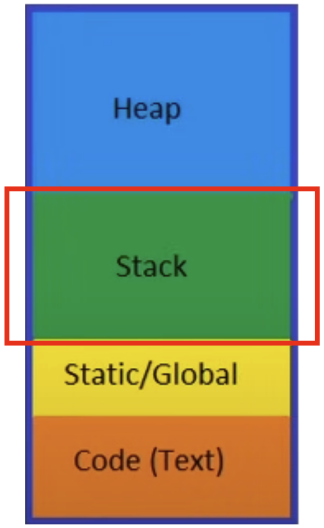


Allocating Memory

Automatic Allocation: Memory is allocated automatically on the stack

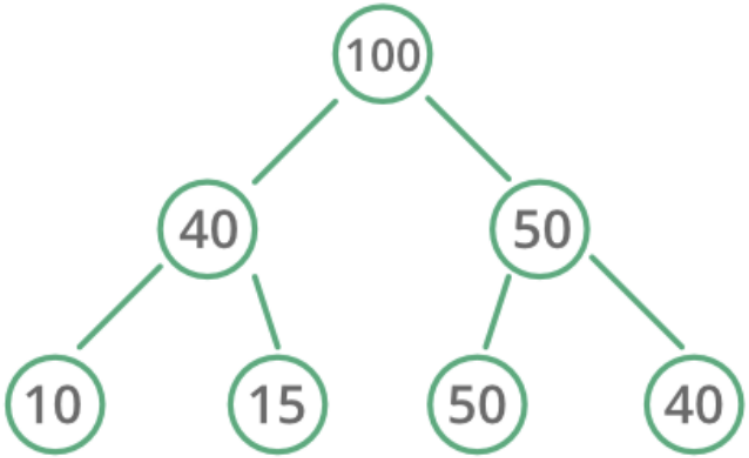
- Last in first out (LIFO)
- local variables within functions
- The stack is managed by the program's flow

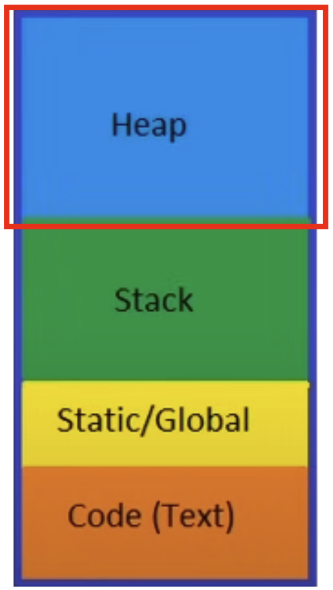




Dynamics Allocation:

- Memory allocated "on the fly" during **run time**
- Memory is managed in the heap
- Allowing for flexible memory usage but requiring explicit allocation and deallocation to avoid memory leaks





Example: Static Allocation

- Occurs at compile-time.
- Memory is allocated for global variables, static variables, and constants, and remains in place for the entire duration of the program.

```
static int count = 10;
```

Example: Automatic/Stack Allocation

- Occurs at runtime, automatically on the stack.
- Typically used for local variables within functions.
- Memory is allocated when the variable is created and automatically freed when it goes out of scope.

```
void example() {
    int number = 42; // automatically allocated on
        the stack
}
```

Example: Dynamic Allocation

- Managed explicitly by the programmer using new and delete.
- Allows for flexible, runtime memory allocation, where the size and duration can vary.
- Deal for large or unknown amounts of memory that may need to persist beyond a single function scope.

```
// dynamically allocate memory on the heap
int* ptr = new int(5);
// free memory when done
delete ptr;
```

Dynamic De-allocation

Memory deallocation in C++ is the process of freeing up dynamically allocated memory that is no longer needed.

- Static/Stack handles deallocation automatically
- Heap handles deallocation mauanlly **by the programmer**
- If memory is not deallocated after it's no longer needed, it can lead to a **memory leak**.
- Memory that cannot be reclaimed or reused by the system, reducing the available memory for other processes

Dynamic Memory Allocation

We can dynamically allocate storage space while the programis running, but we cannot create new variable names “on the fly”

- Creating the dynamic space.
- Storing its address in a **pointer** (so that the space can be accesed)

Allocating space with new

- To allocate space dynamic ally, use the unary operator **new**, followed by the type being allocated.

```
new int; // dynamically allocates an int
new double; // dynamically allocates a double
```

Allocating space with new

- If creating an array dynamically, use the same form, but put brackets with a size after the type:

```
// dynamically allocates an array of 40 ints
new int[40];
// dynamically allocates an array of size doubles
// note that the size can be a variable
new double[size];
```

- These statements above are not very useful by themselves, because the allocated spaces have no names!

Allocating space with new

- The new operator returns the starting address of the allocated space, and this address can be stored in a pointer:

```
// declare a pointer p
// dynamically allocate an int and load address into p
int * p;
p = new int;
// declare a pointer d
// dynamically allocate a double and load address
    into d
double * d;
d = new double;
```

Allocating space with new

```
// we can also do these in single line statements
int x = 40;
int * list = new int[x];
float * numbers = new float[x+10];
```

- Notice that this is one more way of initializing a pointer to a valid target (and the most important one).

Allocating space with new

So once the space has been dynamically allocated, how do weuse it?

- For single items, we go through the pointer. Dereference the pointer to reach the dynamically created target:

```
// dynamic integer, pointed to by p
int * p = new int;
// assigns 10 to the dynamic integer
*p = 10;
// prints 10
cout << *p;
```

Allocating space with new

So once the space has been dynamically allocated, how do weuse it?

- For dynamically created arrays, you can use either pointer-offset notation, or treat the pointer as the array name and use the standard bracket notation:

```
double * numList = new double[size];
for (int i = 0; i < size; i++)
   numList[i] = 0; // initialize elements to 0
numList[5] = 20; // bracket notation
*(numList + 7) = 15; // pointer-offset notation
```

Deallocation of dynamic memory

- To deallocate memory that was created with `new`, we use the unary operator `delete`.
- The one operand should be a pointer that stores the address of the space to be deallocated:

```
int * ptr = new int; // dynamically created int
// ...
delete ptr; // deletes the space that ptr points to
ptr = nullptr;
```

- **Dangling Pointers**: Not setting pointers to nullptr after deletion can lead to accessing freed memory.

Deallocation of dynamic memory

- To deallocate memory that was created with `new`, we use the unary operator `delete`.
- The one operand should be a pointer that stores the address of the space to be deallocated:

```
int * ptr = new int; // dynamically created
    int
// ...
delete ptr; // deletes the space that ptr
    points to
ptr = nullptr;
```

- **Dangling Pointers**: Not setting pointers to nullptr after deletion can lead to accessing freed memory.

Deallocation of dynamic memory

- Note that the pointer ptr still exists in this example. That's a named variable subject to scope and extent determined at compile time. It can be reused:

```
ptr = new int[10]; // point p to a brand new array
```

Deallocation of dynamic memory

- To deallocate a dynamic array, use this form `delete []`

```
int* arr = new int[5]; // Allocates memory for an
    array of 5 integers

for (int i = 0; i < 5; i++) {
   arr[i] = i + 1; // Assign values to each element
}

delete[] arr; // Deallocates the array memory
arr = nullptr; // Set to nullptr to avoid dangling
```

Application Example: Dynamic Resizing Array

- If you have an existing array, and you want to make it bigger (add array cells to it), you cannot simply append new cells to the old ones.
- Remember that arrays are stored in consecutive memory, and you never know whether or not the memory immediately after the array is already allocated for something else.
- For that reason, the process takes a few more steps.

Application Example: Dynamic Resizing Array

- Here is an example using an integer array. Let's say this is the original array:

```
int * list = new int[size];
```

- I want to resize this so that the array called list has space for5 more numbers (presumably because the old one is full).

Application Example: Dynamic Resizing Array

There are four main steps.

- Create an entirely new array of the appropriate type and of the new size. (You'll need another pointer for this).

```
int * temp = new int[size + 5];
```

- Copy the data from the old array into the new array (keeping them in the same positions). This is easy with a for-loop.

```
for (int i = 0; i < size; i++)
   temp[i] = list[i];
```

Application Example: Dynamic Resizing Array

There are four main steps.

- Delete the old array – you don't need it anymore!

```
// this deletes the array pointed to by "list"
delete [] list;
```

- Change the pointer. You still want the array to be called "list" (its original name), so change the list pointer to the new address.

```
list = temp;
```

Application Example: Multidimensional Arrays

Creating dynamically sized matrices. Dynamic 2D arrays are useful for applications like image processing or scientific computations where matrix dimensions are determined at runtime.

```
int rows = 3, cols = 4;
int** matrix = new int*[rows];
for (int i = 0; i < rows; i++) {
   matrix[i] = new int[cols];
}
// Fill and use matrix
for (int i = 0; i < rows; i++) {
   delete[] matrix[i];
}
delete[] matrix;
```

Application Example: Text Processing

Handling text in applications where strings can vary significantly in size (e.g., text editors).

```
char* text = new char[256]; // initial allocation
// ...
// Expand text buffer if needed by allocating new space
// ...
delete[] text; // free when done
```

Application Example: Advanced Data Structures

Trees provide a hierarchical structure for data storage, such as organizing files or databases.

```
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int val) : data(val), left(nullptr),
        right(nullptr) {}
};

TreeNode* root = new TreeNode(10);
root->left = new TreeNode(5);
root->right = new TreeNode(15);
```