Chapter 3 (part 2)

array : declaration, initialisation,

namespace,

dynamic allocation

Arrays : declaration, initialisation

- Arrays are data structures in programming that allow you to store a collection of elements of the same data type under a single variable name.

1. **Declaration:**
   To declare an array, you need to specify its data type and name. The array's data type is the type of elements it will store, and the name is used to refer to the array in your code.

Syntax :

data_type array_name[array_size];

data_type: The type of data the array will store
array_name: The name you choose for the array.
array_size: The number of elements the array can hold. This size is determined when the array is created.

Example:

```
int myArray[5];
double prices[10];
char characters[20];.
```

2. Initialization: Initialization can be done in different ways:

a. **Initializing at the time of declaration:** You can set initial values for the array when declaring it.

Example:

```
int numbers[3] = {1, 2, 3};
```

b. Partial initialization: You can partially initialize an array, and the remaining elements will be set to default values (usually 0 for numeric types).

Example:

```
int data[5] = {1, 2}; // Initializes the first two elements to 1 and 2; the rest are set to 0.
```

c. Initializing with a loop: This is useful for populating an array with values calculated at runtime.

Example:

```
int squares[5];
for (int i = 0; i < 5; i++)
{  squares[i] = i * i;}
```

Array Name as a Pointer:
In C and C++, the name of an array is essentially a pointer to the first element of the array. When you use the array name in an expression without an index, it acts as a pointer to the first element.

Example:
int numbers[5] = {1, 2, 3, 4, 5};
int *ptr = numbers; // Assigns the address of the first element to the pointer.

Array Indexing:
You can access individual elements of an array using square brackets and an index. This is equivalent to pointer arithmetic.

Example:
int secondElement = numbers[1]; // Accessing the second element using array indexing. int thirdElement = *(numbers + 2); // Accessing the third element using pointer arithmetic.

Pointer Arithmetic:

Pointers can be used to traverse an array by incrementing or decrementing the pointer. This is more flexible and powerful than array indexing.

Example:

```
int *ptr = numbers;
int thirdElement = *(ptr + 2); // Accessing the third element using pointer
```

Passing Arrays to Functions:

When you pass an array to a function, you're actually passing a pointer to the first element of the array. This is why changes made to the array within the function are reflected outside the function.

Pass by pointer:

Example:

```
void printArray(int* arr, int size)
 {    for (int i = 0; i < size; i++)
          cout << arr[i] << " ";
  cout << endl;
}

int main()
 {    int myArray[] = {1, 2, 3, 4, 5};
      int size = 5;
     printArray(myArray, size);
     return 0;
}
```

Pass by Reference:

You can also pass an array by reference. This allows the function to work with the original array without creating a copy.

Example:

```
void modifyArray(int (&arr)[5])
 {    for (int i = 0; i < 5; i++)
          arr[i] *= 2;  }

int main()
{    int myArray[] = {1, 2, 3, 4, 5};
  modifyArray(myArray);
 for (int i = 0; i < 5; i++)
 {       cout << myArray[i] << " ";    }
  cout << endl;
 return 0; }
```

Pass by std::array or std::vector: **(Works on C++11)**

In modern C++, it's often recommended to use std::array or std::vector from the Standard Library instead of built-in arrays when passing arrays to functions. These container classes offer safer and more convenient ways to work with arrays.

```
#include <iostream>
#include <array>
#include <vector>

void printVector(const std::vector<int>& vec)
 {    for (int value : vec)
 {        cout << value << " ";    }
 cout << endl;}

int main()
 {    std::vector<int> myVector = {1, 2, 3, 4, 5};
  printVector(myVector);
 return 0;}
```

Pass by a dynamic array (using pointers):
When working with dynamically allocated arrays (created with new in C++), you can pass them to functions as pointers. Be cautious when using dynamic arrays to ensure proper memory management (e.g., using delete when you're done with the array).

```
void processDynamicArray(int* arr, int size)
 {    // Do something with the dynamic array
     // Don't forget to deallocate memory when done
   delete[] arr;}

int main()
{    int* dynamicArray = new int[5];
// Initialize dynamicArray
processDynamicArray(dynamicArray, 5);
 return 0;}
```

namespace

- In C++, a namespace is a mechanism that helps prevent naming conflicts and organizes code into logical groups.

- It allows you to define a scope or a context in which identifiers, such as variables, functions, and classes, can exist without conflicting with identifiers of the same name in other namespaces.

- This is particularly useful when you are working with large codebases or integrating multiple libraries, as it helps ensure that the names of your identifiers do not clash with names defined elsewhere.

```
// Define a namespace
namespace my_namespace
{    int my_variable = 42;
 void my_function()
{       // Function code here    }
}

int main()
{    // Accessing variables and functions from a namespace
  int x = my_namespace::my_variable;
  my_namespace::my_function();
  return 0;
}
```

:: is the scope resolution operator.
Opérateur de résolution de portée

Dynamic allocation

- Dynamic allocation in C++ refers to the process of allocating memory for variables or data structures at runtime (during program execution) rather than at compile time.
- This is in contrast to static allocation, where memory is allocated at compile time, and the size of the allocated memory is fixed.
- In C++, dynamic allocation is typically performed using operators new and delete for single objects and arrays, or using functions from the C Standard Library, such as malloc, calloc, realloc, and free.
- The most common approach in modern C++ is to use new and delete for managing dynamic memory.

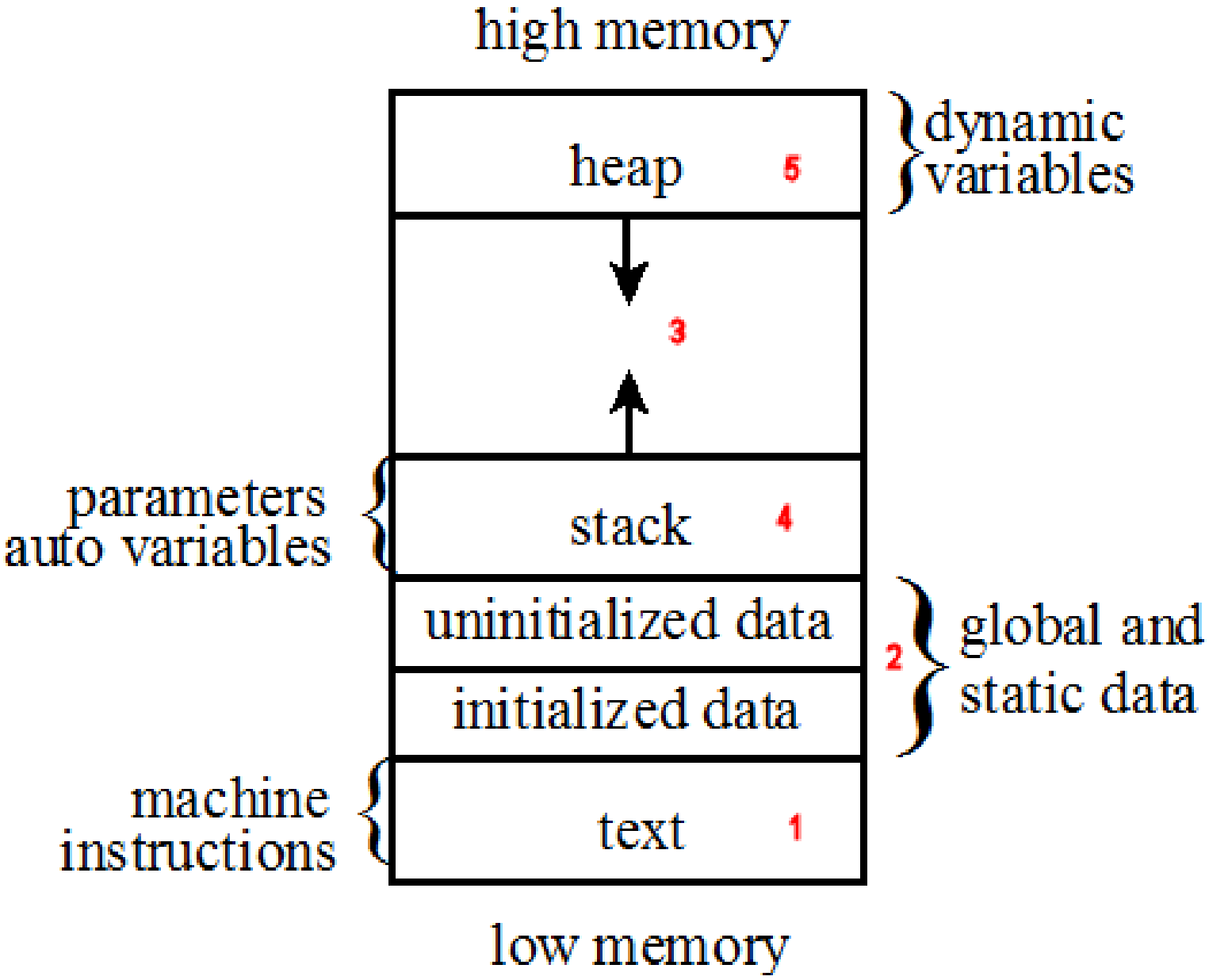
Dynamic Memory Allocation with new:
The new operator is used to allocate memory for a single object or an array on the heap.
It returns a pointer to the allocated memory.
Example:

int* dynamicInt = new int; // Allocate memory for an integer
int* dynamicArray = new int[10]; // Allocate memory for an integer array

Dynamic Memory Deallocation with delete:The delete operator is used to free the memory allocated with new.
It is crucial to release memory to prevent memory leaks.
Example:

delete dynamicInt; // Deallocate memory for the integer
delete[] dynamicArray; // Deallocate memory for the integer array

N.B:Dynamic allocation is beneficial when you need to work with data structures of varying or unknown sizes


high memory
heap
5
dynamic
variables
3
parameters
auto variables
stack
4
uninitialized data
2
global and
static data
initialized data
machine
instructions
text
1
low memory