Unit Overview
Advanced C++ & Software Engineering Concepts

Major Topics: Advanced C++
✓ The structure of a package
✓ Parameter passing
✓ Operator overloading
✓ Silently written & called functions
  ▪ Pointers & dynamic allocation
  ▪ Managing resources in a class
  ▪ Templates
  ▪ Containers & iterators
  ▪ Error handling
  ▪ Introduction to exceptions
  ▪ Introduction to Linked Lists

Major Topics: S.E. Concepts
✓ Abstraction
✓ Invariants
✓ Testing
✓ Some principles
Review
The Structure of a Package

A **client** of a module is *code* that uses it.

**Type conversion**: take value and return value of another type.

- **Implicit**: double d = 4.5 + 3;
- **Explicit**: double d = 4.5 + double(3);
- No conversion: double d = 4.5 + 3.0;

3 has type int.

To add 3 to 4.5 (which has type double), we must use a type conversion to get a double.

3.0, on the other hand, has type double.
Abstraction: Separate the purpose of a module from its implementation.

- **Functional abstraction**
- **Data abstraction**

Key term: **Abstract Data Type**

- An *abstract data type* (ADT) is a collection of data and a set of operations on the data.
- The implementation is not specified.
- ADTs will be a major topic of this course.
Function `printIntArray` is given an array of integers called “arr” and a size_t called “size”. It executes a for loop in which local size_t variable `i` is initialized to 0, the loop continues as long as “i < size” evaluates to true, and `i` is pre-incremented after each loop iteration. Inside the loop, a reference to an item in array arr is retrieved using the bracket operator, with parameter `i`, and then inserted in cout (using overloaded operator<<), followed by an array of chars containing a blank and a null. After the loop, stream manipulator endl is inserted in cout. The function then terminates.

Function `printIntArray` prints an array of ints to cout, given the array and its size. Items are separated by blanks, and followed by a blank and a newline.

Describe this function, in detail.
### Review

Parameter Passing [1/2]

<table>
<thead>
<tr>
<th></th>
<th>By value</th>
<th>By reference</th>
<th>By reference-to-const</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makes a copy</td>
<td>YES ☺*</td>
<td>NO ☹</td>
<td>NO ☹</td>
</tr>
<tr>
<td>Allows for polymorphism</td>
<td>NO ☹*</td>
<td>YES ☺</td>
<td>YES ☺</td>
</tr>
<tr>
<td>Allows passing of const values</td>
<td>YES ☺</td>
<td>NO ☹**</td>
<td>YES ☺</td>
</tr>
<tr>
<td>Allows implicit type conversions</td>
<td>YES ☺</td>
<td>NO ☹</td>
<td>YES ☺</td>
</tr>
</tbody>
</table>

*These are problems when we pass **objects**.  
**Maybe this is bad. When we want to send changes back to the client (which is a big reason for passing by reference), disallowing const values is a good thing.

So, for most purposes, *when we pass objects*, reference-to-const combines the best features of the other two methods.
Review
Parameter Passing [2/2]

We **pass parameters** by reference when we want to modify the client’s copy.

```c
void addThree(int & theInt)
{ theInt += 3; }
```

Otherwise, we generally pass:
- simple types by value.
- objects by reference-to-const.

```c
void func(double d, const MyClass & q);
```

We usually **return** by value, unless we return an object not local to this function.
- Return by reference if we return a pre-existing object for the client to modify.
- Return by reference-to-const if we return a pre-existing object that the client should not modify (in particular, if the object is const).

```c
int & arrayLookUp(int theArray[], int index);
const int & arrayLookUp(const int theArray[], int index);
```
Review
Operator Overloading

Operators can be implemented using global or member functions.

- Global: the parameters are the operands.
- Member: first operand is \*this, the rest are parameters.
- Postfix increment & decrement (n++, n--) get a dummy int parameter, to distinguish them from the prefix versions (++n, −−n).

Implement an operator using a member function, unless you have a good reason not to.

- Good Reason #1: To allow for implicit type conversions on the first argument. Applies to: non-modifying arithmetic, comparison, and bitwise operators.
  - For example: + − * / % == != < <= > >=

- Good Reason #2: When you cannot make it a member, because it would have to be a member of a class you cannot modify.
  - Quintessential examples: stream insertion (<<) and extraction (>>).

We usually use operators only for operations that happen quickly.

- One exception: Assignment for container types.
An **invariant** is a condition that is always true at a particular point in an algorithm.

**Example**

- Suppose that `myArray` is an array of `int`'s with size `myArraySize`.
- We wish to set the variable `myItem` equal to `myArray[i]`, if possible.

```java
if (i < 0) {
    errorMessage("Error: i is too small");
    return;
}
// Invariant: i >= 0
if (i >= myArraySize) {
    errorMessage("Error: i is too large");
    return;
}
// Invariant: (i >= 0) && (i < myArraySize)
myItem = myArray[i];
```
We use invariants:

- To ensure that we are allowed to perform various operations.
- To remind ourselves of the information that is implicitly known in a program.
- To document ways in which code can be used.
- To help us verify that our programs are correct.
We are particularly interested in two special kinds of invariants: **preconditions** and **postconditions**.

A **precondition** is an invariant at the beginning of a function.

- The responsibility for making sure the precondition is true rests with the calling code.
- In practice, a precondition states **what must be true for the function to execute properly**.

A **postcondition** is an invariant at the end of a function.

- It tells what services the function has performed for the client code.
- The responsibility for making sure the postcondition is true rests with the function itself.
- In practice, postconditions **describe the function’s effect using statements about objects & values**.
Preconditions and postconditions are the basis of operation contracts.

- We think of a function call as the carrying out of a contract. The function says to the caller, “If you do this [preconditions], then I will do this [postconditions].”
- If the preconditions are met, then the function is required to make the postconditions true upon its (normal) termination.
  - We consider abnormal termination (exceptions) later.
- If the preconditions are not met, then the function can be considered to have no responsibilities.

Punch Line

- In this class, we write preconditions and postconditions for every function you write (except, possibly, main).
  - See the “Coding Standards”.

Software Engineering Concepts: Invariants
Pre & Post [2/3]
Example

- Write reasonable pre- and postconditions for the following function, which is supposed to store the number 7 in the provided memory.

```c
// store7
// Pre: ptr points to a block of memory large enough to hold an int.
// Post: *ptr == 7.
void store7(int * ptr) {
    *ptr = 7;
}
```

Preconditions: What **must be true** for the function to execute properly?

Postconditions: **Describe the function’s effect** using statements about objects & values.
Another important kind of invariant is a class invariant.

- A class invariant is an invariant that holds whenever an object of the class exists, and execution is not in the middle of a public member function call.

Class invariants are true here

- Class invariants are preconditions of every public member function, except constructors.
- Class invariants are postconditions for every public member function, except the destructor.
- Since we know this, you do not need to list class invariants in the pre- and postcondition lists of public member functions.
- In practice, class invariants are statements about data members that indicate what it means for an object to be valid or usable.
Write reasonable class invariants for the following class.

```cpp
// class Date
// Invariants:
//     1 <= mo_ <= 12
//     1 <= day_ <= k, where k is no. of days in month mo_

class Date {
    public:
        [Lots of code goes here]
    private:
        int mo_;    // Month 1..12
        int day_;   // Day 1..#days in month given by mo_
};  // End class Date
```

Class invariants: **Statements about data members** that indicate what it means for an object to be **valid** or **usable**.
Think about dynamic allocation. In “C”, we do:

```c
Foo * p = (Foo *)malloc(100 * sizeof(Foo));
```

In C++, we prefer:

```cpp
Foo * p = new Foo[100];
```

Why?
- Yes, it’s simpler and cleaner. What other reasons are there?
- Hint: The two lines of code above do not do the same thing.
- Another hint: We’re discussing invariants.
- See the next slide.
In C++, using "new" calls a constructor, thus ensuring that class invariants are true.

With Constructor & Destructor

```
Constructor -> Member Func. -> Member Func. -> Destructor
```

Class invariants are true here

Without Constructor & Destructor

```
Member Func. -> Member Func.
```

Are class invariants true here?

```
Member Func. -> Member Func.
```

Maybe not ...

The job of a constructor is to make the class invariants true.
C++ will **silently write** four important member functions:

- Default ctor.
- Copy ctor.
- Copy assignment. “The Big Three”
- Dctor.

When

- The default ctor is silently written when you declare no ctors.
- The other three are silently written when you do not declare them.

The silently written versions:

- Are **public**.
- Call the **corresponding functions** for all data members.
Some of these can be *silently called* as well.

- The default ctor is called when you declare a variable with no ctor parameters, and when you declare an array (or, generally, any container holding already initialized objects).
- The copy ctor is called when you pass by value and *maybe* when you return by value.
- The dctor is called:
  - On an automatic (local, non-static) object when it goes out of scope.
  - On a static object when the program ends.
  - On a non-static member object when the object it is a member of is destroyed.
  - On a dynamic object when you delete a pointer to it.
Silently written functions are **good**.
- Do not waste effort. If the compiler will write a perfectly good function for you, then do not write it yourself.

So, use them often. And when you do, indicate this in a comment.
- This is a reminder that these functions exist and are part of the class design.

```cpp
class Aardvark {
public:
    // Default ctor
    // Pre: None.
    // Post: None.
    Aardvark();

    // Compiler-generated copy ctor, copy assn, dctor are used.
```
Silently Written & Called Functions
When to Write Them?

When should you write these functions yourself?
- When you need them, but they are not written for you.
- When the silently written ones do not do what you want.

```cpp
class Llama {
    ...

private:
    int * p;
```

Should the copy ctor just copy `p` (shallow copy) or should it also copy the memory that `p` points to (deep copy)?
- The answer depends on what `p` is for.
- The silently written copy ctor does a shallow copy.

**The Law of the Big Three**
- **If you need to declare one of the Big Three** (copy ctor, copy assignment, dctor), **then you probably need to declare all of them.**
- This tends to happen when the class manages a resource (for example, dynamically allocated memory, an open file, etc.). More on this soon.
Silently Written & Called Functions
Eliminating Them [1/2]

We have covered:
- What the compiler writes for you.
- How & when to replace these with your own versions.

But sometimes we want to **eliminate** these functions.

Why would we want this?
- Most common reason: making objects uncopyable.
  - This allows us to put strong controls on the creation and destruction of such objects.
  - It also disallows passing by value.

So, how do we eliminate the copy ctor and copy assignment?
- If we do not define them, then the compiler will, right?
- If we do define them, then they exist, right?

Thus: declare them, but do not define them.
- But what if someone else defines them ...
Silently Written & Called Functions
Eliminating Them [2/2]

How do we eliminate the copy ctor and copy assignment?

- **Declare** the copy ctor and copy assignment **private**.
- Do not **define** them.

```cpp
class Mule {
private:
    // Uncopyable class.
    // Private copy ctor, copy assn. Do not define these.
    Mule(const Mule &);
    Mule & operator=(const Mule &);

Now **no one** can call these functions.

- You (the class author) cannot accidentally call them, because you did not define them.
- Client code *can* define them, but that does not matter; they cannot call them, because they are private.
Simple Class Example
Write It!

TO DO

- Write a simple class that stores and handles a time of day, in seconds.
  - Call it “TimeSec”.
  - Give it reasonable ctors, etc.
    - Can we use silently written functions?
      - Yes, for the Big Three.
      - We will write our own default ctor.
  - Give it reasonable operators.
    - Like what?
      - Pre & post ++, --.
      - Stream insertion: <<.
    - Note: It would be reasonable to add more. We will not, but only due to time constraints.

Partially done. See timesec.h, timesec.cpp, on the web page.
Simple Class Example
TO BE CONTINUED ...

*Simple Class Example* will be continued next time.