**Enumeration Type**

- Data type - a set of values together with a set of operations on those values
- Examples:
  - enum grades {F, PC, P, Cr, D, HD};
  - enum colors {brown, blue, red, green, yellow};
Enumeration Type

- The syntax for enumeration type is:
  
  ```
  enum typeName {value1, value2, ...};
  ```

  where value1, value2, ... are identifiers called enumerators

- (Typically) value1 < value2 < value3 < ...

- Enumeration type is an ordered set of values

If a value has been used in one enumeration type, it cannot be used by another in the same block

The same rules apply to enumeration types declared outside of any blocks

- `enum mathStudent {John, Mike, Jenny};`
- `enum compStudent {Jane, John, Bill};`

Examples

- The following are illegal enumeration types because none of the values is an identifier:

  ```
  enum grades {'A', 'B', 'C', 'D', 'F'};
  enum places {1st, 2nd, 3rd, 4th};
  ```

- The following are legal enumeration types:

  ```
  enum grades {A, B, C, D, F};
  enum places {first, second, third, fourth};
  ```
Enumeration Type

Declaring Variables
- The syntax for declaring variables is: `dataType identifier, identifier,...;`
- The following statement defines an enumeration type `sports`
  ```java
  enum sports {basketball, football, hockey, baseball, soccer, volleyball};
  ```
- The following statement declares 2 variables of the type `sports`.
  ```java
  sports popularSport, mySport;
  ```

Assignment
- The statement:
  ```java
  popularSport = football; //legal
  ```
  stores the word `football` in the variable `popularSport`
- The statement:
  ```java
  mySport = popularSport; //legal
  ```
  copies the contents of the variable `popularSport` in the variable `mySport`

Operations
- No arithmetic operation is allowed on enumeration types. *(They are constants!)*.
  - The following statements are illegal:
    ```java
    mySport = popularSport + 2; //illegal
    popularSport = football + soccer; //illegal
    popularSport = popularSport * 2; // illegal
    ```
## Enumeration Type

### Operations
- `enum` sports {basketball, football, hockey, baseball, soccer, volleyball};
- sports popularSport, mySport;
- popularSport = football;
- popularSport = popularSport + 1; //illegal
- popularSport = static_cast<sports>(popularSport + 1);
- // it is hockey
- popularSport = football;
- popularSport = static_cast<sports>(popularSport - 1);
- // it is basketball

Because an enumeration is an ordered set of values, we can use relational operators with them

\(<, >, ==,...\)

The cast operator can be used to increment, decrement, and compare the values which means that they can be used in loops.

### Input and output
- are defined only for built-in data types such as int, char, double
- the enumeration type can be neither input nor output (directly)
- you can input and output enumeration indirectly
**Enumeration Type**

**Input and output**

- `enum` `sports` `{basketball, football, hockey, baseball, soccer, volleyball};`  
- `sports` `mySport;`  
- `int` `mySportValue;`  

---

**Enumeration Type**

- An example: USA coins

<table>
<thead>
<tr>
<th>Coin</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>penny</td>
<td>1 ¢</td>
</tr>
<tr>
<td>nickel</td>
<td>5 ¢</td>
</tr>
<tr>
<td>dime</td>
<td>10 ¢</td>
</tr>
<tr>
<td>quarter</td>
<td>25 ¢</td>
</tr>
</tbody>
</table>

---

**Enumeration Type**

**Functions**

- Enumeration type can be passed as parameters to functions: either by value or by reference
- A function can return a value of the enumeration type

---

**Enumeration Type**

**Anonymous Data Types**

- Anonymous - a data type in which values are directly specified in the variable declaration with no type name, for example:

  ```cpp
  enum {basketball, football, baseball} mysport;
  // mysport is a variable, not the typename
  ```
Enumeration Type

Anonymous Data Types

- Creating an anonymous type has drawbacks:
  - you cannot pass an anonymous type as a parameter to a function
  - A function cannot return a value of an anonymous type
  - Values used in one can be used in another, but they are treated differently

The typedef Statement

- You can create synonyms or aliases to a previously defined data type by using the `typedef` statement
- The general syntax of the `typedef` statement is:
  ```
  typedef existingTypeName newTypeName;
  ```
- The `typedef` statement does not create any new data types, but only an alias to an existing data type

Example:

```c
typedef int counter;
counter c1, c2, c3;
```

Namespace

- A mechanism to solve the problem of overlapping global identifier names from different vendors
- Define a namespace
- Use a namespace
Namespace

Define a namespace
namespace globalType
{
    const int n=10;
    const double rate = 7.5;
    int count =0;
    void printResult();
}

It defines globalType to be a namespace with four members: constants n and rate, the variable count, and the function printResult.

Namespace

To access a member of a namespace
namespace globalType
{
    const int n=10;
    const double rate = 7.5;
    int count =0;
    void printResult();
} globalType :: count++; globalType :: printResult();

Namespace

Use namespace and namespace member

To simplify the accessing of all namespace members: using namespace namespace_name;
for example using namespace globalType;
After this statement, we can simply put count++; (without globalType and scope resolution operator ::)
To simplify the accessing of a specific namespace member: using namespace_name::identifier;
for example using globalType :: count;
After this statement, we can put count++;

Records (structs)

Records (structs)

struct - a collection of a fixed number of components in which components are accessed by name is called a struct
The components may be of different types
The components of a struct are called the members of the struct
struct is a definition not a declaration.
No memory is allocated
Memory is only allocated when we declare variables
Records (structs)

- The general form (syntax) of a struct is:
  ```
  struct typeName
  {
    dataType1 identifier1;
    dataType2 identifier2;
    
    dataTypen identifiern;
  }
  ```

Example

The general form for a student could be:
```
struct studentType{
  string firstName;
  string lastName;
  string studentNumber;
  double GPA;
};
```
studentType student, newstudent;

Accessing struct Members

- The syntax for accessing a struct member is:
  ```
  structVariableName.MemberName
  ```
- The . (dot) is an operator called the member access operator

```
student.firstName = "William";
student.lastName = "Gates";
student.studentNumber = "s1234567";
student.GPA=0.0;
```
Records (structs)

Input/Output

- There are no aggregate input/output operations on `struct`.
- Data in a struct variable must be read one member at a time.
- Contents of a struct must be written one member at a time.

<table>
<thead>
<tr>
<th>structRatio.cpp</th>
</tr>
</thead>
</table>

struct Variables and Functions

- A struct variable can be passed as a parameter either by value or by reference.
- A function can return a value of the type `struct`.

| structRatio2.cpp |

File input/output using struct

Example 1: read a line from the source copy and display it on the screen

```
stru_in.txt:
s1234567 William 2.2
s1234568 Smith 2.8
s1234569 Chang 3.0
```

| struct_file_in_out.cpp |

Example 2: copy all the source copy to the output file (stu_out.txt)

```
stru_in.txt:
s1234567 William 2.2
s1234568 Smith 2.8
s1234569 Chang 3.0
```

| struct_file_in_out_loop.cpp |