Lecture #3 - enum - namespaces - typedef

- Recursion - Runtime stack - Activation record - Tracing a recursive function

Enumeration type - C++ allows programmers to create new data types.

- **enum** (Reserved word) - A user-defined data type whose domain is an ordered set of literal values expressed as identifiers.

Ex #25: Define a new data type: `colors` (Semicolon)

```
enum colors  { BROWN, BLUE, RED, GREEN, YELLOW };
```

By default, identifiers represent an ordered set of values.

- Separate values with a comma.
- Each identifier has a default integer value.
  - BROWN = 0
  - BLUE = 1
  - RED = 2
  - GREEN = 3
  - YELLOW = 4
- ***colors*** is the name of the enumeration (new data type)
- BROWN becomes a symbolic constant with a value of 0, BLUE = 1, etc.
- The values (BROWN, BLUE, etc.) are in order: BROWN < BLUE

Ex #26: `enum grades {'A', 'B', 'C', 'D', 'F'};` ← Wrong – Values must be identifiers.

Ex #27: `enum grades { A, B, C, D, F };` ← Correct

Ex #28: `enum days {SUN, MON, TUES, WED, THU, FRI, SAT};`

Variable Declaration and Assignment of Enumeration type

Ex #29: (Usually declare enum type global)

```
enum sports { BASKETBALL, FOOTBALL, SOCCER, BASEBALL };

int main( )
{
     sports popularSport;
     sports mySport = FOOTBALL;

     popularSport = BASEBALL;
```

Ex #30: `enum days {SUN, MON, TUES, WED, THU, FRI, SAT};`

`days payDay;` // Declare a variable called payDay of type **days**.

`payDay = FRI;`

Ex #31: (Another way to declare variables)

- ***change*** and ***usCoins*** are variables of type coins

```
enum coins {PENNY, NICKEL, DIME, QUARTER, HALF} change, usCoins;
```

Operations on Enumeration Type

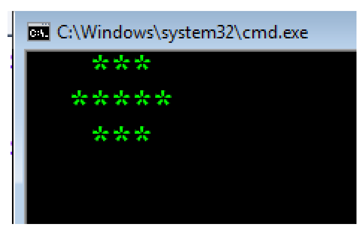
- No math operations on the enumeration type

Ex #32: `popularSport = mySport + 1;` ← Wrong

- Comparison operations can be done. (Relational operators)
  - Enumeration type is an ordered set of values, so relational operators can be used.

Ex #33: `if (mySport == yourSport)` ← OK

Exercise: enum trafficLights




```
#include <iostream>
#include <string>
#include <Windows.h>
using namespace std;

enum trafficLights { GREEN = 1, YELLOW = 2, RED = 4 };

void setTrafficLight(trafficLights bulbColor);
void displayLight(const char * color);

int main()
{
	trafficLights bulbColor = RED;	// Start with a red light
	setTrafficLight(bulbColor);

	bulbColor = GREEN;		// Change to a green light
	setTrafficLight(bulbColor);

	bulbColor = YELLOW;		// Change to a yellow light
	setTrafficLight(bulbColor);
```

```
     bulbColor = RED;              // Change to a red light
     setTrafficLight(bulbColor);

     cout << endl;
     system("pause");
     return 0;
}
void setTrafficLight(trafficLights bulbColor)
{
     switch (bulbColor)
     {
          // Font colors: A = green; C = red; E = yellow; - 0 = black bg
     case GREEN:    displayLight("Color 0A");
          break;
     case YELLOW:   displayLight("Color 0E");
          break;
     case RED:      displayLight("Color 0C");
          break;
     }
     system("cls");
}
// -------------------------------------------------------------
void displayLight(const char * color)
{
     system(color);
     cout << "    ***  \n"
          << "   ***** \n"
          << "    ***  \n";

     Sleep(2000);
}
```

Namespace - C++ supports the use of namespaces.

- **Namespace** - A namespace allows entities like classes, objects and functions to be grouped together under a name.

The format of namespaces is:

```
namespace identifier
{
   entities
}
```

Ex #34:
```
namespace first
{
   int var = 5;
}
```

- To use a variable (or object or function) defined in a namespace, prefix the variable with the namespace name and scope resolution operator **::**

Ex #35:
```
cout << first::var;          //  Output:   5
```

Ex #36:
```
#include <iostream>
using namespace std;

namespace first
{
   int var = 5;
}
namespace second
{
   double var = 3.1416;
}

int main()
{
   cout << first::var << endl;
   cout << second::var << endl;

   return 0;
}
```

```
/* OUTPUT:
5
3.1416   */
```

- **using** - The keyword *using* is used to introduce a name from a namespace into the current declarative region. For example:

Ex #37:
```
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}
```

```
namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main ()
{
  using first::x;
  using second::y;

  cout << x << endl;
  cout << y << endl;

  cout << first::y << endl;
  cout << second::x << endl;
```

```
/* OUTPUT:
5
2.7183
10
3.1416   */
```

- In the first two *cout* statements, **`x`** (without any name qualifier) refers to **`first::x`**, whereas **`y`** refers to **`second::y`**, exactly as the *using* declarations specify.
- The last two *cout* statements show that **`first::y`** and **`second::x`** can be accessed by using their fully qualified names.

- **`using namespace std;`** - Means the program is using the ANSI / ISO Standard C++.
  - The ANSI / ISO Standard C++ became official in 1998.
  - Global identifiers in standard libraries (like cout and cin in <iostream>), are identified in this namespace, and will be recognized by the compiler.

typedef statement – typedef is a reserved word

- ***typedef*** allows a programmer to use a new name with an existing data type.
- It does <u>not</u> create a new data type – just an additional name or alias.

<u>Ex #38</u>: Rather than write **unsigned short int** many times, create an alias for this phrase by using the keyword *typedef*.

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

```
typedef unsigned short int USHORT;     // Creates a new name USHORT that can
                                       //     be used instead of the longer form.
int main()
{
   USHORT width = 5;
   USHORT length;
```

Runtime Stack - A place in memory where data can be saved while a program runs.

- **LIFO** - Last In, First Out
- When a function is called (invoked), values and memory addresses are pushed on the run-time stack.
- They are saved on the stack so that they can be retrieved (popped) from the stack.

When a function is called...

- **Transfer of control** –Program control transfers from the calling block to the function code
- **Run-time stack** - When a function is called, an activation record is placed on the stack.
  - An activation record is also called a stack frame.

When a function is called - An activation record is created, and the following items are pushed on the runtime stack:

1.) **Return address** of the function call.
  - Control returns to the calling block after the function executes.
  - The return address is the memory address of the next program instruction.

2.) **Actual Parameters** - Values (arguments) passed in the function call are pushed on the stack.

3.) **Local variables** - Variables declared in the function block are pushed on the stack.

4.) **Return value** - If a function returns a value, it is first pushed on the stack, and later popped off and returned to the function call.

When a function is finished:

- The activation record for a particular function call is popped off the run-time stack when the final closing brace in the function code is reached, or when a return statement is reached in the function code.
- At this time the function's return value, (if non-void), is brought back to the calling block's return address for use there.
- If a function is void-returning, then no space is allocated on the runtime stack for a return value, and no return value is returned to the calling block.

Memory allocation for a value-returning function:

- Push storage for the return value
- Push the actual parameters (arguments)
- Push the return address
- Push storage for local function variables

De-allocation process for a value-returning function:

- Deallocate storage for local variables
- Pop the return address
- Deallocate the actual parameters
- Pop the return value

Memory Allocation for a void-returning function

- Push the actual parameters (arguments)
- Push the return address
- Push storage for local function variables

De-allocation process for a void-returning function

- Deallocate storage for local variables
- Pop the return address
- Deallocate the actual parameters

Example: The following simple program demonstrates a value-returning function, a **void-returning** function, the runtime stack, and return addresses after execution

Output →

```
Enter two numbers:
3
4

Press any key to continue */
```

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

```
// Function prototypes
int calcSum(int num1, int num2);
void displaySum(int sum);

int main ()
{
     int num1 = 0;   ---------->  0
     int num2 = 0;   ---------->  0
     int sum = 0;    ---------->  0

     cout << "Enter two numbers:  ";
     cin >> num1 >> num2;                3  (num1)
                                         4  (num2)

     sum = calcSum(num1, num2);          7  (sum)
```

ra1

Runtime stack when function is called.

(No local variables in the function)

stack	
ra1	**Push the return address (ra1)**
4	**Push actual parameter (4)** (num2)
3	**Push actual parameter (3)** (num1)
?	**Push storage for return value (retVal)**

(because it is a value-returning function)

```
     displaySum(sum);
```

Runtime stack when function is called.

(No local variables in the function)

ra2	**Push the return address (ra2)**
7	**Push actual parameter (7)** (sum)

(No storage for return value. because it is a void-returning function.)

```
ra2 → return 0;
}
```

```
// Function definition
int calcSum(int num1, int num2)
{
     return num1 + num2;
}
// Function definition
void displaySum(int sum)
{
   cout << "The sum is" << sum;
}
```

Recursive Functions - A function that calls itself.

- **Recursive call** - A function call in which the function being called is the same as the one making the call.
  - In other words, recursion occurs when a function calls itself.
- Recursion requires systems and languages that support dynamic memory allocation.
  - **Dynamic allocation** - Function parameters and local variables are not bound to addresses until an activation record is created at runtime.
- **Direct recursion -** Recursion in which a function directly calls itself.
- **Indirect recursion -** Recursion in which a chain of two or more function calls returns to the function that originated the chain.

Ex #1: This is a recursive function. It does not work, because it runs like an infinite loop, continually displaying the message.

```
void showMessage()
{
   cout << "This is a recursive function\n";
   showMessage();
}
```

Recursive call

Problem with Ex #1 above - The program is like an infinite loop.

Runtime stack - When a function is called, temporary data is pushed on the runtime stack.

- The data is removed when the function is finished.
- Data on the stack from the first function call is not removed before the next function is called.
- Data from the second function call is pushed on the stack on top of the first data.
- Data from the third call is pushed on top of the second, and so on…

Result: The stack grows until the computer eventually runs out of memory and the program crashes.

Solution: Include code to control the number of times the function is called.

<u>Ex</u> #2:

```
int times = 10;
void showMessage(int times)
{
     if (times > 0)
     {
          cout << "This is a recursive function\n";
          showMessage(times - 1);
     }
}
```

Solving Problems with Recursion

- A problem can be solved with recursion if it can be broken down into successive smaller problems that are identical to the overall problem.
- Recursion can be a powerful tool for solving repetitive problems and an important topic in upper-level computer science courses.

Recursion - Be careful when using recursion.

-Recursive solutions can be less efficient than iterative solutions.

-Still, many problems lend themselves to simple, elegant, recursive solutions.

-When recursion is not possible or appropriate, a recursive algorithm can be implemented non-recursively by using a looping structure.

<u>In general, a recursive function works like this</u>:

1.) To use recursion, identify at least one case in which the problem can be solved without recursion. This is called the **base case**.

- **Base case** - A non-recursive way out of the function.

2.) If the problem cannot be solved now, then the function reduces it to a smaller but similar problem and calls itself to solve the smaller problem. (**recursive case**)

- By reducing the problem with each recursive call, the base case will eventually be reached and the recursion will stop.
- **Recursive case** - Each recursive function call leads to a smaller case of the original problem, which eventually leads to the base case.

Two considerations, when using a recursive function:

- Ask if there is a non-recursive solution to the problem.
- Each recursive function call should involve a <u>smaller case</u> of the original problem thus leading to the base case. (making progress)

Recursive Function - countChars function.

This program demonstrates a recursive function for counting the number of times a character appears in a string.

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

int countChars(char letter, char message [], int index);

const int SIZE = 25;

int main()
{
    char letter = 'd';
    char message [SIZE] = "abcdddef";

    cout << "the letter d appears "
         << countChars(letter, message, 0) << " times.\n";
    return 0;
}
// ============================================================

// ============================================================
// This function counts the number of times the character search appears in the string str.
// The search begins at the subscript stored in subscript.

int countChars(char letter, char message [], int index)
{
    if (message [index] == '\0')
    {
        return 0;      // Base case:   The end of the string is reached.
    }
    else if (message[index] == letter)
    {
        // Recursive case:  A matching character was found.
        // Return 1 plus the number of times the search character
        //       appears in the rest of the string.
        return 1 + countChars(letter, message, index + 1);
    }
    else
    {
        // Recursive case:  A character that does not match the search
        //       character was found.  Return the number of times the
        //       search character appears in the rest of the string.
        return countChars(letter, message, index + 1);
    }
// ============================================================
```

```
/* OUTPUT
The letter d appears 3 times.
Press any key to continue . . . */
```

Recursive **gcd function** (greatest common divisor)

- The following function finds the largest divisor of two positive numbers (x and y).

  Ex:
  ```
  x = 20
  y = 12
  Largest divisor is 4
  ```

  Ex:
  ```
  x = 24
  y = 12
  Largest divisor is 12
  ```

- The definition states that the gdc of ***x*** and y is ***y***, if ***x/y*** is zero. (**base case**)
- Otherwise, the answer is the gdc of ***y*** and the remainder of ***x/y***. (**general case**)

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

int gcd(int, int);

int main()
{
   int num1 = 0;
   int num2 = 0;

   cout << "Enter two positive numbers:\n";
   cin >> num1 >> num2;

   cout << "The greatest common divisor of " << num1 << " and "
        << num2 << " is " << gcd (num1, num2) << endl;

   return 0;
}
// ==== gcd() ========================================
int gcd (int x, int y)
{
   if (x % y == 0)
      return y;
   else
      return gcd (y, x % y);
}
// ===================================================
```