# Advanced Programming Concepts with C++ CSI2372 – Fall 2018

Jochen Lang
EECS, University of Ottawa
Canada

Université d'Ottawa | University of Ottawa

uOttawa

L'Université canadienne
Canada's university

uOttawa.ca

# This lecture

Java in C++

- **Basic Object-oriented C++**
  - Strongly-typed Enumerations
  - Operators, Ch. 4.1-4.9
  - Selection and Iteration Statements, Ch. 1.4, 5.3-5.5
  - Static casts, Ch. 4.11.3-5.12.6
  - Overview of std::string
  - Introduction to std::array and std::vector

# Strongly Typed Enumeration Example

```
#include <iostream>

enum class ID : unsigned long long {
  Zero=0ULL, Other, Large=2346781693637789ULL
};

int main(){
  ID num = ID::Zero;
  if (num == ID::Zero) {
    num = ID::Large;
  }
  std::cout << static_cast<unsigned long long>(num)
    << std::endl;
  return 0;
}
```

# Why Enumerations?

- **Could just use `const`, i.e., `const int Red=0;`**
  - Readability
  - Ease of modifying the numeric representations
  - Strong typing i.e.,
    - Value `Red` cannot be assigned to a variable of type `Day`.
- **Limitations of enum prior to C++11**
  - Underlying type is always an `int`
  - `enum` types implicitly convert to `int`
  - Unscoped `enum` definitions end up in the surrounding scope
    - All the above is addressed in C++11 by enum class (strongly typed enumerations)

# Operators (Ch. 4)

  - Arithmetic operators
  - Relational and logic operators
  - Bitwise operators
  - Assignment operators
  - Others
- **Operator properties**
  - Unary, binary and ternary operators
  - Operators have a precedence and associativity (LR and RL)

# Arithmetic Operators

- In general … close to Java

```
double dVal=21.0, dDiv=3.14;
double dRes = dVal/dDiv;
int iVal=21, iDiv=5;
auto iMod = iVal%sVal;
auto iRes = iVal/iDiv;
```

- Be aware:
  - Mixing types (more on type conversion later)
  - Integer division and modulo operator
    - C/C++ has signed and unsigned integral types (except for boolean)

# Logic Operators

- In general … close to Java
- Be aware: bool values can be converted to arithmetic types and vice versa
  - true has a value of 1
  - false has a value of 0

```
int iVal = 5;

if ( iVal == true ) {
  std::cout << "iVal == true" << std::endl;
}
if ( iVal ) {
  std::cout << "iVal is true" << std::endl;
}
```

# Operator Precedence

- **Table of Precedence: Lippman, pp.166/167**

  Operator precedence and associativity (LR and RL) is colour-coded.

  1. `::` (scoping: global, class, namespace)
  2. `()` `[]` `->` (member select) `.` (member select)
  3. `++` (postfix) `--` (postfix) typeid() explicit casts
  4. `++` (prefix) `--` (prefix) ! ~ (bitwise complement)
  5. - (unary) + (unary) * (dereference) & (address of) sizeof new new[] delete delete[] noexcept() (C++11)
  6. `->*` (ptr to member select) `.*` (object to member select)
  7. `*` (multiply) `/` `%`
  8. + -
  9. `<<` `>>`

# Operators (cont'd)

10. `<` `<=` `>` `>=`
11. `==` `!=`
12. `&` (bitwise AND)
13. `^` (bitwise XOR)
14. `|` (bitwise OR)
15. `&&`
16. `||`
17. `?` `:` (conditional)
18. `=` `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `|=` `^=`
19. `throw`
20. `,`

# Operator Precedence Examples

```
int iVal = 7, oiVal = 3, rVal = 13;

rVal += 2 + 3 * 8 / 4 + 2;
rVal = ++iVal / oiVal--;
rVal = iVal << 2 >> 4 / 3;
rVal = (iVal & 5 || oiVal-- && 1) + 3;
rVal = iVal = oiVal = 0;
```

- **Note: Precedence defines grouping not order of evaluation**
- **Rule of Thumbs:**
  - If in doubt use parentheses.
  - Avoid relying on the order of evaluation.

# Selection Statements: Examples

- **Decision statements**
  - if else
  - switch
  - initializer allowed in selection statements with C++17
  - ~~goto~~

```
if (counter == 1) {
    result = myFunction( x );
    counter++;
}
switch (auto k=1.51; counter) {
case 0:
    x = 3.0*k;
    y = 1.5;
    break;
case 1:
    x = 8.0*k;
    y = 9.5;
    break;
default:
    x = -1.0; y=-1.0;
}
```

# Iteration (Loops)

- **Control Statements**
  - range-based for
  - for loop

```
for (auto val:elements) {
    auto result = myFunction(val);
    resultSum += result;
}

for (int i=0; i<last; i++) {
    auto result = myFunction(elements[i]);
    resultSum += result;
}
```

# Iteration (Loops)

- **Control Statements**
  - while loop
  - do while loop
  - break and continue

```
do {
   auto element = myClass.getNextElement();
   if ( element == -1 ) break;
} while ( element != searchElement );

auto keepGoing = true;
while ( keepGoing ) {
    myClass.update();
    auto result = myClass.evaluate();
    if (result == -1) keepGoing = false;
}
```

# Implicit vs. Explicit Type Conversion

- **Implicit type conversion**
  - Applied by the compiler to built-in and class types
- **Occurs**
  - Operands with mixed types
  - Conversion to bool
  - Assignment to variable
  - Function calls
  - Const conversion, enumeration, conversion of library types
- **Explicit type conversion by Casting**
- **Be aware: Conversions are a rich source of errors!**

# Static Cast

```
int iVal; double dVal;
iVal = (int) dVal;
iVal = int (dVal);
```

- **Old-style casts**
  - Similar syntax than Java
  - Avoid: Use named cast operators instead!
- **Named Casts**
  - static_cast
    - Used to signal intentional conversion
    - Avoid compiler warning for loss of precision

```
char cVal; double dVal;
cVal = static_cast<char>(dVal);
```

  - Other named casts:
    - `reinterpret_cast const_cast dynamic_cast`

# Strings

- **C++ strings in namespace std**
  - Class with similar use than in Java
  - Dynamic memory management
  - Methods to work with strings
  - Operators for string manipulation

- **Use whenever possible over old style c-strings!**

# C++ `string` Class

- **Defined in string**
  - Commonly used operators

    `=  +  []  >>  <<  >  <  !=`
  - Commonly used methods

    `find compare insert length c_str`

```
#include <string>
using namespace std::string;
string s1 = “Not a sentence”;
string s2(“This is”);
s2 += s1;
s2.insert(7,” ”);
s2.replace(8,1,”n”);
string s3{" in C++11"};
cout << s2 << s3 << endl;
```

# Introduction to `std::array`

- **Fixed size Array `std::array`**
  - Need to `#include <array>`
  - `std::array` are not initialized, they only aggregate the underlying type and can be brace initialized
  - `std::array` can be copied, assigned and compared
  - `std::array` does not cause any performance overhead

# Example: Using `std::array` with fundamental data types

```
#include <array>
#include <iostream>

void manipulatePrint( std::array<int,10> iArr_copy ) { … }

int main( int argc, char* argv[] ) {
  array<int,10> iArr; // Uninitialized array of size 10  int
  // loop over the elements and set them to their rank
  // using an iterator
  for (auto iter = iArr.begin();iter != iArr.end();++iter) {
    *iter = num++;
  }
  array<int,10> oIArr = iArr; // Copy to another array
  if ( oIArr == iArr ) {…}  // Equal compare the arrays
  if ( oIArr != iArr ) {…} // Not equal compare the arrays
  manipulatePrint( iArr ); // Pass the array by value
  return 0; }
```

# Introduction to `std::vector`

- **Growable Array `std::vector`**
  - Similar to `ArrayList` or `Vector` (deprecated) in Java
  - Vectors adjust their size based on the number of element stored in the vector
  - Vectors can be copied, assigned and compared
  - Vectors offer same random (constant time) access than arrays
  - Vectors are containers and not just aggregates, e.g., they have additional constructors

# Example: Using `std::vector` with fundamental data types

```
#include <vector>
#include <iostream>

void manipulatePrint( std::vector<int> copy_iVec );

int main( int argc, char* argv[] ) {
  vector<int> iVec(10,0); // int vector of size 10
  // loop over the elements and print
  for ( vector<int>::iterator iter = iVec.begin();
        iter != iVec.end(); iter++ ) {
     std::cout << *iter << std::endl;
  }
  vector<int> oIVec = iVec; // Copying vector
  if ( oIVec == iVec ) { … } // Equal compare vectors
  if ( oIVec != iVec ) { … } // Not equal compare
  manipulatePrint( iVec ); // Pass the vector to a function
  return 0; }
```

# Next Lecture

Java in C++

- **Basic Object-oriented C++**
  - Classes, Ch. 2.6 , (7.1)
    - Example: BoundingShape
  - Construction
    - Method overloading, Ch. 6.4, 7.3
    - Default parameters, Ch. 12.4.2, (7.4.1)
  - Constructor types, Ch. 7.5, 13.1.1
  - Destruction 7.1.5,13.1.3