Advanced Programming Concepts with C++
CSI2372 – Fall 2019

Jochen Lang &
Mohamed Taleb
EECS

uOttawa

L'Université canadienne
Canada's university

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>
using namespace std;

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

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

```
2346781693637789
```

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)
    - Example: **enum** { aaa, bbb, ccc = 25, ddd, eee, fff = 1, ggg, hhh = fff + ccc }; //aaa = 0, bbb = 1, ccc = 25, ddd = 26, eee = 27, fff = 1, ggg = 2, hhh = 26

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)

downcast from b2 to d successful

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%iVal;
auto iRes = iVal/iDiv;
```

dRes = 6.6878980891719 75

iMod = 0

iRes = 4

- 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;
}
```

```
iVal is true
```

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;
```

```
23
2
16
4
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
  - Exp.:
    - **signed char** or **signed short** can be converted to **int**;
    - **unsigned char**, **char8_t** (since C++20) or **unsigned short** can be converted to **int** if it can hold its entire value range, and **unsigned int** otherwise;
    - **char** can be converted to **int** or **unsigned int** depending on the underlying type: signed char or unsigned char
- **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

static_cast<Type>(expression)
* General-purpose type casting

- **Old-style casts**
  - Similar syntax than Java
  - Avoid: Use named cast operators instead!

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

- **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`

reinterpret_cast<Type>(expression)
* Implementation-dependent casting

const_cast<Type>(expression)
* Cast-out “constantness”

dynamic_cast<Type>(expression)
* Downcasting from a superclass to a subclass

Examples

-
  ```
  int i = 5;                              // i is not declared const
  const  int& refci = i;
  const_cast <int&> (refci) = 6;          // OK: modifies i
  cout << "i = " << i << '\n’;            // i = 6
  ```

-
  ```
  struct S1 : S { };                      // standard-layout
  S1 s1 = { };
  auto p1 = reinterpret_cast<S*>(&s1); // value of p1 is "pointer to the S sub-object of s1"
  auto i = p1->x;
  p1->x = 1;
  ```

-
  ```
  class Parent { virtual ~Parent() { } };
   class Child : Parent { virtual void name() { } };
  int main() {
   Parent* p1 = new Parent;
   if(Child* c = dynamic_cast<Child*>(p1)) {
       cout << "downcast from p1 to c successful \n";
       c->name(); }
   Parent* p2 = new Child;
   if(Child* c = dynamic_cast<Child*>(p2)) {
       cout << "downcast from p2 to c successful \n";
       c->name(); }   delete p1; delete p2   }
  ```

```
downcast from p2 to c successful
```

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, substr, swap, replace, copy, assign, etc.`

```
#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;
```

This is not a sentence in C++11

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: `array` with fundamental data types

```
#include <iostream>
#include <iterator>
#include <array>
using namespace std;
void manipulatePrint(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++; // iter is the iterator position
  }
  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
  - Commonly used methods

    `empty, size, max_size, resize, begin, end, rbegin, rend, capacity, etc.`

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

```
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;
void manipulatePrint(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++ ) {
       cout << *iter << 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; }
```

Declaring an iterator to a vector

Next Lecture

Java in C++

- **Basic Object-oriented C++**
  - Classes, Ch. 2.6 , (7.1)
    - Example: Point2D
  - Construction
  - Constructor types, Ch. 7.5
  - Destruction 7.1.5