CS-1201 Object Oriented Programming

Memory Allocation

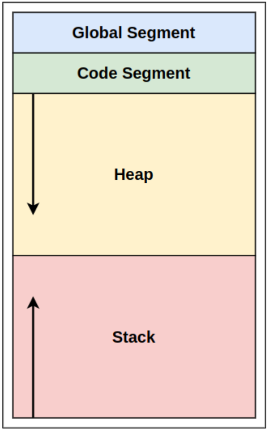
Arbish Akram

Department of Computer Science
Government College University

Memory Segments

- Memory is generally divided into the following segments:
  - **Global Segment**
  - **Code Segment**
  - **Stack Segment**
  - **Heap Segment**
- Each segment serves a specific purpose in program execution.

Memory Segments



Global and Code Segments

Global Segment

- Stores global and static variables.
- These variables have a lifetime equal to the entire duration of the program's execution.
- Essential for data that needs to be accessible throughout the program.

Code Segment

- Also known as the **text segment**.
- Contains the actual machine code or instructions that comprise the program.
- Includes functions and methods that define the program's behavior.

Stack and Heap Segments

Stack Segment

- Used for managing local variables, function arguments, and control information.
- Handles return addresses for function calls.
- Operates on a last-in, first-out (LIFO) principle.

Heap Segment

- Also known as **dynamic memory**.
- Allows allocation and deallocation of memory at runtime.
- Ideal for storing large data structures or objects with unknown sizes.

Dynamic Memory Allocation

- Control the allocation and deallocation of memory in a program.
- Applicable for objects and arrays of built-in or user-defined types.
- Performed using:
  - `new` - allocates memory.
  - `delete` - deallocates memory.

Using the `new` Operator

- `new` dynamically allocates memory required to hold an object or array at runtime.
- Memory is allocated in the **free store** (or **heap**).
- Access the allocated memory using the pointer returned by `new`.
- `Time *timePtr = new Time;`
- If memory allocation fails, `new` throws an exception.

Using the `delete` Operator

- `delete` deallocates memory and returns it to the free store.
- To release a dynamically allocated object:
- `delete timePtr;`
- Calls the destructor for the object, then releases the memory.

Memory Leaks

- Not releasing dynamically allocated memory when it's no longer needed can cause the system to run out of memory prematurely.
- This issue is commonly referred to as a **memory leak**.
- Always use `delete` after using `new` to avoid leaks.

Dynamic Memory Allocation: Example

```cpp
#include <iostream>
using namespace std;
int main() {
    // Allocate memory for an integer
    int* intPtr = new int;
    *intPtr = 10;
    delete intPtr; // Deallocate memory

    // Allocate memory for an array
    int* arrPtr = new int[5];
    arrPtr[0] = 1;
    delete[] arrPtr; // Deallocate memory

    return 0;
}
```

Example

```
int main() {
    int size;
    cout << "Enter the size of the array: ";
    cin >> size;
    // Dynamically allocate memory for the array
    //int* arr = new int[size];
    int arr[size];
    // Store the square of each index in the array
    for (int i = 0; i < size; ++i) {
        arr[i] = i * i;
    }
    // Display the values in the array
    cout << "The squares of each index are: ";
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
    //delete[] arr;   // Deallocate the memory
    return 0;
}
```