Section 1.2
Basic Language Features

1. Overview
2. Terminology
3. Operators
4. Control and data structures
5. Variables and data types
6. Standard I/O streams
7. Functions
8. References

1.2.1 Overview

- History of C++
  - C++ is a general purpose object-oriented programming language
  - it was developed by Bjarne Stroustrup, starting in 1979
  - initially known as “C with classes”, i.e. object-oriented C

Overview (cont.)

- Characteristics of C++
  - it provides all the standard high-level OO functionality
    - polymorphism
    - generic programming using templates
    - operator overloading
    - multiple inheritance
    - exception handling
  - it also allows calls to low-level C functions
    - memory management operations, basic I/O
    - bit-wise operations
    - C libraries for TCP/IP sockets, for threads, etc.

1.2.2 Terminology

- Expression
  - a sequence of operations that resolve to a single value
  - examples:
    - `a + b - 3 * c`
    - `d * pow(g, a)`

- Statement
  - an expression terminated by a semi-colon
  - examples:
    - `tmpValue = a + b - 3 * c;`
    - what happens here: `d * pow(g, a);`
  - statements **also** resolve to a single value
    - value may be `void`, or it may be ignored

Terminology (cont.)

- Block
  - a sequence of statements between a pair of matching braces
  - examples:
    - functions
    - loops, `if`-statements
    - free-floating blocks!
  - spot the difference:

```
int x = 8, size = 5, i = 0;
while (i <= size) {
  x = i;
  ++i;
}
cout << x << endl;
```

```
int x = 8, size = 5, i = 0;
while (i <= size) {
  int x = i;
  ++i;
}
cout << x << endl;
```

Terminology (cont.)

- Scope
  - the part of the program where a variable can be used
- Block scope:
  - variables are declared inside a block
    - these are called *local variables*
  - any kind of block can have local variables
- Global scope (also called *file scope*)
  - variables are declared outside of *any* block
    - these are called *global variables*
  - industrial quality software **never** uses global variables

1.2.3 Operators

- Types of operators:
  - arithmetic: `+ - * / % ++ --`
  - relational: `== != < > <= >=`
  - logical: `&& || !`
  - bitwise: `~ & | ^ >> <<`
  - assignment: `= += -= *= /= %=`
  - conditional: `?:`

Operators (cont.)

- Characteristics of operators:
  - arity
    - number of operands
    - operators can be *unary*, *binary*, or *ternary*
  - precedence
    - order in which operators execute
    - remember BEDMAS?
  - associativity
    - order in which operators of the same precedence execute
    - associativity can be left-to-right, or right-to-left

1.2.4 Control and Data Structures

- Control structures
  - conditional
    - example: **`if-else`**
  - selective
    - example: **`switch`**
  - iterative
    - example: **`for`**, **`while`**, ~~**`do-while`**~~
  - jump
    - example: **`break`**, **`continue`**

Control and Data Structures (cont.)

- Data structures
  - C-style **`structs`**
    - in C++, they contain variables and functions, just like classes
    - by default, their members are *public*
    - this is **bad** software engineering

  - classes
    - like Java, they contain variables and functions
    - by default, their members are *private*
    - using classes follows the *principle of least privilege*
    - this is **good** software engineering
  - **do not** use **`structs`** in assignments or tutorials

1.2.5 Variables and Data Types

- Program memory is like a bookshelf full of boxes
  - the boxes have different sizes
  - each box has a label or name
  - each box contains something
  - each box is in a specific location on the shelf

- Variables are like those boxes
  - they come in different sizes, based on data type
    - examples: **`int`**, **`double`**, **`char`**
  - they have a name
  - they contain something, called a *value*
  - they have a location in memory
    - represented as a memory *address* where the value is stored

Variables and Data Types (cont.)

- Characteristics of variables:
  - name
    - this is the identifier that the program uses to access the content
  - value
    - this is the content of the variable
  - data type
    - this determines the number of bytes required in memory
  - location in memory
    - this is the address in memory where the value is stored

Variables and Data Types (cont.)

- Data types
  - primitive (built-in)
    - examples: **`int`**, **`float`**, **`double`**, **`char`**, **`bool`**
  - user defined
    - defined by programmer (you!)
    - these are your *classes*
  - memory address
    - example: pointers

1.2.6 Standard I/O Streams

- C++ library stream objects
  - declared in the **`iostream`** library
  - encapsulated in the **`std`** namespace

- **`cout`**
  - object of class **`ostream`**, used for standard output

- **`cin`**
  - object of class **`istream`**, used for standard input

- **`cerr`**
  - object of class **`ostream`**, used for standard error

1.2.7 Functions

- In C++, there are two types of functions:
  - *global functions*
    - they are defined outside of any block
    - they can be called from any function and any class in the program
    - example: **`main()`**
  - *member functions*
    - they are defined within a class
    - they may be called on an object of that class
    - *static* member functions can also be called on the class itself
    - some OO languages call these functions *methods*

Functions (cont.)

- Important terminology:
  - function *declaration*
    - this is the function prototype
  - function *implementation*
    - this is the code between the braces
    - it's the "body" of the function
- Member functions:
  - declaration and implementation are stored in **different files**

Function Design

- Characteristics of *correctly designed* functions:
  - they take data into the function, do something, return a result
    - two ways to return results: return value or using parameters
  - they are single-purpose
    - they have a single goal, they do *one thing only*
  - they encapsulate (hide) their functionality
    - other programmers know **what** function does, not **how** it does it
  - they should be reusable
    - in the same program, and in other programs

Function Design (cont.)

- Return values:
  - they are often used to indicate success or failure
  - they are used to return results from **very simple** functions
    - mostly “getter” functions
  - normally, results are returned using an ***output parameter***
    - this allows multiple values to be returned to calling function
    - … more on this coming up…

Function Design (cont.)

- Parameters
  - they are treated as local variables inside a function

- Parameter modifiability:

  - pass-by-value
    - the parameter value is *copied* from the calling function
    - the function works on the local copy

  - pass-by-reference
    - parameter is memory address of variable declared in calling function
    - this allow variables in calling function to be modified
    - in C++, this can be done using *pointers* or *references*

Function Design (cont.)

- Every parameter has a role to play in the function:
  - input parameter
    - this is a value required by function to do its work
    - it is data sent by calling function
  - output parameter
    - this stores a result of the function doing its work
    - this is how results are correctly “returned” to the calling function
    - in assignments, use this technique instead of return values!
  - input-output parameter
    - this is both an input parameter, and an output parameter
    - it contains a value that is needed by the function
    - it gets modified by the function

1.2.8 References

- To pass parameters by reference:
  - we need to use a memory address as a variable
  - the best way to do this is with *pointers*
    - pointers are very powerful, but can be difficult to learn
    - ... more on pointers later ...
  - an alternative to pointers: *references*
    - they are used for pass-by-reference
    - they are not very versatile
    - they are not a language feature in C, only in C++

References (cont.)

- What is a reference not?
  - a reference is **not** a real variable! it does not occupy memory
  - it must be assigned a value on declaration
  - its value can never change

- So what is it?
  - a **binding**, or an alias, for an existing variable
  - declaring and initializing a reference creates a bond between:
    - the reference name
    - an existing variable
  - the bond is unbreakable
  - most common usage is for passing parameters by reference