## Unit Overview
### Advanced C++ & Software Engineering Concepts

<table>
<thead>
<tr>
<th>Major Topics: Advanced C++</th>
<th>Major Topics: S.E. Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Expressions</td>
<td>✓ Invariants</td>
</tr>
<tr>
<td>✓ Parameter passing I</td>
<td>✓ Testing</td>
</tr>
<tr>
<td>Operator overloading</td>
<td>✓ Abstraction</td>
</tr>
<tr>
<td>Parameter passing II</td>
<td></td>
</tr>
<tr>
<td>Invisible functions I</td>
<td></td>
</tr>
<tr>
<td>Integer types</td>
<td></td>
</tr>
<tr>
<td>Managing resources in a class</td>
<td></td>
</tr>
<tr>
<td>Containers &amp; iterators</td>
<td></td>
</tr>
<tr>
<td>Invisible functions II</td>
<td></td>
</tr>
<tr>
<td>Error handling</td>
<td></td>
</tr>
<tr>
<td>Using exceptions</td>
<td></td>
</tr>
<tr>
<td>A little about Linked Lists</td>
<td></td>
</tr>
</tbody>
</table>
Review
An **expression** is something that has a value. Determining that value is **evaluation**.

Every expression has a **type**.

```plaintext
int abc;          // int is a type.
vector<int> vv;   // vector<int> is a type

abc               // Expressions of type int
34
abc * 34 + vv[2]

42.7              // Expression of type double
cout << "Hello"   // Expression of type std::ostream
vv                 // Expression of type std::vector<int>
```
A C++ expression is either an Lvalue or an Rvalue—never both.

An **Lvalue** has a value that persists beyond the current expression. Variables, things pointed to by pointers, and parts of Lvalues are all Lvalues.

```cpp
abc    // Lvalue
*p    // Lvalue
vv[3]  // Lvalue
x.qq   // Lvalue
```

Historically, the “L” in “Lvalue” stood for “left”, as in the left side of an assignment operator. But in C++, a const variable is still an Lvalue.

So a C++ Lvalue is, roughly, an expression that can go on the left side of an assignment operator, or **could**, if it were not const.

We can take the address of an Lvalue. If it is non-const, then we can pass it by reference.
An **Rvalue** is an expression that is not an Lvalue.

42.7    // Rvalue
abc + 34    // Rvalue

```c
int add(a, b)
{ return a+b; }
```

```c
add(6, 8)    // Rvalue
```

Important. An Rvalue is something that is *about to go away*. That means that we can “mess it up” without causing problems.
Review
Expressions — TRY IT (Exercises)

Consider the following C++ code.

```cpp
nn = rst;
cc = nn + 4;
for (int i = 0; i < 5; ++i)
    cout << cc + i << "\n";
```

Classify each of the following expressions as *Lvalue* or *Rvalue*. *Answers are on the next slide.*

1. `nn`  
2. `rst`  
3. `4`  
4. `nn + 4`  
5. `cc + i`  
6. `i < 5`  
7. `cout`  
8. `"\n"`
Consider the following C++ code.

```cpp
nn = rst;
cc = nn + 4;
for (int i = 0; i < 5; ++i)
    cout << cc + i << "\n";
```

Classify each of the following expressions as *Lvalue* or *Rvalue*.

**Answers:**

1.  `nn`  *Lvalue*  
2.  `rst`  *Lvalue*  
3.  `4`  *Rvalue*  
4.  `nn + 4`  *Rvalue*  
5.  `cc + i`  *Rvalue*  
6.  `i < 5`  *Rvalue*  
7.  `cout`  *Lvalue*  
8.  `"\n"`  *Rvalue*
C++ provides three primary ways to pass a parameter or return a value.

**By value:**

```cpp
void p1(Foo x);         // Pass x by value
Foo r1();               // Return by value
```

**By reference:**

```cpp
void p2(Foo & x);       // Pass x by reference
Foo & r2();             // Return by reference
```

**By reference-to-const** (some people say “const reference”):

```cpp
void p3(const Foo & x); // Pass x by reference-to-const
const Foo & r3();       // Return by reference-to-const
```
For many purposes, when we pass objects, reference-to-const combines the best features of the first two methods.
For most parameter passing, we pass either by value or by reference-to-const.
- By value: simple types (int, char, etc.), pointers, iterators.
- By reference-to-const: larger objects, or things we are not sure of.

We normally return by value.

And then there are special cases ...

We pass by reference, if we want to send the value of the parameter back to the caller.

We *might* return by reference or by reference-to-const, if we are returning a value that is not going away.
- The former if the caller gets to modify the value; the latter if not.
Operator Overloading
C++ allows **overloading** of most operators.

- Define standard operators for new types.
- No new operators, and no changes in **precedence**, **associativity**, or **arity**.
- Function name is `operator` plus the symbol, e.g., `operator-`.

Subtraction for a class `Num` as a **global** function:

```cpp
Num operator-(const Num & a, const Num & b);
```

Or as a **member** function; the first operand is the object (`*this`):

```cpp
class Num {
public:
    Num operator-(const Num & b) const;
};
```
Operators with the same symbol are distinguished by parameters.

\begin{verbatim}
Num operator-(const Num & a, const Num & b); // a-b  
Num operator-(const Num & a);                // -a
\end{verbatim}

Some cannot be distinguished by the parameters we would expect: in particular, ++a and a++. The latter gets a dummy \texttt{int}; it is always zero and may be ignored.

\begin{verbatim}
class Num {  
public:  
    Num & operator++();       // ++a  
    Num operator++(int dummy); // a++
}\end{verbatim}

Why are different return methods used here?
To input or print our objects we use C++ standard-library streams.

- We will look at stream **insertion** (**operator<<**).
- Stream **extraction** (**operator>>**) is similar.

The stream insertion operator:

- Takes an output stream (**std::ostream**) and some object.
- Returns the output stream.

As we have observed, this makes the following work:

```cpp
cout << a << b;  // Same as (cout << a) << b;
```
Stream insertion:

- **Must** be global.
  - Otherwise, it is a member of `std::ostream`, which we cannot write.
- Gets its stream by reference.
  - Because it modifies the stream (by outputting to it).
- Gets its object to be printed by reference-to-const.
- Returns its stream by reference.
  - The stream is not going away. Also, we do not copy streams.

```cpp
std::ostream & operator<<(std::ostream & theStream,
                          const MyClass & theObject)
{
    theStream << theObject.x << ", " << theObject.y;
    return theStream;
}
```

This is an example. In practice, write whatever is appropriate for the type your code deals with.
Global function:

```
Num operator-(const Num & a, const Num & b);
```

Member function:

```
class Num {
public:
    Num operator-(const Num & b) const;
};
```

Suppose there is an implicit type conversion from double to Num.

- If we write `Num - Num` as a global, then we get, for free,
  - `Num - double`
  - `double - Num`
- But if it is a member, then we only get the first one. 😞
Use global functions for overloaded arithmetic, comparison, and bitwise operators that do not modify their first operand.

- +  -  binary *  /  %  ==  !=  <  >  <=  >=  &  |  ^

Use global functions for overloaded operators whose first operand is a type you cannot add members to.

- Common examples: stream insertion <<, stream extraction >>.

Use member functions for other overloaded operators.

- =  [ ]  unary *  +=  -=  *=  /=  ++  --  etc.
Parameter Passing II
There are actually four parameter-passing methods in C++:
by value, by reference, by reference-to-const, and ....

By Rvalue reference—introduced in the 2011 C++ Standard:

```cpp
void p4(Foo && x);  // Pass x by Rvalue reference
Foo && r4();        // Return by Rvalue reference
```

Technically, there is a fifth method: by Rvalue reference-to-const:

```cpp
void p5(const Foo && x);
const Foo && r5();
```

However, this method is, AFAIK, useless.
Parameter Passing II
By Rvalue Reference [1/3]

void p4(Foo && x);
Foo && r4();

Passing by Rvalue reference is much like by reference-to-const:
- No copy is made.
- Proper calling of virtual functions works.
- Implicit type conversions are allowed.

The differences:
- Only non-const Rvalues may be passed by Rvalue reference.
- Rvalues *prefer* to be passed by Rvalue reference.
  - This matters when we overload a function. Given a choice of which version to use, preference is given to passing by Rvalue reference, when an Rvalue is passed.
- Modification of the passed value is allowed.

So, what is the point?
Again:
- Only non-const Rvalues may be passed by Rvalue reference.
- Rvalues *prefer* to be passed by Rvalue reference.
- Modification of the passed value is allowed.

Suppose we have the following function:

```cpp
void g(const Foo & p);  // Takes any Foo value
```

But now we write another version:

```cpp
void g(const Foo & p);  // Gets Lvalues, cannot modify
void g(Foo && p);       // Gets Rvalues, CAN modify
```
void g(const Foo & p);  // Gets Lvalues, cannot modify
void g(Foo && p);       // Gets Rvalues, CAN modify

How does this help?

- We have a way of figuring out whether an argument is an Rvalue—
  that is, whether it is about to go away.
- In that case, we may be able to speed up processing by doing
  things that mess up the value of an Rvalue argument.

We do not pass by Rvalue reference very often.

Passing by Rvalue reference is mostly done in low-level code—in
particular, when we construct new objects from existing objects
of the same type. Given an Lvalue, we run a slow copy
constructor. Given an Rvalue, we can run a fast move
constructor.

More on this soon.
Parameter Passing II

Summary

<table>
<thead>
<tr>
<th></th>
<th>By Value</th>
<th>By Reference</th>
<th>By Reference-to-Const</th>
<th>By Rvalue Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makes a copy</td>
<td>YES 😞</td>
<td>NO 😞</td>
<td>NO 😞</td>
<td>NO</td>
</tr>
<tr>
<td>Allows for polymorphism</td>
<td>NO 😞</td>
<td>YES 😞</td>
<td>YES 😞</td>
<td>YES</td>
</tr>
<tr>
<td>Allows implicit type conversions</td>
<td>YES 😞</td>
<td>NO 😞</td>
<td>YES 😞</td>
<td>YES</td>
</tr>
<tr>
<td>Allows passing of:</td>
<td>Any copyable value 😊</td>
<td>Non-const Lvalues 😞?</td>
<td>Any value* 😊</td>
<td>Non-const Rvalues*</td>
</tr>
</tbody>
</table>

*By reference-to-const and by Rvalue reference look similar. But Rvalues prefer to be passed by Rvalue reference.

void g(const Foo & x);  // Can take any Foo value

However:

void g(const Foo & x);  // Gets Lvalues, which it cannot modify
void g(Foo && x);       // Gets Rvalues, which it CAN modify
Invisible Functions I
Here is a simple class `Dog`:

```cpp
// class Dog
// What member functions does this class have?
// Invariants: None.
class Dog {

    // ***** Dog: Data members *****
    private:
        int _a;
        Cat _b;
}; // End class Dog
```

What member functions does class `Dog` have? *See the next slide ...*
Class `Dog` has several invisible member functions. These are automatically written by the compiler. Prototypes for 6 of these are shown below.

- **Ctor** means constructor. **Dctor** means destructor.

```cpp
class Dog {
public:
    Dog();                            // 1. Default ctor
    ~Dog();                           // 2. Dctor
    Dog(const Dog & other);           // 3. Copy ctor
    Dog & operator=(const Dog & rhs); // 4. Copy assignment
    Dog(Dog && other);                // 5. Move ctor*
    Dog & operator=(Dog && rhs);      // 6. Move assignment*
}
```

*Move ctors and move assignment operators were added in C++11.*

These are the **Big Five**.
TO DO

- Look at some code that does possibly unexpected things using invisible functions.

See invisible.cpp.

The Point

- “Invisible functions” are real functions that are really called and really execute. If we let the compiler write them for us, then we never see them, but they are there.
- These functions provide useful functionality. Their existence is a Good Thing. But if we write them ourselves, then we need to exercise care, lest we produce code that behaves oddly.

Next we look at each of these six functions: what it is and when it is called. Then we discuss what the compiler may write for us, when it will do so, and when we should write the functions ourselves.
Invisible Functions I will be continued next time.