Techimax

Fundamentals of Programming in C++

Scope of a Variable

The scope of a variable is defined as the **extent** of the program code within which the variable can we **accessed** or **declared** or worked with.

Function parameters, as well as variables defined inside the function body, are called **local variables**.
Local variables have **block scope**, which means they are in scope from their point of definition to the end of the block they are defined within.

Variables declared **outside of a function** are called **global variables**.
They can be accessed from **any portion** of the program.

Local Variables

```
int main() // outer block
{
    int x = 5; // x enters scope and is created here
    {              // nested block
        int y = 7; // y enters scope and is created here
    }              // y goes out of scope and is destroyed here
    // y can not be used here because it is out of scope in this block

} // x goes out of scope and is destroyed here
```

Global Variables

```
// global variable
int global = 5;

void display()
{
    cout << global << endl;
}

int main()
{
    display();
    global = 10;
    display();
}

// 5
// 10
```

Global vs Local Variables

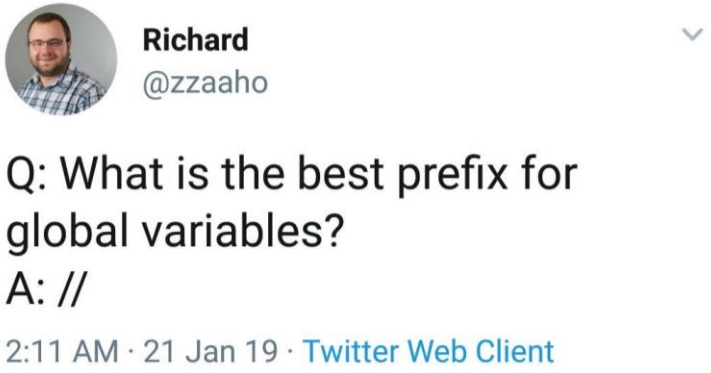
```
int x = 0;

int main()
{
    // Local x
    int x = 10;
    // :: → scope resolution operator
    cout << "Value of global x is " << ::x;
    cout << "\nValue of local x is " << x;
    return 0;
}
```

```
Value of global x is 0
Value of local x is 10
```

Why should we avoid using global variables ?

Global variables can be altered by any part of the code, making it difficult to remember or reason about every possible use.

A global variable can have no access control. It can not be limited to some part of the program.



Consider using a g” or “g_” prefix for global variables to help differentiate them from local variables.

Static Variables

```
void demo()
{
    // static variable
    static int count = 0;
    cout << count << " ";

    // value is updated and
    // will be carried to next
    // function calls
    count++;
}

int main()
{
    for (int i = 0; i < 5; i++)
        demo();
}
```

When a variable is declared as **static**, space for it gets allocated for the **lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the **value of variable in the previous call gets carried through** the next function call. This is useful for implementing coroutines in C/C++ or any other application where previous state of function needs to be stored.

```
0 1 2 3 4
```

Static Variables

```
void demo()
{
    // static variable
    static int count = 0;
    cout << count << " ";

    // value is updated and
    // will be carried to next
    // function calls
    count++;
}

int main()
{
    for (int i = 0; i < 5; i++)
        demo();
}
```

When a variable is declared as **static**, space for it gets allocated for the **lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the **value of variable in the previous call gets carried through** the next function call. This is useful for implementing coroutines in C/C++ or any other application where previous state of function needs to be stored.

```
0 1 2 3 4
```

An Introduction to Vectors

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

int main()
{
    vector<int> v1;

    for (int i = 0; i < 7; i++)
        v1.push_back(i);

    for (int i = 0; i < 7; i++)
        cout << v1[i] << " ";
}
```

Vectors are same as **dynamic arrays** with the ability to **resize itself automatically** when an element is inserted or deleted, with their storage being handled automatically by the container.
Vector elements are placed in **contiguous storage** so that they can be accessed and traversed using iterators.

To dive deeper, refer to the official documentation

http://www.cplusplus.com/reference/vector/vector/

```
0 1 2 3 4 5 6
```

Command Line Arguments

Command line arguments are optional string arguments that are passed by the operating system to the program when it is launched. The program can then use them as input (or ignore them). Much like **function parameters** provide a way for a function to provide inputs to another function, command line arguments provide a way for people or programs to provide inputs to a program.

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

// argc → argument count
// argv → argument vector
int main(int argc, char **argv)
{
    cout << "no of arguments : " << argc << endl;
    for (int i = 0; i < argc; i++)
    {
        cout << i << " " << argv[i] << endl;
    }
}
```

In Code::Blocks, choose "Project -> Set program's arguments".

```
>_ a.exe 4 2 "hello"
no of arguments : 4
0 a.exe
1 4
2 2
3 hello
```

Command Line Arguments

Task : Print the sum of numbers passed using cmd line arguments

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

int main(int argc, char **argv)
{
    int sum = 0;

    for (int i = 1; i < argc; i++)
    {
        cout << argv[i] << " ";
        sum += atoi(argv[i]);
        // Parses the C-string str interpreting its content
        // as an integral number,
        // which is returned as a value of type int.
    }
    cout << '\n'
         << sum << endl;
}
```

```
>_ a.exe 5 4 -3
5 4 -3
6
```