

Pointers

1. What is a Pointer?

A **pointer** is a special type of variable that stores the memory address of another variable. Instead of holding a value directly, it "points to" the location in memory where the value is stored. This allows for more powerful and flexible programming, particularly for tasks involving dynamic memory management and efficient data manipulation.

2. Declaring and Initializing Pointers

You declare a pointer by specifying the data type of the variable it will point to, followed by an asterisk (*) and the pointer's name.

Syntax:

```
data_type *pointer_name;
```

Initialization can be done at declaration or later. A common practice is to initialize a pointer to `nullptr` (or `NULL` in C) to indicate that it doesn't point to any valid memory address.

```
#include <bits/stdc++.h>

using namespace std;

int main()
{
    int var = 10;

    // Declare and initialize a pointer to an integer
    int *ptr = &var;

    // 'ptr' now holds the memory address of 'var'
    cout << "Value of var: " << var << endl;
    cout << "Address of var: " << &var << endl;
    cout << "Value of ptr (address it holds): " << ptr << endl;
    return 0;
}
```

3. Address-of (&) and Dereference (*) Operators

- The **address-of operator (&)** is a unary operator that returns the memory address of its operand.
- The **dereference operator (*)** is a unary operator that returns the value located at the memory address stored in the pointer.

```
#include <bits/stdc++.h>

using namespace std;

int main()
{
    int var = 25;

    int *ptr = &var;

    // Use the dereference operator to get the value

    int value_at_ptr = *ptr;

    cout << "Value of var: " << var << endl;

    cout << "Address stored in ptr: " << ptr << endl;

    cout << "Value at the address stored in ptr: " << *ptr <<
endl;

    // You can also use the dereference operator to change the
value

    *ptr = 50;

    cout << "New value of var: " << var << endl;

    return 0;
}
```

4. Pointer Arithmetic

You can perform arithmetic operations like addition and subtraction on pointers. When you increment a pointer, it moves forward by the size of the data type it points to, not by a single byte.

```
#include <bits/stdc++.h>

using namespace std;

int main()
{
    int numbers[] = {10, 20, 30, 40, 50};

    int *ptr = numbers; // A pointer to the first element of the
array

    cout << "First element: " << *ptr << endl;

    // Move the pointer to the next integer address

    ptr++;

    cout << "Second element: " << *ptr << endl;

    // Move it back

    ptr--;

    cout << "First element again: " << *ptr << endl;

    // Point to the third element

    ptr = ptr + 2;

    cout << "Third element: " << *ptr << endl;

    return 0;
}
```

5. Pointer to a Pointer (Double Pointer)

A pointer can also store the memory address of another pointer. This is called a pointer to a pointer or a double pointer. It is declared using two asterisks (**).

When to use it:

- To create dynamic 2D arrays (an array of pointers, where each pointer points to an array of values).
- In a function, to modify the pointer variable of the caller itself (i.e., to make the caller's pointer point to a new memory location).

How to Initialize and Access:

You initialize a double pointer with the address of a single pointer.

- `ptr_to_ptr`: Holds the address of the single pointer.
- `*ptr_to_ptr`: Dereferences once to give the address of the final variable.
- `**ptr_to_ptr`: Dereferences twice to give the value of the final variable.

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int val = 10;

    // Declare a pointer 'ptr' and store the memory address of
    'val'.
    int *ptr = &val;

    // Print the address stored in 'ptr', which is the same as the
    address of 'val'.
    // Then, dereference 'ptr' to print the value it points to
    (10).
    cout << "ptr holds address: " << ptr << ", Address of val is:
    " << &val << ", Value *ptr points to: " << *ptr << endl;

    // Declare a pointer-to-a-pointer 'ptr2' and store the memory
    address of 'ptr'.
    int **ptr2 = &ptr;
```

```

    // Print the address stored in 'ptr2', which is the same as
the address of 'ptr'.
    // Then, dereference 'ptr2' once (*ptr2) to get the value it
points to (the address of 'val').
    cout << "ptr2 holds address: " << ptr2 << ", Address of ptr
is: " << &ptr << ", Value *ptr2 points to: " << *ptr2 << endl;

    // Use double dereferencing to access and modify the original
variable 'val'.
    // *ptr2 resolves to 'ptr'. **ptr2 resolves to 'val'.
    **ptr2 = 50;

    // Print the new value of 'val' to confirm it was changed via
the double pointer.
    cout << "New value of val: " << val << endl;
    return 0;
}

```

6. Pointers and Arrays

In C++, arrays and pointers have a close relationship. The name of an array can be used as a pointer to its first element.

```

#include <bits/stdc++.h>

using namespace std;

int main()
{
    int arr[5] = {5, 10, 15, 20, 25};

    // 'arr' acts as a pointer to arr[0]

    int *ptr = arr;

    cout << "Using array name as a pointer:" << endl;

    for (int i = 0; i < 5; i++)
    {

```

```

        // *(arr + i) is equivalent to arr[i]

        cout << "Element " << i << ": " << *(arr + i) << endl;
    }

    cout << "\nUsing a separate pointer:" << endl;

    for (int i = 0; i < 5; i++)
    {

        cout << "Element " << i << ": " << *ptr << endl;

        ptr++; // Move to the next element
    }

    return 0;
}

```

7. Function Argument Passing Methods in C++

In C++, arguments can be passed to functions in three primary ways: by value, by pointer, and by reference. Each method has distinct characteristics and use cases, particularly concerning how they handle the original data sent from the caller.

7.1. Pass-by-Value

In pass-by-value, the function receives a **copy** of the argument's value. The function works with this local copy, and any modifications made to the parameter inside the function **do not** affect the original variable in the calling scope.

When to use it:

- When you don't want the function to modify the original variable.
- For small data types (like int, char, double) where the cost of copying is low.

```

#include <bits/stdc++.h>

using namespace std;

void modify(int val)

```

```

{
    val = 50;
}

int main()
{
    int val = 10;

    modify(val);

    cout << val << endl;

    return 0;
}

```

7.2. Pass-by-Pointer

In pass-by-pointer, the function receives the **memory address** of the argument. By dereferencing this address (using the * operator), the function can directly access and modify the original variable in the calling scope.

When to use it:

- When the function needs to modify the original variable.
- When passing large objects or arrays to avoid the high cost of copying.
- To indicate optional arguments (by allowing a nullptr to be passed).

```

#include <bits/stdc++.h>

using namespace std;

void modify(int *ptr)
{
    *ptr = 50;
}

int main()
{
    int val = 10;

    modify(&val);

    cout << val << endl;

    return 0;
}

```

```
}
```

7.3. Pass-by-Reference

In pass-by-reference, the function receives an **alias** (or reference) to the original argument. The syntax is simpler than pointers, but the effect is similar: the function can directly access and modify the original variable. A reference acts as another name for the same memory location.

When to use it:

- When the function needs to modify the original variable (often preferred over pointers for its cleaner syntax).
- When passing large objects to avoid copying, and you are certain the argument will not be null.

```
#include <bits/stdc++.h>

using namespace std;

void modify(int &val)
{
    val = 50;

    // this is the same variable as in the main function

    cout << &val << endl;
}

int main()
{
    int val = 10;

    cout << &val << endl;

    modify(val);

    cout << val << endl;

    return 0;
}
```


8. Arrow Operator

In C++, when a pointer is pointing to an object, we use the arrow operator (`->`) to access the properties of that object.

```
#include <bits/stdc++.h>
using namespace std;
class Student
{
    public:
    int a,b;
    Student(int a,int b)
    {
        this->a=a;
        this->b=b;
    }
};
int main()
{
    Student obj(10,20);
    cout<<obj.a<<" "<<obj.b<<endl;
    return 0;
}
```

Pattern Printing

Remember two things:

1. The number of lines, which is controlled by the outer loop.
2. In each line, identify which values are changing, as those will be controlled by certain variables. Inside the inner loop, print the number of items based on those variables. Then, update how those values are changing after printing a new line.

1. Left-aligned triangle pattern

```
*
**
***
****
*****
*****
*****
*****
*****
*****
*****
```

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n;
    cin >> n;
    int k = 1;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= k; j++)
        {
            cout << "*";
        }
        cout << endl;
        k++;
    }
    return 0;
}
```

2. Left-aligned number pyramid

```
1
12
123
1234
12345
```

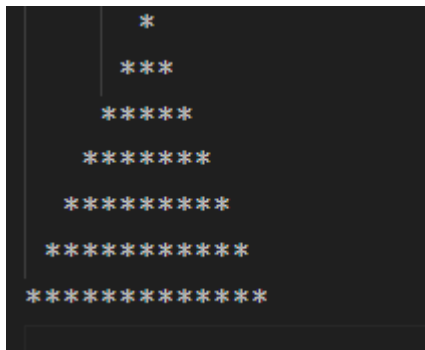
```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n;
    cin >> n;
    int k = 1;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= k; j++)
        {
            cout << j;
        }
        cout << endl;
        k++;
    }
    return 0;
}
```

3. Half diamond pattern

```
*
**
***
****
*****
*****
*****
*****
*****
****
***
**
*
```

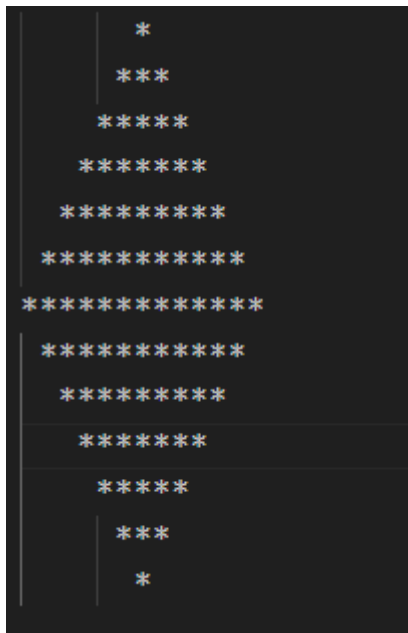
```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n;
    cin >> n;
    int k = 1;
    for (int i = 1; i <= n*2-1; i++)
    {
        for (int j = 1; j <= k; j++)
        {
            cout << "*";
        }
        cout << endl;
        if(i<n) k++;
        else k--;
    }
    return 0;
}
```

4. Full Pyramid Pattern



```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n;
    cin >> n;
    int k = 1, s = n - 1;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= s; j++)
        {
            cout << " ";
        }
        for (int j = 1; j <= k; j++)
        {
            cout << "*";
        }
        cout << endl;
        s--;
        k+=2;
    }
    return 0;
}
```

5. Full Diamond Pattern



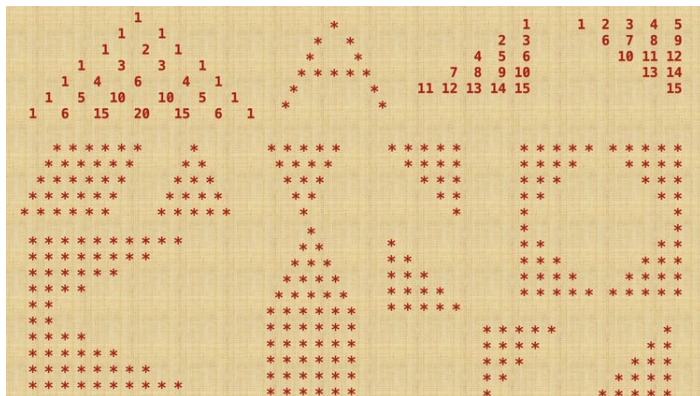
```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n;
    cin >> n;
    int k = 1, s = n - 1;
    for (int i = 1; i <= n * 2 - 1; i++)
    {
        for (int j = 1; j <= s; j++)
        {
            cout << " ";
        }
        for (int j = 1; j <= k; j++)
        {
            cout << "*";
        }
        cout << endl;
        if (i < n)
        {
            s--;
            k += 2;
        }
        else
    }
```

```


    {
        s++;
        k -= 2;
    }
}
return 0;
}

```


6. Patterns you can practice




Patterns in C Programming



Program in C to print the
Hollow Star Triangle
Pattern

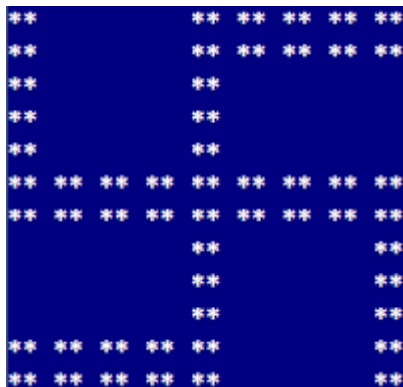


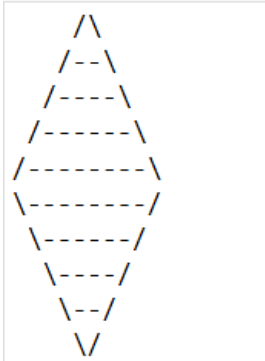
Program in C to print the
Hollow Number
Diamond Pattern



Program in C to print the
Characters' Triangle
Pattern

www.educba.com



[illegible]

And the ultimate challenge:

