Chapter 10.1 – 10.2

Simple Data Types: Built in and User-Defined

Built in Simple Types

- **DataType** - a specific set of data values (the domain) along with a set of operations on those values
- **Simple Data Types** - DataTypes where the domain is made up of indivisible data values (data values that do not have components that can be accessed individually)
- **Strings are not simple DataTypes** because each character in a string can be accessed - strings are a collection of characters
- Sizes of a variable are measured in multiples of the size of a char. The size of a char is 1 byte. Can use the integer value returning `sizeof(SomeType)` operator to obtain the number of bytes allocated for a specific DataType - this includes structures
Expressions

- **Assignment Expression** - a C++ expression with a value and the side effect of storing the expression value into a memory location
  - \( \text{alpha} = \text{beta} \times 5 \quad // \text{note - no semicolon} \)
  - An assignment is an expression, so it can be used anywhere an expression is allowed

- **Expression Statement** - A statement formed by appending a semicolon to an expression
  - \( \text{alpha} = \text{beta} \times 5; \)

Additional C++ Operators

- **Combined assignment operators**: +=, -=, *= and /= are add and assign, subtract and assign, multiply and assign and divide and assign respectively
  - Statement | Equivalent Statement
  - \( j *= 5; \) | \( j = j \times 5; \)
  - \( k -= 3; \) | \( k = k - 3; \)

- **Increment and decrement** - ++Var, Var++, --Var, Var--

- **Bit wise operators** - manipulate individual bits within a memory cell - can be very useful in certain situations
More Additional C++ Operators

• The Cast operation - explicit form of coercion to show that the type conversion is intentional. Two forms
  given: int intVar; float floatVar; long double myVar;

• \texttt{intVar = int(floatVar); // functional notation}
  • Looks like a function call
  • Can be used for DataTypes that consist of a single identifier only

• \texttt{intVar = (int)floatVar; // Prefix notation} - parentheses required
  • Parentheses surround the DataType not the expression
  • Can be used for DataTypes that consist of multiple identifiers
    \texttt{myVar = (long double) floatVar;}

Conditional Operator (the ?:) page 328 ch. 7

• A ternary (Three-operand) operator with the following syntax:
  • Expression1 ? Expression2 : Expression3
  • The computer evaluates Expression1
    • If the value of Expression1 is true, then the value of the entire expression is expression2
    • If the value of Expression1 is false, then the value of the entire expression is expression3
  • Examples:
    \texttt{min = (a < b) ? a : b; // replaces the following}
    \texttt{if (a<b) \quad // if-then-else code segment}
    \texttt{\quad min = a;}
    \texttt{else}
    \texttt{\quad min = b;}
    \texttt{y = ( x >= 0) ? x : -x; // y = absolute value of x}

See program UserDefinedTypes_01.cpp
for examples of additional C++ operators
**Working with Character Data**

- **Char is defined to be an integral type**, and has size of 1 byte. Char values are stored as integer numbers on the computer.
- Each computer uses a particular character set. Two common sets are ASCII (128 characters) and EBCDIC (256 characters).
- Computer cannot tell the difference between character data and integer data in memory
  ```cpp
  int someint = 97;
  char somechar = 97;
  cout << someint << " " << somechar << endl;
  97 a
  ```
- **DataType of a variable determines how it will be printed**

**Comparing Characters**

- Can use `<`, `<=`, `>=`, `>` for comparisons. Need to consider character sets. In ASCII, lowercase letters occupy 26 consecutive positions in the character set. In EBCDIC, they don’t.
- Better to take advantage of the is... functions in the header file cctype. (i.e. islower(char))
  ```cpp
  #include <cctype>
  ```
- To convert char numbers to real numbers: ‘3’ - ‘0’ = 3. Can use a char to obtain an input and then make the conversion
- Also in cctype are the functions toupper(char) and tolower(char) to change the case of a char accordingly
- Can access individual characters of a string by giving its position number in square brackets. The first character is in position 0:
  ```cpp
  StringObject[Position]
  ```
- See program UserDefinedTypes_02.cpp for examples involving char variables
User Defined Simple Types – Typedef Statement

- **Typedef Statement** - allows you to introduce (create) a new name for an existing type
  - **Syntax Template:** TypedefStatement
    ```c
    typedef ExistingTypeName NewTypeName;
    typedef int Boolean;  // makes Boolean a new name for int
    ```

User Defined Simple Types – Enumeration type

- **Enumeration Types** - a user-defined DataType whose domain is an ordered set of literal values expressed as identifiers
  - **Enumerator** - one of the values in the domain of an enumeration type
    - **Syntax Template:** EnumDeclaration
      ```c
      enum Name {Enumerator, Enumerator ...};
      ```
      - Enumerator
        ```c
        Identifier = ConstIntExpression
        ```
      - **ConstIntExpression** - an integer expression consisting of literal or named integer constants only
**Enum Datatype**

- `enum Month { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG};`
- Here `Month` is a new `DataType`, and JAN, et. al. are identifiers, and by default the first enumerator (JAN) has a value of 0, FEB has a value of 1, etc.
- Values can be specified, but there is rarely a reason to do so
  ```cpp
  enum Month {JAN = 1, FEB = 2 , ... };
  ```
- Remember - enumerators must be identifiers, and identifiers within the same scope must be unique. Since enumerators are identifiers, they must follow the rules for any C++ identifier
  ```cpp
  enum Days { "Sun", "Mon", ... }; // invalid - uses literals
  enum Hours { 1_Oclock, 2_Oclock, ... }; // invalid
  enum Hours {One_Oclock, Two_Oclock, ... }; // valid
  ```

**Enum Assignment**

- Enumerators are **identifiers**; they are not variables
- Enumeration `DataType` variable declarations:
  - `Month date;` // date is a variable of `DataType` Month
  - `Hours time;` // time is a variable of `DataType` Hours
  - `Month new_date;` // new_date is a variable of `DataType` Month
- Assignment:
  ```cpp
  date = JAN;  // invalid, Must use an Enumerator and not its value
  new_date = MAR;  // prevents an assignment of an out-of-range value
  date = 1;
  ```
enum Assignment

- For enum (enumeration) DataTypes:
  - implicit type coercion IS DEFINED from an enumeration type to an integral type
  - implicit type coercion IS NOT DEFINED from an integral type to an enumeration type

    SomeInt = MAR; // This is valid. MAR coerced to an int
    date = 3;  // this is invalid. Can’t coerce an int to an enumeration

enum Incrementation

- Must use an explicit type conversion - a type cast - to increment an enum variable

  Invalid increments:
  - date = date +1; // Error - can’t store an int into an enum
  - date + 1 is a valid expression - date is coerced to an integer, and the result is an integer value, the assignment causes the error
  - date++; // Error for the same reason above
  - for (date = JAN; date <= JUN; date++) // error due to date++

  Valid increments:
  - date = Month(date + 1); using explicit type coercion (type casting)
  - for(date = JAN; date <= JUN; date = Month(date + 1))
**enum Input and Output**

- Values of enumeration type **MUST BE input or output indirectly**
- **For input**, one strategy is to read in a string and then look at as many letters as are necessary to determine what it is - a `switch` statement with additional `if` statements will work
- **For output**, a `switch` statement is used to print a character string corresponding to the enumeration value
- Value returning functions can return enumeration values.

```cpp
Month find_month( string str) // start of function definition
{
    Month month_var;

    return month_var; // Or, could return an enumerator
    // i.e. return JAN;
}
```

**Named and Anonymous Types**

- **Named Type** - A user-defined type whose declaration includes a type identifier that gives a name to the type
  - `enum CoinType {PENNY, NICKEL, DIME, QUARTER};`
  - `CoinType change;` // change is a variable of type CoinType

- **Anonymous Type** - a user-defined type whose declaration does not have an associated type identifier
  - `enum {PENNY, NICKEL, DIME, QUARTER} change;`
  - change is a variable of anonymous type - no name provided for the type in the declaration

See Program UserDefinedTypes_03.cpp for examples of enumerated DataTypes
User-Written Header Files

- Can put information - i.e. user defined types into a file, and then include these files in programs that require the information without having to rewrite it every time.
- `enum Change {PENNY, NICKEL, DIME, QUARTER};`
- save that information in a file `Change.h`
- In a program requiring that enumeration type, use the preprocessor directive `#include`:

```
#include "Change.h"
```

Note that quote marks instead of `< >` is used. Also, the above requires that the file is located in the same directory as the program.

Or, you can specify the exact path to the file:

```
#include "/some/path/to/the/file/Change.h"
```

Type Coercion in Arithmetic and Relational Expressions

- In an expression, if two operands on either side of an operator are of different DataTypes, one of them is temporarily promoted to match the DataType of the other
- Step 1: Each `char`, `short`, `bool` or enumeration value is promoted to `int`. If both operands are now `int`, the result is an `int` expression
- Step 2: If step 1 still leaves a mixed type expression, the following precedence of types is used:
  - From lowest precedence to highest precedence
    - `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`
  - The value of the operand of the “lower” type is promoted to that of the “higher” type and the result is an expression of the higher type
Type Coercion in Assignments, Argument Passing and Return of a Function Value

- In general *promotion* from one type to another does not result in a loss of information
- However, *demotion* (conversion of a value from a “higher” type to a “lower” type) of data values can cause loss of information
  - Demotion from a longer integral type to a shorter integral type results in a discarding of the most significant bits of the binary number representation
  - Demotion from a floating-point type to an integral type causes truncation of the fractional part
  - Demotion from a longer floating-point type to a shorter floating-point type may result in a loss of digits of precision
- Read the Case studies pages 496-504
- Read Testing and Debugging, Coping with input errors and Summary - pages 504-506