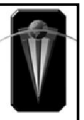
Advanced Programming Concepts with C++
CSI2372 – Fall 2013

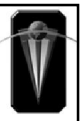
Jochen Lang
EECS, University of Ottawa
Canada



This lecture

More C++ Fundamentals

- Operators, Ch. 4.1-4.9 (Deitel 4.11,4.12, 5.8, App. A)
- Selection and Iteration Statements, Ch. 1.4, 5.3-5.5 (Deitel 4.5-4.7, 5.4-5.7)
- Scope, Ch. 5.2, (18.2.3) (Deitel 6.10)
- Modifiers, Ch. 2.4 (Deitel 6.9)
- Type conversions, Ch. 2.1.2, 4.11-4.12.2 (Deitel App. C)
- Static casts, Ch. 4.11.3-5.12.6 (Deitel 4.9,4.12)

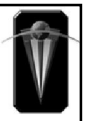


Operators (Ch. 4)

- 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

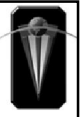
- In general … close to Java
- Be aware: Mixing types

```
int iVal=21;
double dVal=3.14;
int iRes = iVal/dVal;
double dRes = iVal/dVal;
```

- Be aware: Integer division and modulo operator
  - *C/C++ has signed and unsigned integral types (except for boolean)*

```
int iVal 21;
unsigned short sVal 5;
iVal/sVal;
iVal%sVal;
iVal/-sVal;
iVal%-sVal;
```

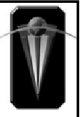
Logic Operators



- 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;
}
```

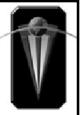

Bitwise Operators



- In general … close to Java
- Operators will have different effects on signed and unsigned integral types
  - *Example*

```
char cVal = -1;
unsigned char ucVal = 255;

ucVal &= cVal;
cVal &= ucVal;

ucVal >>= 1;
cVal >>= 1;
```


Aside: `Bitset` Objects



`Bitset` objects are the better choice than bitwise operators with integral types

- Not limited in size by type sizes
- Direct operations for setting and unsetting bits
- Example

```
bitset<52> attendanceLecture1;

attendanceLecture1.set(12);    // set bit 12
attendanceLecture1.unset(13);  // unset bit 13
```


Operator Precedence

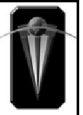


Table of Precedence: Lippman, pp.166/167

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

1 ::(scoping: global, class, namespace)

2 () [] -> (member select) . (member select)

3 ++(postfix) --(postfix) typeid() (named casts)

4 ++(prefix) --(prefix) ! ~ (bitwise complement) -(unary) +(unary) *(dereference) &(address of) type sizeof new new[] delete delete[] noexcept() (C++11)

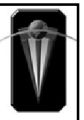
5 ->* (ptr to member select) .* (object to member select)

6 * (multiply) / %

7 + -

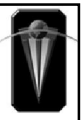
8 << >>

Operators (cont'd)



9 < <= > >=
10 == !=
11 & (bitwise AND)
12 ^ (bitwise XOR)
13 | (bitwise OR)
14 &&
15 ||
16 ? : (conditional)
17 = += -= *= /= %= >>= <<= &= |= ^=
18 throw
19 ,

Operator Precedence Examples



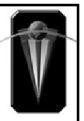
```
int iVal = 7, oiVal = 3, rVal = 13;

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

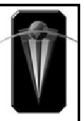


Decision statements

- if else
- switch
- ~~goto~~

```
selection statement :
   if (expression) statement
   if (expression) statement
      else statement
   switch (expression) statement

statement:
   labeled statement
   expression statement
   compound statement
   selection statement
   iteration statement
   jump statement
   declaration statement
```


Selection Statements: Examples

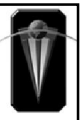


Decision statements

- if else
- switch
- ~~goto~~

```
if (counter == 1) {
   result = myFunction1( x );
   counter++;
}

switch (counter) {
case 0:
   x = 3.0;
   y = 1.5;
   break;
case 1:
   x = 8.0;
   y = 9.5;
   break;
default:
   x = -1.0; y=-1.0;
}
```

Iteration (Loops)

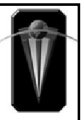


Control Statements

- for loop
- while loop
- do while loop
- break and continue

```
iteration statement :
    while (expression) statement
    do statement while (expression)
    for (for-init-statement
        expression_opt; expression_opt)
        statement

for-init-statement:
    expression statement
    declaration statement
```


Iteration (Loops)

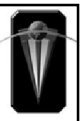


Control Statements

- for loop
- while loop
- do while loop
- break and continue

```
for (int i=0; i<lastNo; i++) {
    result[i] = myFunction(x[i]);
    resultSum += result[i];
}

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

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


Scope of Names



Local or block scope

- A name declared inside a block is accessible from the point of declaration to the end of the block.

Global (file) scope

- A name declared outside any function can be accessed inside the file from the point of declaration.

Class scope

- A name of a class element is local to the class, i.e., can only be accessed inside the class or must be used together with . -> or ::

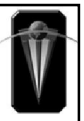
Function scope

- Only for labels; accessible everywhere in the function

Prototype scope

- Names are only accessible in the prototype declaration

Example



```
float PayRate = 1.5;
float CalculateWage(float);

int main( void )
{
float hours;
float pay = CalculateWage(hours);
return 0;
}

float CalculateWage( float workHours )
{
float res = workHours * PayRate;
return res;
}
```

Global scope

- PayRate
- CalculateWage

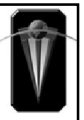
Local scope (inside main)

- hours
- pay

Local scope (inside CalculateWage)

- workHours
- res

Storage Class Modifiers



static, extern, const

- Commonly used, details next

volatile

- Signals variable may be modified outside the current program. Useful when interfacing with hardware. Before C++11, it was also used in the context of shared memory.

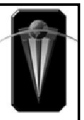
register

- Deprecated in C++11**. Hints** to compiler that the declared variable should be placed in a CPU register – wa rarely used.

auto

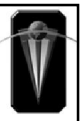
- Before C++11 there used to be `auto` but it was rarely (ever?) used – it is the default. In the C++ auto is used to signal that the compiler should figure out the type of variable from its initialization

Scope, Storage class and Linkage



- Three different concepts
- Definitions and keywords are intertwined
  - *Scope: Accessibility of a name*
  - *Storage class: Existence of the variable*
  - *Linkage: Known in current only or also in other translation units*

Storage Class Modifiers

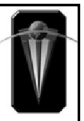


static

- Static declares a variable/function to have static duration. It is allocated at program (thread) initialization and remains accessible until the program (thread) exits.
- A static variable inside a function is initialized once and remains unchanged between function calls.
- Static data members of classes exist once per class and must be initialized within the same file scope.
- `static` and `extern` are related but different

```
void myFunction()
{
static int cnt = 0;
cnt++;
std::cout << "Call no.: " << cnt << std::endl;
}
```


Storage Class Modifiers

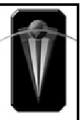


extern

- `extern` declares that a global variable/function of static duration exists.
- A extern variable may be initialized in the same file or in another file within the project.
- Declarations in a different language need to be included with, e.g., `extern "C" { … }`

```
int cnt = 0;
void myFunction() {
extern int cnt;
cnt++;
std::cout << "Call no.: " << cnt << std::endl;
}
```

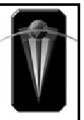
Differences between Extern and Static Linkage



- static names have internal linkage (Exception: static members of a class). Name is not visible outside the current translation unit.
- extern names have external linkage

Use	Static	Extern
Function declarations within a block	No	Yes
Names in a block	Yes	Yes
Names outside any block	Yes	Yes (default)
Functions	Yes	Yes (default)
Methods of a class	Yes	No
Attributes of a class	Yes	No

Storage Class Modifiers

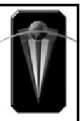


const

- `const` declares that a variable, object is constant and will not change
- Constant variables are especially important with pointers
- Use `const` as much as possible

```
const int arrLength = 1000;
int myArray[arrLength];  // Allowed in C++!

int myFunction( const int myNum ) {
res = 5 * myNum;
myNum = 3; // Illegal!
}
```


Explicit Type Conversion



Implicit type conversion

- Applied by the compiler to built-in and class types

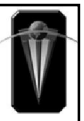
Occurs

- 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



Old-style casts

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

- Similar syntax than Java
- Avoid: Use named cast operators instead!

Named Casts

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

- `static_cast`
  - *Used to signal intentional conversion*
  - *Avoid compiler warning for loss of precision*
- Other named casts:

`reinterpret_cast const_cast dynamic_cast`

u Ottawa

Next Lecture

Complex Data Types

- Enumerations
- Structures and unions
- Arrays
- Strings
- Pointers

Jochen Lang