Slot 07 - Advanced Topics in Pointers

Presenter:

Dr. LE Thanh Tung

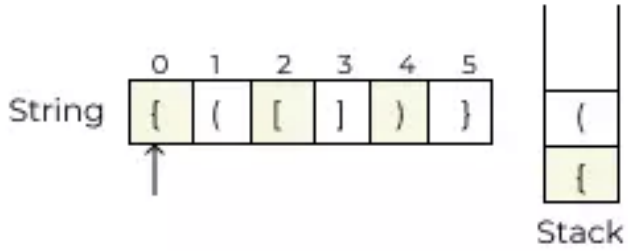
Content

1. Balanced Parentheses
2. Pointer in Function
3. Function Pointers

Balanced Parenthesis Problem

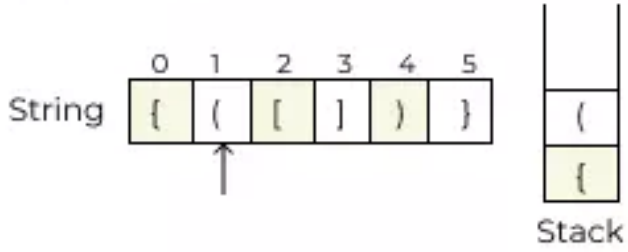
Step 1

Closing brackets Check top of the stack is same or not



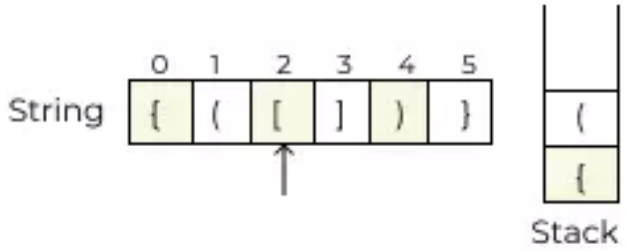
Step 2

Closing brackets Check top of the stack is same or not



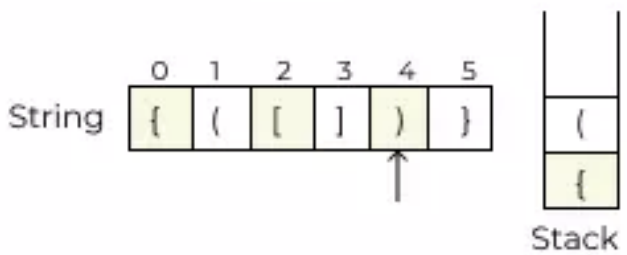
Step 3

Closing brackets Check top of the stack is same or not



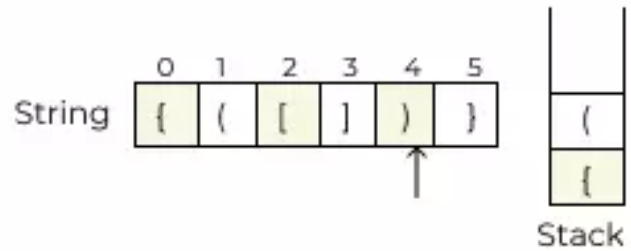
Step 4

Closing brackets Check top of the stack is same or not



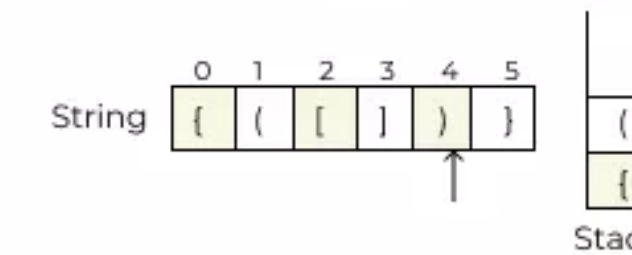
Step 5

Closing brackets Check top of the stack is same or not



Step 6

Closing brackets Check top of the stack is same or not



Balanced Parentheses

1. Define the opening and closing parentheses
2. Define the stack and its operations via Linked List
3. Repeat:
   a) Put the open bracket into stack
   b) If meeting the close bracket, pop the stack
      i. If it is not available, return False

   Until the end of sample

Pointer in Function

- What is the output of the following program:

```
int * createAnInteger(int value = 0)
{
        int myInt = value;
        return &myInt;
}

int main()
{
        int *pInt = createAnInteger(10);
        cout << *pInt << endl;

        return 0;
}
```

Pointer in Function

```
int * createAnInteger(int value = 0)
{
        int myInt = value;
        return &myInt;
}

int main()
{
        int *pInt = createAnInteger(10);
        cout << *pInt << endl;

        return 0;
}
```

- myInt is allocated by Automatic memory allocation and is reclaimed by the OS after exiting the function
- However, the address of myInt, which is reclaimed by OS, is assigned to pInt and utilizing in the next command.

  → illegal memory access

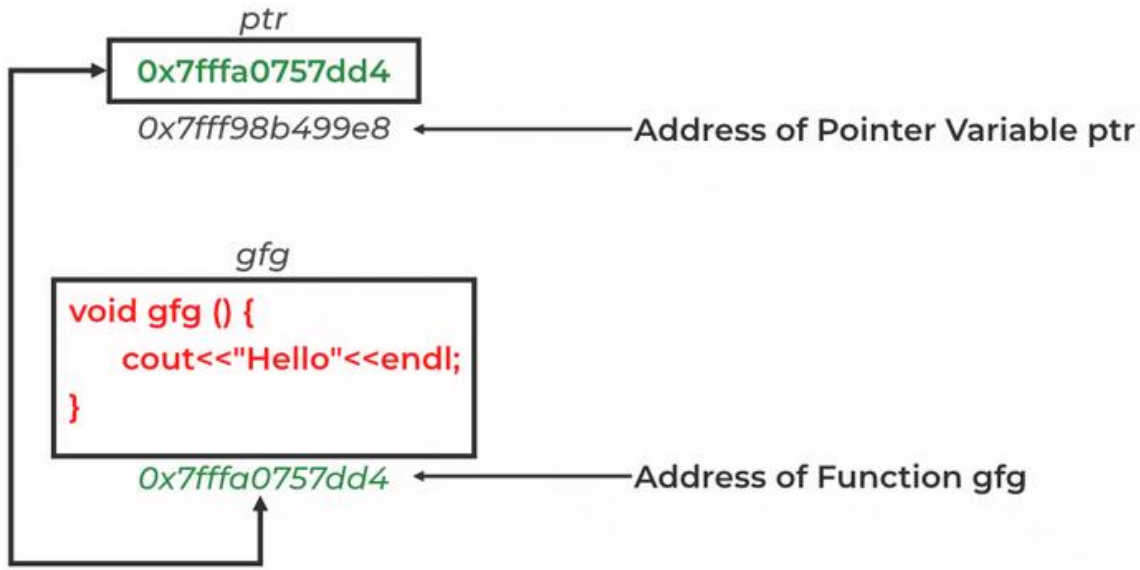
Address of Function

- Every function resides in memory and so it also has an address like all other variables in the program
- In C/C++, name of a function can be used to find address of function

```
1  #include <iostream>
2
3  using namespace std;
4
5  void greets(){
6      cout << "Hello Everyone";
7  }
8
9  int main(){
10     cout << reinterpret_cast<void*>(greets);
11     return 0;
12 }
```

Function Pointer in C++

- The function pointer is used to point functions, similarly, the pointers are used to point variables
- It is utilized to save a function's address
- The function pointer is either used to call the function or it can be sent as an argument to another function



Function pointers syntax

- Declare a function pointer:

```
<return_type> (*<name_of_pointer>)( <data_type_of_parameters> );
```

- For example,

```
void swapValue(int &value1, int &value2)
{
        int temp = value1;
        value1 = value2;
        value2 = temp;
}
```

```
int main()
{
        void(*pSwap) (int &, int &) = swapValue;
        cout << pSwap << endl;
        cout << swapValue << endl;

        return 0;
}
```

Wrong Declaration

- Only pointers declared with the appropriate return_type and list of parameters can point to the function

```
// function prototypes
int foo();
double goo();
int hoo(int x);

// function pointer assignments
int (*funcPtr1)() = foo; // okay
int (*funcPtr2)() = goo; // wrong -- return types don't match!
double (*funcPtr4)() = goo; // okay
funcPtr1 = hoo; // wrong -- fcnPtr1 has no parameters, but hoo() does
int (*funcPtr3)(int) = hoo; // okay
```

Dereference Function Pointer

- We can utilize deference function pointer to call the function

```
void swapValue(int &value1, int &value2)
{
        int temp = value1;
        value1 = value2;
        value2 = temp;
}

int main()
{
        void(*pSwap) (int &, int &) = swapValue;

        int a = 1, b = 5;
        cout << "Before: " << a << " " << b << endl;
        (*pSwap)(a, b);
        cout << "After:  " << a << " " << b << endl;

        return 0;
}
```

Dereference Function Pointer

- However, we also call the function indirectly via the function pointer

```
#include <iostream>
using namespace std;

int multiply(int a, int b) { return a * b; }

int main()
{
    int (*func)(int, int);

    // func is pointing to the multiplyTwoValues function

    func = multiply;

    int prod = func(15, 2);
    cout << "The value of the product is: " << prod << endl;

    return 0;
}
```

Passing Function Pointer

- How to reuse the following function to sort an array in decreasing order

```
void selectionSort(int *arr, int length)
{
        for (int i_start = 0; i_start < length; i_start++)
        {
                int minIndex = i_start;

                for (int i_current = i_start + 1; i_current < length; i_current++)
                {
                        if (arr[minIndex] > arr[i_current])
                        {
                                minIndex = i_current;
                        }
                }

                swap(arr[i_start], arr[minIndex]); // std::swap
        }
}
```

Passing Function Pointer

- If we can control the comparison operations between two elements, it is easy to choose the sorting direction

```
void selectionSort(int *arr, int length)
{
    for (int i_start = 0; i_start < length; i_start++)
    {
        int minIndex = i_start;

        for (int i_current = i_start + 1; i_current < length; i_current++)
        {
            // replace comparison expression by ascending function
            if (ascending(arr[minIndex], arr[i_current]))
            {
                minIndex = i_current;
            }
        }

        swap(arr[i_start], arr[minIndex]); // std::swap
    }
}
```

Passing Function Pointer

- First, we need to define two comparison functions

```
bool ascending(int left, int right)
{
        return left > right;
}

bool descending(int left, int right)
{
        return left < right;
}
```

Passing Function Pointer

- Secondly, we put the Function Pointer into Selection Sort as parameter

```
bool (*comparisonFunc)(int, int);
```

```
void selectionSort(int *arr, int length, bool (*comparisonFunc)(int, int))
```

Passing Function Pointer

- In this case, we can change the direction of sorting via the function pointer

```
bool ascending(int left, int right)
{
        return left > right;
}

bool descending(int left, int right)
{
        return left < right;
}
```

```
int main()
{
        int arr[] = { 1, 4, 2, 3, 6, 5, 8, 9, 7 };
        int length = sizeof(arr) / sizeof(int);

        cout << "Before sorted: ";
        printArray(arr, length);

        selectionSort(arr, length, descending);

        cout << "After sorted:  ";
        printArray(arr, length);

        return 0;
}
```

Passing Function Pointer

- In Selection Sort, the dynamic comparison is re-written by Function Pointer

```
void selectionSort(int *arr, int length, bool (*comparisonFunc)(int, int)){
    for (int i_start = 0; i_start < length; i_start++){
        int minIndex = i_start;
        for (int i_current = i_start + 1; i_current < length; i_current++){
            // use function pointer as ascending or descending function
            if (comparisonFunc(arr[minIndex], arr[i_current])) {
                minIndex = i_current;
            }
        }

        swap(arr[i_start], arr[minIndex]);
    }
}
```

Case study

- 1. Implement Selection Sort on doubly linked list by using a function pointer to specify the criteria for sorting in ascending or descending order

- 2. Get middle node in the linked list by using only one loop
- 3. Implement balanced parentheses via stack
- 4. Tìm dãy con tăng dần có độ dài dài nhất trong linked list, in nó ra

THANK YOU
for YOUR ATTENTION