Introduction
CS 201

This slide set covers the basics of C++. You should know most of these basic concepts from your Java classes. To refresh your knowledge, you may read Chapters 2, 3, 4, and 5 from your Deitel & Deitel book.

Let's convert the following simple Java program to C++

```java
import java.util.*;

public class GradeBook {

    public void displayMessage( int courseNo ) {
        System.out.println( "Welcome to " + courseNo );
    }

    public static void main( String [] args ) {
        int cno;
        Scanner scan = new Scanner( System.in );
        System.out.print( "Enter course no: " );
        cno = scan.nextInt();
        displayMessage( cno );
    }
}
```

C++ Basics

- Each program should have one main function (which is a *global, non-member function*) and may have one or more classes

- A class is a user-defined type to create objects
  - Class definition tells the compiler the class' data members and member functions
  - (Default) Access-specifier `private` makes a data member or a member function accessible only to the member functions of the same class
  - Access specifier `public` indicates that a data member or a member function is accessible to other functions and the member functions of other classes (through the dot operator)

- `#include` instructs the C++ preprocessor to replace directive with a copy of the contents of the specified file

```cpp
#include <iostream>
using namespace std;

class GradeBook {
public:
   void displayMessage( int courseNo ) {
      cout << "Welcome to " << courseNo << endl;
   }
};
int main() {
   int cno;
   GradeBook G1;                      // Do not use the new operator; it is
                                      // an invalid use (different than Java)
   cout << "Enter course no: ";
   cin >> cno;
   G1.displayMessage( cno );
   return 0;
}
```

Declarations

- Variables are declared to store values in the computer's memory

- **Data members** (variables in a class definition)
  - They exist throughout the life of a object
  - Each object maintains its own copy of the non-static data members

- **Local variables** (those declared in a function)
  - They cannot be used outside that function body
  - The automatic (non-static) local variables are lost when the function terminates

- **Global variables** (we will see them later)

```cpp
class GradeBook {
public:
   void dummy() {
      int a, b;          Local
      char c;            variables
      double d, e;
   }
private:
   int courseNo, a;      Data
   double avgGPA;        members
};
int main() {
   int x, y;             Local
   GradeBook G1, G2;     variables

   return 0;
}
```

Data types

```
long double
double
float
unsigned long int(or unsigned long)
long int                (or long)
unsigned int            (or unsigned)
int
unsigned short int     (or unsigned short)
short int              (or short)
unsigned char
char
bool
```

Promotion hierarchy for fundamental data types

For instance, **short**, **long**, **int**, and **char** are integer data types that differ with respect to the number of bytes that they use to represent a value

Minimum standards:

- **short**: 2 bytes
  (minimum range -32,768 to 32,767)

- **long**: 4 bytes
  (minimum range -2,147,483,648 to 2,147,483,647)

- **int**: either `short` or `long`

- **char**: 1 byte
  (minimum range -128 to 127)

Let's extend our `GradeBook` *class by adding*

- **Data member** and its **get/set** functions
    - Public member functions that allow clients of a class to access its private data member
    - They allow the creator of the class to control how clients access the class' private data
    - As a rule of thumb, the data members should be declared as private where most of the member functions are declared as public (it is appropriate to declare certain member functions as private provided that they are accessed only by the other member functions of the same class)

- **Constructors** (functions called when an object is created)

- **Overloaded functions** (those with the same name but different signatures)

Constructors

- They are the functions that are called when an object is created
    - The function name should be the same with the class name
    - It cannot return any value (not even `void`)

- If no constructor is explicitly coded, the default constructor provided by the compiler will be called when an object is created
    - This constructor, which is provided by the compiler, just constructs the object's data members without any initialization
    - This corresponds to just allocating memory (with a garbage value) for the data members of primitive types and calling the default constructors for those that are objects of other classes
    - It does not have any parameters
    - *Although data members can be initialized later after the object is created, it is a good software engineering practice to ensure that an object is fully initialized before the client code invokes the object's member functions. In general, you should not rely on the client code to ensure that an object gets initialized properly.*

Constructors

- If at least one constructor is explicitly coded, the matching one will be called upon an object's declaration
  - The default constructor provided by the compiler cannot be called anymore
  - You should also code the default constructor (with no argument) if you need to call it

```cpp
class GradeBook{
public:
  GradeBook( int no ){
    setCourseNo( no );
  }
  GradeBook( ){
    courseNo = 0;
  }
  ...
private:
    int courseNo;
};
```

OR

```cpp
class GradeBook{
public:
  GradeBook( int no = 0){
    setCourseNo( no );
  }
  ...
private:
    int courseNo;
};
```

```cpp
int main(){
  GradeBook G1, G2(201);
  GradeBook G3(101);

  // invalid declaration in C++
  // the new operator can be
  // used when G4 is an object
  // pointer
  // GradeBook G4 = new GradeBook;

  return 0;
}
```

allProgram.cpp

```cpp
#include <iostream>
#include <string>
using namespace std;

class GradeBook {

public:
  GradeBook( int no ) {
    setCourseNo( no );
  }
  GradeBook( ) {
    courseNo = 0;
  }

  void setCourseNo( int no ) {
    if ((no >= 100) && (no <= 999))
      courseNo = no;
    else{
      cout << "invalid grade" << endl;
      courseNo = 0;
    }
  }
  int getCourseNo() {
    return courseNo;
  }

  // file continues
  void displayMessage( string msg ) {
    cout << msg << " ";
    cout << getCourseNo() << "!" << endl;
  }
  void displayMessage() {
    cout << "Welcome to CS ";
    cout << getCourseNo() << "!" << endl;
  }

private:
    int courseNo;

}; // end of class

int main( ){
  GradeBook gb1( 101 ), gb2;

  gb2.displayMessage();
  gb2.setCourseNo( 201 );
  gb2.displayMessage("Hello CS");

  return 0;
}
```

*Let's place a class in a separate file for **reusability***

- `.cpp` file is known as a source-code file

- Header files (generally with `.h` filename extensions)
  - Separate files containing class definitions
  - Allow the compiler to recognize the classes when used elsewhere (of course, provided that the header file is included)

- Driver files (with the `main` function)
  - To run your program
  - And also to test the software

Separating interface from implementation

- Interface of a class describes *what* services class' clients can use without revealing *how* these services are implemented

- It is a good programming practice to separate interface from implementation
  - Client code does not need to know implementation (*implementation details are hidden*)
  - Client code should not break if implementation changes, as long as interface stays the same

- **Header file** includes the class' interface that gives the class definition
  - Prototypes of public member functions for the client
  - Prototypes of public and private member functions and also data members for the compiler

- **Source file** includes the implementation of class' member functions
  - Implementations are placed outside the class definition
  - Binary scope resolution operator (::) is used to tie each member function to the class definition

`#include` preprocessor directive

- It instructs the C++ preprocessor to replace directive with a copy of the contents of the specified file

- Angle brackets are used to include a file in the C++ Standard Library
  - Preprocessor looks for the specified file only in the C++ Standard Library directory

- Quotes are used to include a user-defined header file
  - Preprocessor looks for the specified file first in the current directory
  - If it does not find the file, it searches the file in the C++ Standard Library directory

GradeBook.h

```cpp
#ifndef __GRADEBOOK_H
#define __GRADEBOOK_H

#include <string>
using namespace std;

class GradeBook {
public:
  GradeBook( int no );
  GradeBook( );
  void setCourseNo( int no );
  int getCourseNo( );
  void displayMessage( string msg );
  void displayMessage();

private:
  int courseNo;
}; // end of class

#endif
```

program.cpp

```cpp
#include "GradeBook.h"

int main() {
  GradeBook gb1( 101 ), gb2;

  gb2.displayMessage();
  gb2.setCourseNo( 201 );
  gb2.displayMessage("Hello CS");

  return 0;
}
```

GradeBook.cpp

```cpp
#include <iostream>
using namespace std;
#include "GradeBook.h"

GradeBook::GradeBook( int no ) {
  setCourseNo( no );
}
GradeBook::GradeBook( ) {
  courseNo = 0;
}
void GradeBook::setCourseNo( int no ) {
  if ((no >= 100) && (no <= 999))
    courseNo = no;
  else {
    cout << "invalid grade" << endl;
    courseNo = 0;
  }
}
int GradeBook::getCourseNo() {
  return courseNo;
}
void GradeBook::displayMessage( string msg ) {
  cout << msg << " ";
  cout << getCourseNo() << "!" << endl;
}
void GradeBook::displayMessage() {
  cout << "Welcome to CS ";
  cout << getCourseNo() << "!" << endl;
}
```
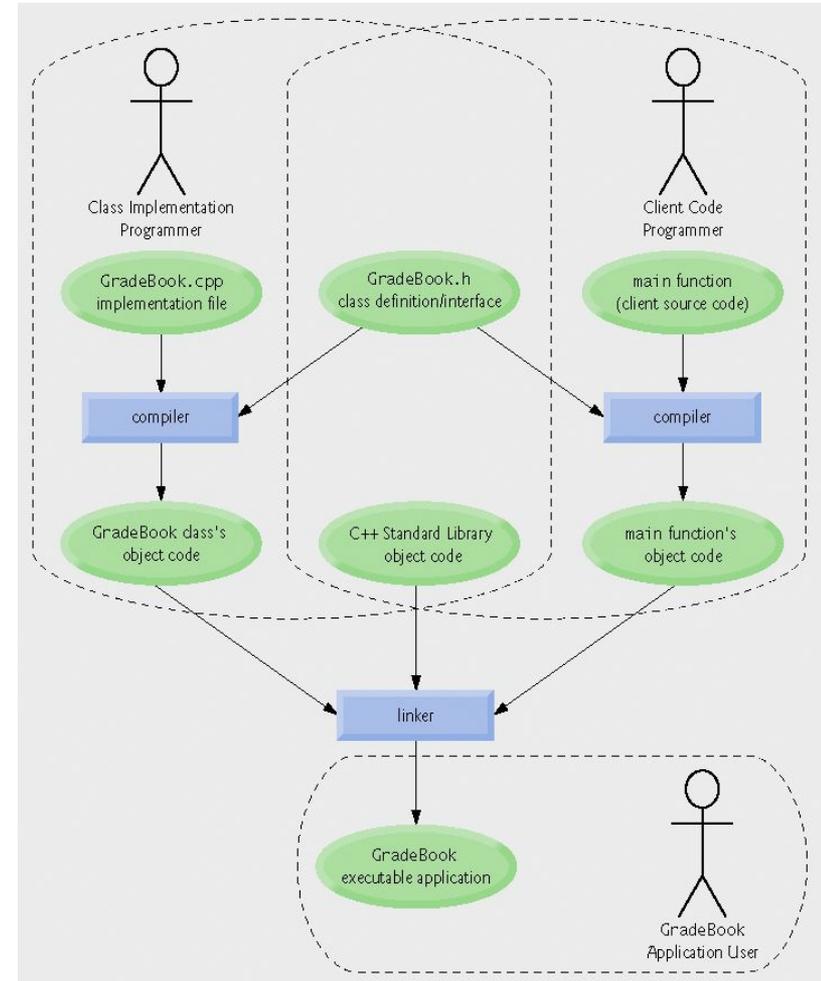
Compilation and linking

- Source-code file is compiled to create its object code (class' object code)
  - Class implementer only needs to provide the header file and the object code to the client

- Client must compile his/her own code by including the provided header file
  - By including it, the compiler knows how to create and manipulate objects of the class

- Executable is created by linking
  - Object code of the client code,
  - Object code of the class, and
  - Object code of any C++ Standard Library code used in the application

Compilation and linking processes that produce an executable

Control structures

- There are three types of control statements
  - Sequence statements, which are executed sequentially
  - Selection statements: **if, if...else, switch**
  - Repetition statements: **for, while, do...while**

- These statements are combined by either *sequencing* or *nesting*

Example:

Write a class function that takes grades from the user and returns
their average to the caller

First draft (may have some bugs)

```cpp
double GradeBook::computeAvg() {
  int grade, sum, count;

  cout << "Enter grades (negative value to stop): ";
  cin >> grade;

  while ( grade >= 0 ) {
    sum += grade;
    count++;
    cout << "Enter grades (negative value to stop): ";
    cin >> grade;
  }
  return sum / count;
}
```

Example:

Write a class function that takes grades from the user and returns
their average to the caller

Final version (bugs are fixed)

```cpp
double GradeBook::computeAvg() {
  int grade, sum = 0, count = 0;

  cout << "Enter grades (negative value to stop): ";
  cin >> grade;

  while ( grade >= 0 ) {
    sum += grade;
    count++;
    cout << "Enter grades (negative value to stop): ";
    cin >> grade;
  }
  if ( count )
    return static_cast<double>( sum ) / count;
  else
    return 0;
}
```

Remarks related with this example and more

- If it is not done explicitly, local variables (and also data members) are uninitialized, containing **"garbage" values**

- When two operands of the division operator are integers, the result is truncated (**integer division**)
  - Fractional part of the resulting quotient is lost

- **Unary cast operator** performs an explicit conversion, which creates a temporary copy of its operand with the specified data type
  - Example: `static_cast<double>( total )`

- An implicit conversion will occur when the arguments are of different types
  - When the operands of a binary operator or an assignment operator have different data types
  - When the data types of a function parameter and its argument are different

Remarks related with this example and more

- Integer values can be used to represent a boolean value
  - For compatibility with C, which does not have a boolean data type
  - Any nonzero value (compilers typically use 1) represents `true` and 0 represents `false`

- It is the programmer's responsibility not to confuse the equality (==) and assignment (=) operators
  - This will NOT produce a compile-time error

- Counting loops should not be controlled with floating-point values
  - This may result in imprecise counter values and inaccurate tests for termination, since some floating-point values approximately represent their real counterparts
  - e.g., fl(0.1) approximate, fl(0.5) exact

- Increment and decrement operators
  - Pre- and post-increment (decrement) operators will have different effects when they are used in other statements

Example:

Write a code fragment that calculates and displays the factorial of a given positive number *N*

```cpp
int N, product;

cin >> N;
product = 1;
for ( int i = 1; i <= N; i++ )
  product *= i;

cout << "Factorial of " << N << " is " << product << endl;
```

Example:

Write a code fragment that calls a function (for example, factorial function) until the user wants to stop

```cpp
char ans;
int N;

do {
  cout << "Enter a non-negative integer: ";
  cin >> N;
  factorial( N );

  do {
    cout << "Do you want to stop (Y or N)? ";
    cin >> ans;
  } while ( ans != 'Y' && ans != 'y' &&
            ans != 'N' && ans != 'n' );

} while ( ans == 'N' || ans == 'n' );
```

Loop body always executes at least once

Example:

Write a code fragment that takes an integer grade and displays its letter equivalent, calculated by the following rules: A: 90-100, B: 80-89, C: 70-79, D: 60-69, F: 0-59

```cpp
int grade;
char letter;

cin >> grade;
if ( grade < 0 || grade > 100 )
    cout << "Invalid grade\n";
else {
  switch ( grade / 10 ) {          ← This expression should be of an integer
    case 10:                          (long, int, short, char) data type
    case 9: letter = 'A'; break;
    case 8: letter = 'B'; break;
    case 7: letter = 'C'; break;
    case 6: letter = 'D'; break;
    default: letter = 'F'; break;  ← Default is optional and executes if no
  }                                    matching case label is found
  cout << "Your letter grade is " << letter << endl;
}
```

break and **continue** statements

- They alter the flow of control

- **break** statement
  - Can be used for **while, do...while, for**, and **switch** statements
  - Causes an immediate exit from a control structure
  - For the **switch** statement without **break**, execution will fall to the next case label

- **continue** statement
  - Can be used for **while, do...while**, and **for** statements (just for loops)
  - Skips remaining statements in the loop body

What are the outputs of the following two code fragments?

```
for ( int i = 1; i <= 5; i++ ) {
   if ( i == 3 )
      break;
   cout << i << endl;
}
cout << "done" << endl;
```

```
for ( int i = 1; i <= 5; i++ ) {
    if ( i == 3 )
       continue;
    cout << i << endl;
}
cout << "done" << endl;
```