CIS 129
Advanced Computer Programming

Chapter 10

Mr. Horence Chan

Enumeration

- Enumerations (enum) is a set of named integer values
- To define an enumeration type, you need the following items:
  - A **name** for the data type
  - A set of **values** for the data type
  - A set of **operations** on the values
- The syntax for enumeration is:
- `Enum typeName {value1, value2, …};`

Enumeration

- E.g. `enum Day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};`
- **Day** is the name for the data type
- **Sun, Mon, Tue, Wed, Thu, Fri, Sat** are enumeration values (enumerators), all of them are _____
- Without specification, the first value is _____, and the next one is increment by _____.
- i.e. Sun=0, Mon=1, Tue=_____, …, Sat=_____

Enumeration

- We can specify the values of some enumerators if necessary, those without specification will increment by _____ based on previous value
- E.g. `enum Day {Sun=7, Mon=1, Tue, Wed, Thu, Fri, Sat};`
- Values of each enumeration are:
- Sun=7 , Mon= 1, Tue=_____, Wed=_____,…

Declaring Variables

```
#include <iostream>
using namespace std;
int main(){
enum fruit {apple=10, orange=20, grape=50};
fruit quantity;
quantity = orange;
cout <<"The quantity of oranges is "
       << quantity << endl;
return 0;
}
```

- In this example:
- Defines an enumeration type called **fruit**
- Declares **quantity** to be variables of type `fruit`.
- `quantity` is also call enumeration object

Enumeration Assignment

```
#include <iostream>
using namespace std;
int main(){
enum fruit {apple=10, orange=20, grape=50};
fruit quantity;
quantity = orange;
cout <<"The quantity of oranges is "
       << quantity << endl;
return 0;
}
```

- Once a variable is declared, we can store values in it
- Stores **orange** in `quantity`

Output:

The quantity of orange is _____

Operations on Enumeration Types

```
#include <iostream>
using namespace std;
int main(){
enum fruit {apple=10, orange=20, grape=50};
fruit quantity;
quantity = orange;

quantity = quantity * 2;
quantity = apple + orange;
quantity++;

return 0;
}
```

- No _____ operations are allowed on enumeration
- **Increment** and **decrement** operations are not allowed to enumeration types

Operations on Enumeration Types

```
#include <iostream>
using namespace std;
int main(){
enum fruit {apple=10, orange=20, grape=50};
fruit quantity, remain;
quantity = orange;
remain = static_cast<fruit>(orange - 15);
cout << "The number of oranges remain today is "
      << remain << endl;
return 0;
}
```

- _____ operator is used for operation

Output:

The number of oranges remain today is _____

Relation Operations

```
#include <iostream>
using namespace std;
int main(){
enum fruit {apple=10, orange=20, grape=50};
fruit quantity_apple, quantity_orange;
quantity_apple = apple;
quantity_orange = orange;

if (quantity_apple > quantity_orange) {
cout << "The quantity of apple is larger than orange.";
}
else {
cout << "The quantity of orange is larger than apple.";
}return 0;
}
```

- Relation operations can be used on enumeration

Output:

Loops

```
#include <iostream>
using namespace std;
int main(){
enum fruit {empty, apple, orange, grape};
fruit quantity, total_quantity;
total_quantity = empty;

for (quantity = apple; quantity <= grape; quantity = static_cast<fruit>(quantity + 1))
{
      total_quantity = static_cast<fruit>(total_quantity + quantity);
}
cout << "Total Quantity of fruit: " <<total_quantity;
return 0;
}
```

Output:

Total Quantity of fruit: _____

Functions and Templates

- Functions can take arguments of **specific** types and have a **specific** return type.
- Templates allow us to work with **generic** *types*.
- Through templates, rather than repeating function code for each new type we wish to accommodate,
- We can create functions that are capable of using the same code for different types.

Functions and Templates

- For example, this example calculate the sum of two integers

```
int sum(const int x, const int y) {
     return x + y;
  }
```

- To calculate the sum of “doubles”, it must be modified to the following:

```
double sum (const double x, const double y) {
     return x + y;
}
```

- Since copying the entire function for each new datatype can be problematic.
- To overcome this we rewrite `sum` as a function template.

Templates

- The format for declaring a function template is:

```
template <class identifier> function_declaration;
```

or

```
template <typename identifier> function_declaration;
```

- Both forms are equivalent to one another, regardless of what datatype *identifier* ends up being.
- We can then use *identifier* to **replace** all occurrences of the datatype we wish to generalize.

Templates

- So, we rewrite our sum function:

```
template <typename T> T sum(const T a, const T b) {
    return a + b;
}
```

- Now, when `sum` is called, it is called with a particular **datatype**, which will replace all `T`s in the code. To invoke a function template, we use:

```
function_name <type> (parameters);
```

Templates

- In the `main` function:

```
int main() {
    cout << sum<int>(1, 2) << endl;
    cout << sum<float>(1.21, 2.43) << endl;
    return 0;
}
```

- This program prints out `3` and `3.64` on separate lines.
- The *identifier* can be used in any way inside the function template, as long as the code **makes sense** after *identifier* is replaced with some datatype.

Templates

- It is also possible to invoke a function template without giving an explicit type, in cases where the generic type *identifier* is used as the type for a parameter for the function. In this example, the following would also have been valid:

```
int main() {
      cout << sum(1, 2) << endl;
      cout << sum(1.21, 2.43) << endl;
return 0;
}
```

How about

```
      cout << sum(1, 2.43) << endl; // ok or error ?
      cout << sum<float>(1, 2.43) << endl; // ok or error ?
```

Templates

- Templates can also specify more than one type parameter. For example:

```
#include <iostream>
using namespace std;
template <typename T, typename U>
U sum(const T a, const U b) {
    return a + b;
}
int main() {
    cout << sum<int, float>(1, 2.5) << endl;
    return 0;
}
```

Output:

Standard Template Library

- Part of the C++ Standard Library, the **Standard Template Library** (STL) contains many useful container classes and algorithms.
- As you might imagine, these various parts of the library are written using templates and so are generic in type.
- The containers found in the STL are lists, maps, queues, sets, stacks, and vectors.
- The algorithms include sequence operations, sorts, searches, merges, heap operations, and min/max operations.

Standard Template Library

```
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;
int main() {
        set<int> iset;
        iset.insert(5);
        iset.insert(9);
        iset.insert(1);
        iset.insert(8);
        iset.insert(3);
        cout << "iset contains:";
        set<int>::iterator it;
        for(it=iset.begin(); it != iset.end(); it++)
                {cout << " " << *it;}
        cout << endl;
}
```

- In this example, we create an integer set and insert several integers into it.
- We then create an iterator corresponding to the set (`set<int>::iterator it;`)
- An iterator is basically a _____ that provides a view of the set. (Most of the other containers also provide iterators.)
- By using this iterator, we display all the elements in the set and print out :

```
iset contains: _____
```

- Note that the set automatically _____ its own items.

Standard Template Library

```
        int searchFor;
        cin >> searchFor;
        if(binary_search(iset.begin(), iset.end(), searchFor))
                cout << "Found " << searchFor << endl;
        else
                cout << "Did not find " << searchFor << endl;
return 0;
}
```

- Next, we ask the user for an integer, search for that integer in the set, and display the result.

Sample Output: (Two cases is show here)

2

Did not find 2

3

Found 3

Standard Template Library

```
#include <iostream>
#include <algorithm>
using namespace std;
void printArray(const int arr[], const int len) {
        for(int i=0; i < len; i++)
                cout << " " << arr[i];
        cout << endl;
}
int main() {
        int a[] = {5, 7, 2, 1, 4, 3, 6};
        sort(a, a+7);
        printArray(a, 7);
        rotate(a,a+3,a+7);
        printArray(a, 7);
        reverse(a, a+7);
        printArray(a, 7);
        return 0;
}
```

Output:

1 2 3 4 5 6 7

Operator Overloading

- We can make operators to work for user defined classes.
- This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as **operator overloading**
- The list of overloadable operators:

```
+       -       *       /       +=      -=      *=      /=      %       %=      ++      --
=       ==      <       >       <=      >=      !       !=      &&      ||
<<      >>      <<=     >>=     &       ^       |       &=      ^=      |=      ~
[]      ()      ,       ->*     ->      new     new[]   delete          delete[]
```

Operator Overloading

```
class Sum {
private:
    int X, Y;
public:
    Sum(int x = 0, int y = 0) {X = x;    Y = y;}
    Sum operator + (Sum &obj) {
        Sum total;
        total.X = X + obj.X;
        total.Y = Y + obj.Y;
        return total;
    }
    void print() { cout << X << "x + " << Y << "y" << endl; }
};
int main(){
    Sum f1(2, 3), f2(9, 7);
    Sum f3 = f1 + f2;
    f3.print();
    return 0;
}
```

- For example, we want to calculate the sum of “2x + 3y” and “9x + 7y”
- By using `Sum operator +` , we can calculate the sum of the objects in class `Sum`.

Output

11x + 10y

Exceptions

- Sometimes functions encounter errors that make it impossible to continue normally.
- To avoid the program terminate when an error is encounter, we can **throwing an exception**.
- We can specify how it should be handled when an exception (error) occurs.
- The program can _____ when an error is encounter.

Exceptions

```
#include <iostream>
using namespace std;
int main() {
  try {
    int age = 9;
    if (age >= 12) {
      cout << "Access granted - you are old enough.";
    } else {
      throw (age);
    }
  }
  catch (int myNum) {
    cout << "Access denied - You must be at least 12 years old.\n";
    cout << "Age is: " << myNum;
  }
  return 0;
}
```

- In this example, the codes inside `catch` are execute if the `age` is smaller than `12`
- It use `throw (age)` to call it

Output:

Access denied - You must be at least 12 years old.

Age is: 9

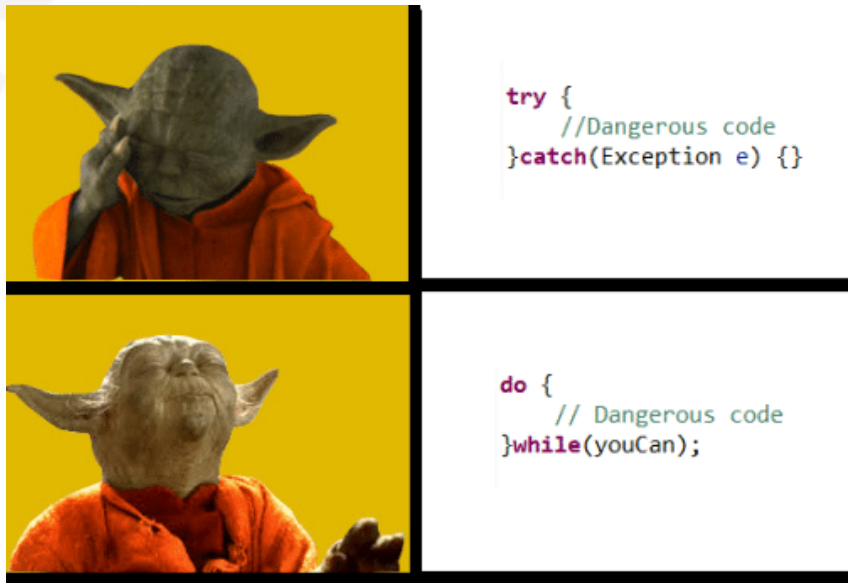
Exceptions

```
double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}
int main () {
   int x = 50;
   int y = 0;
   double z = 0;
   try {
      z = division(x, y);
      cout << z << endl;
   } catch (const char* msg) {
     cout << msg << endl;
   }
   return 0;
}
```

- One of a common usage of exceptions is to check if the divisor is zero.
- In this example, if the divisor (`b`) is zero, it print out the `msg`, rather than output a error message

Output

Division by zero condition!



friend Functions/Classes

- Recall that **private** fields/methods of a class _____ be access outside the class
- If you want to allow a function that is not a member of a given class to access the private fields/methods of that class.
- We can specify that a given external function gets full access rights by placing the signature of the function inside the class, preceded by the word **friend**.

friend Functions/Classes

```
#include <iostream>
using namespace std;
class Square {
   double width;
   public:
      friend void printArea( Square square );
      void setWidth( double wid );
};
void Square::setWidth( double wid ) {
   width = wid;
}
void printArea( Square square ) {
cout << "Area of square : "
        << square.width*square.width <<endl;
}
```

- `setWidth()` is a member function
- `printArea()` is not a member function of any class
- because `printArea()` is a friend of `Square`, it can directly access _____ member of this class (`Square`)

friend Functions/Classes

```
int main() {
        Square square;
        square.setWidth(10.0);
        printArea( square );
    return 0;
}
```

// set box width with member function

// Use friend function to print the width.

Output:

Area of square : _____

Preprocessor Macros

- Macros is a small snippets of code that depend on arguments.
- Macros are like small functions that are not type-checked
- They are implemented by simple textual substitution.
- Because they are not type-checked, they are considered less robust than functions.

```
using namespace std;
#include <iostream>
#define print cout <<
int main(){
        print "Hello World";
}
```

I'm taking a c++ class and last year I took a python class. Yes, this code works

Preprocessor Macros

```
#include <iostream>
#include <string>
#define sum(x, y) (x + y)
using namespace std;
int main(){
	cout<<sum(1,2)<<endl;
	cout<<sum(3.14,6.89)<<endl;
	//cout<<sum("a", "b")<<endl;
return 0;
}
```

- For example , we can write:
- `#define sum(x, y) (x + y)`
- Now, every time `sum(a, b)` appears in the code, for any arguments `a` and `b`, it will be replaced with _____
- Note that we can't add string using this macros

Output:

3

10.03