Learning Objectives

- Learn how to create and manipulate your own data type using `enum`
- Familiar with `typedef` statements
- Using `struct`
- Programming tips
  - Compiling options in g++
  - Importance of programming styles
  - Coding guidelines of structures

### Enumeration Type

- Data type: a set of values together with a set of operations on those values
- To define a new simple data type, called *enumeration* type
- Syntax for enumeration type
  ```
  enum typename {value1,value2,value3,...}
  ```
  - `value1,value2,...` are identifiers called *enumerators*
  - `value1 < value2 < value3`
- If a value has been used in one enumeration type $\Rightarrow$ cannot be used by another in the same block

### Examples of Enumeration Types

- **Ex1:**
  ```
  enum colors {brown,blue,red,green}
  ```
- **Ex2:**
  ```
  enum standing {FRESHMAN,SOPHOMORE, JUNIOR,SENIOR}
  ```
- **Ex3:** illegal! Why? Not valid identifiers!
  ```
  enum grades {'A','B','C','D','F'}
  enum year {1st,2nd,3rd,4th,5th}
  ```
- **Ex4:** illegal! Why? Repeated enumerators!
  ```
  enum Math {John,Peter,Sean,Joe}
  ```
  ```
  enum Comp {Paul,Judy,Joe,Mary,Jane}
  ```
**Enum Manipulation**

- Assignment and copy are legal
  - Ex: sport PopularSport = BASEBALL;
  - Ex: sport MySport = PopularSport;
- Arithmetic operations are NOT allowed!
  - Ex: ASport = FOOTBALL + 2;
  - Ex: BSport = 2 * Volleyball;
  - Ex: CSport++, DSport--;
- But casting is legal
  - Ex: NewSport = static_cast<sports>(FOOTBALL + 1)
- Relational operations are legal
  - Ex: if (ASport > BSport)

**Typedef Statement**

- You can create *synonyms* to a previously defined data type by using typedef
- Syntax of the typedef statement:
  ```
typedef existTypeName newTypeName;
  ```
  - Not create any new data types
  - Alias to an existing data type
- Examples:
  - typedef char[255] myString;
  - typedef unsigned int Positive;
  - typedef double * DoublePtr;

**Records**

- Recall: *aggregate* meaning "grouping"
  - array: collection of values of same type.
    - Ex: int scores[100];
    - scores: 85 79 92 57 68 80 ...
  - struct: collection of values of different types
    - employee: R. Jones 123 Elm 6/12/55 $14.75
- Major difference: must first "define" struct
  - Prior to declaring any variables

**Record in C++: struct**

- Syntax for defining a structure type:
  ```
  struct Structure_Tag
  {
    Type_1 Member_Name_1;
    Type_2 Member_Name_2;
    ...
    Type_N Member_Name_N;
  };
  ```
- Structured data type: tag + body
- Fixed number of components in body
- Consist of different *types* of components (fields)
- Elements accessed by *name*, not index
**Structure Types**

- **Define struct globally (typically)**
  - No memory is allocated
    - Just a "placeholder" for what our struct will "look like"
  - Example of struct definition:
    ```
    struct CDAccount //name of struct "type"
    {
        double balance; // member names
        double interestRate;
        int term;
    };
    ```
  - Common pitfall: forget `;`

**typedef & struct in C/C++**

- **[Type 1] use struct only**
  ```
  struct struct_tag
  {
      struct_body;
  };  
  ```
- **[Type 2] use struct with typedef ⇒ most common in C/C++**
  ```
  typedef struct {struct_body} struct_tag;
  ```
  Remember?

**struct Declare & Access**

- With structure type defined, now declare variables of this new type:
  - `ex: CDAccount account;`
    - Just like declaring simple types
    - Variable `account` now of type `CDAccount`
    - `account` contains *member values*
  - **Dot Operator (.)** to access members
    - `account.balance`
    - `account.interestRate`
    - `account.term`

**struct Assignments**

- Given structure named `CropYield`
  - Declare two structure variables:
    ```
    CropYield apples, oranges;
    ```
    - Both are variables of "struct type `CropYield`"
    - **Simple** assignments are legal:
      ```
      apples = oranges;
      ```
    - Simply copies each member variable from `apples` into member variables from `oranges`
Initializing Structures

- Can initialize at declaration
  - Ex:
    ```
    typedef struct {
        int month;
        int day;
        int year;
    } Date;
    ...
    Date DueDate = {12,31,2003};
    ```
    - Declaration provides initial data to all three member variables

struct in Functions

- Passed like any simple data type
  - Pass-by-value
  - Pass-by-reference
  - Or combination
- Can also be returned by function
  - Return-type is structure type
  - Return statement in function definition sends structure variable back to caller
  - Ex:
    ```
    CropYield Trade(CropYield fruit1, CropYield& fruit2) {
        ...
        return fruit3;
    }
    ```

Exercise: enum & struct

- You are asked to design a program for ABC airline to book tickets where each ticket needs to contain the following fields:
  - Flight number
  - Passenger's name
  - Starting airport
  - Destination airport
  - Seat number
  - Boarding date
  - Boarding time
- You can book a ticket or show each field

Compiling Options in g++

- Simplest way ⇒ generate a.out as executable
  - Ex: `g++ myprog.cpp > ./a.out`
- Option 1 –Wall: show all warning messages
  - Ex: `g++ -Wall myprog.cpp > ./a.out`
- Option 2 –o: name your executable like testprog
  - Ex: `g++ myprog.cpp -o testprog > ./testprog`
- Query the full listing of g++ options by man
  - Ex: `man g++`
Motivation of Programming Style

- An sample program (look like yours?)

```cpp
#include <iostream>
using namespace std;
int main(){
    int x,y,z;
    double p,q,r;
    cin>>x>>r;
    if(x>r){ y=r; cout<<(x*=y);} 
    else{y=x; q=++r; for(int i=0;i<q;i++){ if(y>r)cout<<(i*y);} 
    else cout<<(i/r)reurn 0; }
}
```

Motivation of Programming Style

- Programming style
  - a.k.a. programming norms
  - Avoid the in-readability problem of differing coding styles and habits in a team-work programming
    ⇒ develop a consistent writing standard
  - Avoid the bug occurrence

- Hence, define a simple and canonical rules for programming:
  - Coding style (CS)
  - Naming convention (NC)

Programming Style

Indent Styles

- K&R style:
  - a.k.a. kernel style because of Unix kernel
  - indented per level by 8 spaces in C, 4 in C++

- Whitesmiths style:
  - Statements are aligned at same level
  - indented per level by 8 spaces in C, 4 in C++

- GNU style:
  - Used in EMACS & Free Software Foundation code
  - Always indented 4 spaces per level and 2 spaces for { and }

- Allman style: the most common
  - a.k.a. BSD style because of BSD utilities
  - indented by 8 spaces in C, 4(or 3) spaces in C++

Examples of Indent Styles

<table>
<thead>
<tr>
<th>K&amp;R Style</th>
<th>Whitesmiths Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (&lt;cond&gt;) {</td>
<td>if (&lt;cond&gt;)</td>
</tr>
<tr>
<td>&lt;body&gt;</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&lt;body&gt;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Allman Style</td>
<td>GNU Style</td>
</tr>
<tr>
<td>if (&lt;cond&gt;) {</td>
<td>if (&lt;cond&gt;)</td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>&lt;body&gt;</td>
<td>&lt;body&gt;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
### Inter-Function New Line

- Add a new line after each declaration of functions.
- Ex:
  ```
  ... int function_01 ()
  {
    ...
  }

  int function_02 ()
  {
    ...
  }
  ```

### Inter-Statement New Line

- In the function body, new line is not placed between statements with close relationship (usually under the same control structure). Elsewhere, place a new line.
- Ex:
  ```
  while (entry_cond) {
    statement_1;
    if (dc_maker) {
      statement_2;
    } else {
      statement_3;
    }
    statement_4;
  }
  ```

### Single Statement for Single Task

- One statement only serves to:
  - declare one variable or
  - do one thing only (as fewer operations as possible)
- Ex:
  ```
  int height, weight, depth;
  int height; int weight; int depth;
  ```
  
  Bad Style Good Style
  ```
  x = y + z; a = b * c; p = q % r;
  ```
  ```
  x = y + x;
  a = b * c;
  p = q % r;
  ```

### Stand-alone Control Struct with {} 

- Control struct keywords if, else, for, do, while, always use a stand-alone line
- Append the compound statement by using {} from the next line
- Ex:
  ```
  if (age>18) cout<<"y";
  ```
  ```
  if (age > 18) {
    cout << "y";
  }
  ```
  ```
  for (i=0;i<10;i++) cout<<i;
  ```
  ```
  for (i=0;i<10;i++) {
    cout << i;
  }
  ```
Variable Declaration & Initialization
- Declare variables close to where they are used
- Initialize the variable immediately.
- Ex:

<table>
<thead>
<tr>
<th>Bad Style</th>
<th>Good Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>int LegalAge;…statement_1;…LegalAge = 18;…statement_2;…if (Age &gt; LegalAge) {…}</td>
<td>int LegalAge = 18; if (Age &gt; LegalAge) {…}</td>
</tr>
</tbody>
</table>

Spacing in Statements (1/2)
- Use at least one space after keywords like const, case, if and while
- Use at least one space after ‘,’ and ‘;’ if this is not the end of line
- No space after functions ⇒ append “(“ directly
- Do not reserve space after "(" and in front of ");", ;, and ;"
- Ex:

```cpp
while(flag)
{
    cout << "yes";
}
```

Spacing in Statements (2/2)
- Reserve spaces before and after the binary operators
  - -, +, -, *, /, %, +=, &&, ||, >, <, ==
- No space in front of unary operators
  - -, !, ++, --
- Ex:

`x=a<b?a:b;`

<table>
<thead>
<tr>
<th>Bad Style</th>
<th>Good Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=a&lt;b?a:b;</td>
<td>x=a&lt;b?a:b;</td>
</tr>
<tr>
<td>for (i=0; i&lt;10; i++) {if (a&gt;=b&amp;&amp;!c) {y*=i;}}</td>
<td>for (i=0; i&lt;10; i++) {if ((a&gt;=b)&amp;&amp;(c)) {y*=i;}}</td>
</tr>
</tbody>
</table>

Breaking Up Long Lines
- Long lines are hard to read and print
  - Limit length in 80 characters
  - Break up at low-precedence operators or next arguments
  - Left-justify the start of the first statement
- Ex:

```cpp
if ((long_expression_1) && (long_expression_2) && (long_expression_3)) {
    statements;
}
for (long_initialization;
long_condition;
long_update)
{
    statements;
}
```
Words about Comments

- Comments improve the readability of our programs ⇒ a form of etiquette (規矩)
  - May go up to 1/3 code lines for big projects
  - You compiler skips comments
- Block comment: /* …*/ ⇒ must be pair wise
- Line comment: //

```c
/* This program is to compute the cable bill */
int main () {
    statements;
}
```

```
for (i=0; i<5; i++) {
    if (i%2) // i is counter
        cout << "even";
    } // end of if
} // end of for
```

Commenting Guidelines

- Avoid redundant comments
  - Ex: i++; // i=i+1
- Modify your comment right after you modify your code
- Place your comment on top of to the right of your statement
  - Ex: // accumulate the sum
    ```c
    sum = sum + term;
    ```
- Too many comments instead disturb readers
  - At most 1/3

Summary (1/2)

- Use enum to create and manipulate your own data type
  - Syntax and rules to use enumerations
  - Arithmetic operations are not allowed
- Familiar with typedef statements
  - Alias to the existing datatype if using alone
- Using struct
  - A collection of values of different types
  - Two types of using struct
  - Declare, access by dot operation
  - Initialization and using struct in functions

Summary (2/2)

- Understand more options in g++
- Programming style includes:
  - Naming convention
  - Coding style ⇒ only introduce part of this
- Some programming guidelines for
  - Indent style
  - New lines
  - Spacing
  - Control struct with {}
  - Variable usage
  - Comments