Miscellaneous Topics

Ajit Rajwade

Contents

1) Static Variables

- When a variable is declared as **static**, storage space for the variable is allocated and the variable is initialized **only once**.
- This happens before the program begins its execution.
- Even if the static variable is declared in a function, it will **retain its value across multiple function calls**.
- That is the value of a static variable in one function call will carried over to other function calls, until the program exits.
- Consider an example on the following slide.

```
void demo_static_variable()
{
    // 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_static_variable();
    return 0;
}
```

The output of the program is as follows:
01234

If `count` were not declared as `static` in `demo_static_variable`, then it would be set to 0 inside each call to the function.

Note that the scope of `count` is still restricted to the function where it was declared. `count` is local to `demo_static_variable`, but static and thus it retains its value across multiple calls to this function.

Static variables are useful in counting the number of times a function has been called during a program (including recursive calls).

```
for(i=0;i<10;i++)
{
  int a = 10;
  a++;
  cout << a << endl;
}
```

```
for(i=0;i<10;i++)
{
   static int a = 10;
   a++;
   cout << a << endl;
}
```

Static variables also exhibit similar behaviour when declared at the beginning of a block. For example, the code snippet on the left will print the value 11 ten times. The code snippet on the right will however print all values from 11 to 20, because the initialization of static variable `a` to 10 occurs only the first time. Thereafter `a` retains its value and does not get re-initialized to 10 in the beginning of every iteration of the for loop.

```
 for(j=0;j<2;j++){
 for(i=0;i<10;i++)
 {
    static int a = 10;
    a++;
    cout << a << endl;
 }
}
```

The code snippet here on the left will print values from 11 to 20 when j = 0 (first iteration of the outer for loop) and then the values 21 to 30 when j = 1.

2) Measurement of Execution Time

- A popular method is using the `time` function.
- This exists in the header file `ctime`.
- It requires declaration of variables of type `time_t`.
- The `time` function tells you the time elapsed since 1st January 1970 12:00 am (called the Epoch) till a particular point in the program.
- This `time` is measured in seconds.
- This feature can be used to find the time lapsed between two points in a program.
- See next slide for an example.

```
#include<iostream>
#include<ctime>
using namespace std;
void myfun(int n){
        int i,j;
        int a;

        for(i=0;i<n;i++){
                for(j=0;j<n;j++){
                        a = i+j+i*j;
                }
        }
}
int main(){
        time_t start, end;
        int n = 50000;

        time(&start);

        myfun(n);

        time (&end);

        double time_exec = double(end-start);
        cout << time_exec << " secs";
}
```

2) Measurement of Execution Time: One more method

- Use a function called `clock()` which returns the number of clock ticks at a certain point since the program was launched.
- The difference between the number of clock ticks at two points tell us the time spent in executing the code between these two points.
- To convert to seconds, you need to divide this number by `CLOCKS_PER_SEC` which is defined (by `#define`) in the `ctime` header file.
- See next slide.

```
#include<iostream>
#include<ctime>
using namespace std;
void myfun(int n){
        int i,j;
        int a;

        for(i=0;i<n;i++){
                for(j=0;j<n;j++){
                        a = i+j+i*j;
                }
        }
}
int main(){
    // clock_t clock(void) returns the number of clock ticks elapsed since the program was launched
    clock_t start, end;

    // Recording the starting clock tick
    start = clock();

    myfun();

    // Recording the end clock tick.
    end = clock();

    // Calculating total time taken by the program.
    double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    cout << "Time taken by program is : " << time_taken << " sec.";
}
```

3) Random Number Generation

- C++ has a function called `rand()` which generates an integer uniformly at random within the range 0 to RAND_MAX.
- RAND_MAX is typically the largest value of unsigned integer.
- The exact algorithm for generation of random numbers (rather pseudo random numbers) is beyond the scope of our course.
- The function `rand()` exists within the header file `cstdlib` which you must include.
- Each time `rand()` is called, a new number is generated which is a carefully chosen function of the previous number generated by `rand()`, in the program.
- The first number in this sequence is called the **seed**.
- C++ allows you to choose whatever seed you want.
- For the same value of the seed, `rand()` will always generate the same sequence of numbers each time the program is executed.
- To avoid this, you should set **seed to be the current time elapsed** since the Epoch using the time function. This is done via a function called `srand`.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main()
{
    // This program will create
some sequence of
    // random numbers on every
program run
    for (int i = 0; i < 5; i++)
        cout << rand() << " ";

    return 0;
}
```

```
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;
int main() {
    time_t t;
     time (&t);
     srand(t);
// This program will now create a
// different sequence of random numbers
// on every program run as the seed was
// set equal to the current time
    for (int i = 0; i < 5; i++)
        cout << rand() << " ";

    return 0;
}
```

3) Random Number Generation

- How can you generate random integers from 0 to N-1? You can just use a modulo: `rand()%N` instead of `rand()`.
- How to generate fractional numbers between 0 and 1 uniformly at random: use `double(rand())/RAND_MAX` instead of `rand()`.

4) Pointers to Functions

- A pointer to a function is a pointer whose value is the address of a function.
- For example:

```
int f (int); // declare a function

int (*pf)(int); // declare a pointer to a function

pf = &f; // address of f assigned to pf
```

- Pointers to functions allow us to define functions of functions.
- See following example.

```
// Returns the sum f(0) + f(1) + f(2) + . . . +
f(n-1):
int sum(int (*pf)(int k), int n) {
    int s = 0;
    for (int i = 1; i <= n; i++)
    s += (*pf) (i);
    return s;
}
int square(int k) {
return k*k;
}
int cube(int k){
return k*k*k;
}

int main () {
cout << sum(square,4) << endl; //1+4+9+16
cout << sum(cube,4) << endl;
}
```

The call `sum(square,4)` will return `square(1) + square(2) + square(3) + square(4).` Similarly for `sum(cube,4);`

4) Pointers to Functions

- These can be used in place of a `switch` case as can be seen below:

```
#include <iostream>
void add(int a, int b){ cout << "Addition is " << a+b; }
void subtract(int a, int b){ cout << "Subtraction is " << a-b; }
void multiply (int a, int b){ cout << "Product is " << a*b); }
int main() {
// fun_ptr_arr is an array of function pointers
void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
unsigned int ch, a = 15, b = 10;
cout << “enter your choice (0): add, (1): subtract, (2): multiply”; cin
>> ch;
if (ch > 2) return 0;
(*fun_ptr_arr[ch])(a, b);
 return 0;
}
```

5) Constant Pointers and Pointers to Constant Datatypes

- A pointer to a constant integer (say) is a variable which contains the address of a constant integer.
- A constant pointer is a variable which contains an address that cannot be changed.

```
int a; const int b=10;
int * p; // a pointer to an int
++(*p); // ok: increments int *p
++p; // ok: increments pointer p
int * const cp = &a; // a constant pointer to an int
++ (*cp) ; // ok: increments int *cp
++cp; // illegal: pointer cp is constant
const int * pc; // a pointer to a constant int
++ (*pc) ; // illegal: int *pc is constant
++pc; // ok: increments pointer pc
const int * const cpc = &b; // a constant pointer to a constant int
++(*cpc); // illegal: int *cpc is constant
++cpc; // illegal: pointer cpc is constant
```

6) Function Overloading

- C++ allows you to define multiple functions with the same name as long as they have different parameter type lists.
- This is called **function overloading**.
- For example: Consider the function `int gcd(int a, int b)` defined in earlier lectures. We can additionally define the following functions:

```
int gcd (int p, int q, int r) { return gcd(gcd(p,q),r);}

int gcd(int u, int v, int w, int x) {

return gcd(gcd(u,v),gcd(w,x));

}
```

7) Function templates

- We often need to write separate functions which essentially perform the same operation but on different datatypes.
- For example
    - A function to swap two variables of some datatype (char, bool, int, float, double, etc.)
    - A function to return the absolute values of a number (int, float, double, etc.)
- Is it possible to write the “body” of the function once and pass the datatype as a parameter of some sort?
- The answer is yes - via a feature called function templates.
- See examples on the next slides.

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

// declaration of a function template: note the syntax
template <typename T>
T myabs(T x)
{
        if (x >= 0) return x;
        return -x;
}

int main()
{
    // calling functions involving templates
    // no change in syntax
        int a = myabs(-10);
        float b = myabs(-1.234);
        double c = myabs(4.44);

        cout << a << " " << b << " " << c;
}
```

```
#include<iostream>
using namespace std;
template <typename T>
void tswap(T& x, T& y){
T temp = x;
x = y;
y = temp;
}
int main(){
        int a = 20, b = 30;
        tswap(a,b);
        cout << a << " " << b << endl;

        float c = 2.01, d = 3.22;
        tswap(c,d);
        cout << c << " " << d << endl;

        double x = 5.55, y = 3.33;
        tswap(x,y);
        cout << x << " " << y;
}
```

```
#include<iostream>
using namespace std;
template <typename T, typename U>
T addval(T& x, U& y){
        return x+y;
}
int main(){
    int a = 20, b = 30;
    cout << a << " " << b << " " << addval(a,b) <<
endl;

    float c = 2.01, d = 3.22;
    cout << c << " " << d << " " << addval(c,d) <<
endl;

    int x = 5; double y = 3.22;
    cout << x << " " << y << " " << addval(x,y);
```

```
Output:
20 30 50
2.01 3.22 5.23
5 3.22 8
```

You can have functions with more than one parameterized datatype, as illustrated here.

8) Time complexity analysis of the GCD function

- Let us recall the gcd computation from earlier slides (assume m >= n):

```
int main()
{
    int m,n,i;
    cout << “Enter the positive numbers (largest first): “; cin >> m >>
n;
    while (m%n !=0)
    {
        int remainder = m%n;
        m = n;
        n = remainder; //if n does not divide m, GCD(m,n) = GCD(n,m%n)
    }
    cout << “Their GCD is: “ << n; // if n divides m, GCD(m,n) = n
    return 0;
}
```

8) Time complexity analysis of the GCD function

- Let m_i, n_i denote the values of m,n in the beginning of the `i`th iteration of the while loop respectively.
- Let R_i be the value of the remainder computed in the `i`th iteration of the while loop.
- At the end of the `i`th iteration, we have $n_{i+1} = R_i = m_i \% n_i$. Also $m_{i+1} = n_i$ (due to the `m = n` statement).
- n_{i+1} must be smaller than m_{i+1} (which is equal to n_i) as the remainder modulo n_i must be smaller than n_i.
- In iteration `i+1`, we have $n_{i+2} = R_{i+1} = m_{i+1}$ mod n_{i+1}.
- Let `q` be the quotient when m_{i+1} is divided by n_{i+1}. Then $m_{i+1} = qn_{i+1} + R_{i+1}$.
- But $m_{i+1} > n_{i+1}$ and hence $q >= 1$, due to which $m_{i+1} >= n_{i+1} + R_{i+1} = n_{i+1} + n_{i+2}$.
- But $n_{i+1} > n_{i+2}$ since `n` must decrease across iterations. Hence $m_{i+1} >= n_{i+1} + n_{i+2} >= 2\, n_{i+2}$.
- Thus `n` drops by a factor of at least 2 in every 2 iterations.
- Also `n` never drops below 1. Hence the number of iterations is at the most 2 $\log_2 n_0$ where n_0 is the initial value of `n`.