

# Course Overview

## Expressions

## Parameter Passing I

---

CS 311 Data Structures and Algorithms  
Lecture Slides  
Monday, August 24, 2020

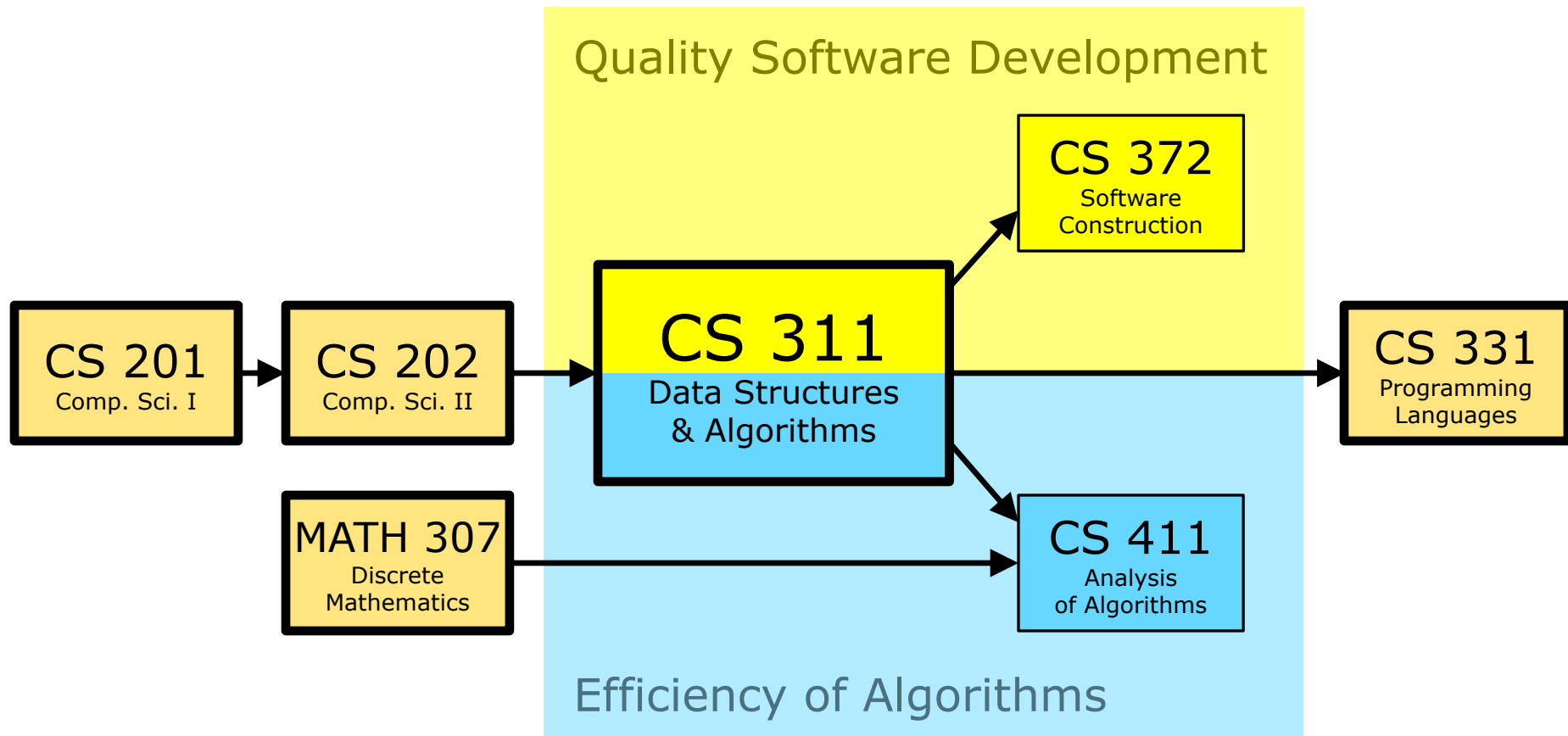
Glenn G. Chappell  
Department of Computer Science  
University of Alaska Fairbanks  
[ggchappell@alaska.edu](mailto:ggchappell@alaska.edu)  
© 2005–2020 Glenn G. Chappell  
Some material contributed by Chris Hartman

---

# Course Overview

# Course Overview

## CS 311 in the Comp. Sci. & Comp. Eng. Programs



BS in Computer Science. All are required.

BS in Computer Engineering. Bold-border: required.

CS 411: Approved elective. CS 372: Ask your advisor.

# Course Overview

## Goals

---

Upon successful completion of CS 311, you should:

- Have experience writing and documenting high-quality code.
- Understand proper error handling, enabling software components to support robust, reliable applications.
- Be able to perform basic analyses of algorithmic efficiency, including use of big- $O$  and related notation.
- Be familiar with various standard algorithms, including those for searching and sorting.
- Understand what data abstraction is, and how it relates to software design.
- Be familiar with standard container data structures, including implementations and relevant trade-offs.

# Course Overview

## Programming Language

---

We will achieve our goals, in part, by doing an in-depth study of a particular programming language, along with its standard libraries: ANSI C++ (2017 standard) and the Standard Template Library.

You will need to have access to an up-to-date C++ compiler.

Any version of a major compiler released within the last year should be fine.

Visual Studio 2017  
is **not** acceptable.

You may use the CS labs (Duckering, 5th floor), which have appropriate C++ compilers available.

# Course Overview

## Topics

---

The following topics will be covered, roughly in order:

- Advanced C++
- Software Engineering Concepts
- Recursion
- Searching
- Algorithmic Efficiency
- Sorting
- Data Abstraction
- Basic Abstract Data Types & Data Structures:
  - Smart Arrays & Strings
  - Linked Lists
  - Stacks & Queues
  - Trees (various kinds)
  - Priority Queues
  - Tables
- Briefly: external data, graph algorithms.

**Goal: Practical generic containers**

A **container** is a data structure holding multiple items, usually all the same type.

A **generic** container is one that can hold objects of client-specified type.

# Course Overview

## Projects

---

Your primary task this semester is to complete eight high-quality, tested, documented software projects. Descriptions of these will be posted on the class webpage.

1. High-Quality Class
2. Moderately Smart Array
3. Potpourri
  - Various functions involving exceptions, Linked Lists, recursion
4. Recursive Backtracking
5. Frightfully Smart Array
6. Linked Lists
7. Trees
8. Using Tables


## Course Overview

### Terminology & Notation

---

We will be covering a lot of terminology and notation.

**Terminology** is the words we use when discussing some technical topic. When I introduce new terminology, it is in boldface.



Some terminology (which you should already know): when we **add** the **numbers three** and **five**, we obtain the number **eight**.

**Notation** is the symbols we use in technical discussions.

Some notation (which you should already know):  $3 + 5 = 8$ .

It is very important to know the terminology and notation we will be using. Without it, we cannot even begin to talk about the class material. So *watch out for it!*



# Unit Overview


## Advanced C++ & Software Engineering Concepts

Our first unit: Advanced C++ and Software Engineering Concepts.

- Some of this will be review from CS 201 & 202.

### Major Topics

- Advanced C++
  - Expressions
  - Parameter passing I
  - Operator overloading
  - Parameter passing II
  - Invisible functions I
  - Integer types
  - Managing resources in a class
  - Containers & iterators
  - Invisible functions II
  - Error handling
  - Using exceptions
  - A little about Linked Lists
- Software Engineering Concepts
  - Invariants
  - Testing
  - Abstraction



These two will be covered concurrently.

Later in the semester we will cover other advanced C++ topics.

---

# Expressions

# Expressions

## What is an Expression?

---

An **expression** is something that has a value.

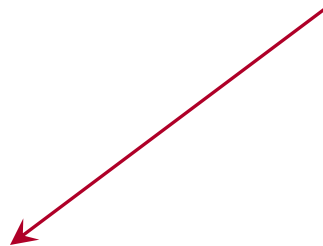
**Evaluating** an expression means determining its value.

Examples of C++ expressions:

- `abc`
- `42.7`
- `(n+3)*14-q+vv[6]`
- `foo(x)`
- `cout << "Hello!"`

Q. What is the value of this expression?

A. ...



# Expressions

## What is an Expression?

An **expression** is something that has a value.

**Evaluating** an expression means determining its value.

Examples of C++ expressions:

- `abc`
- `42.7`
- `(n+3)*14-q+vv[6]`
- `foo(x)`
- `cout << "Hello!"`

Q. What is the value of this expression?

A. The value is `cout`.

So we can do this:

```
(cout << "Hello!") << x;
```

... which is the same as this:

```
cout << "Hello!" << x;
```

The following are *not* C++ expressions:

- `int abc;`
- `return abc;`
- `for (int i = 0; i < 10; ++i) cout << i << "\n";`
- `using std::cout;`

## Expressions Types [1/2]

---

We classify expressions according to the kind of value each represents. An expression's classification is its **type**.

```
int abc;
```

int is a type. abc is a variable of type int.

```
34          // Expression of type int
abc + 34     // Expression of type int
42.7        // Expression of type double
cout << x    // Expression of type std::ostream
vector<int> vv;
vv          // Expression of type std::vector<int>
vv[2]       // Expression of type int
```

## Expressions Types [2/2]

---

When we need a value whose type is different from that of a given expression, a **type conversion** may be done.

```
double dd = 34;    // 34 has type int;  
                  // this will be converted to double
```

Type conversions can be **explicit** (stated in the code) or **implicit**.  
The above type conversion is implicit. That below is explicit.

```
double dd2 = static_cast<double>(abc);
```

A type conversion creates a new value; it does not modify the original. For example, above, `abc` is unchanged.

## Expressions

### Lvalues & Rvalues [1/4]

---

Every C++ expression is either an Lvalue or an Rvalue.

An **Lvalue** (say “ELL value”) has a value that persists beyond the current expression. For example, every variable is an Lvalue.

```
int abc;           // abc is an Lvalue
const double dd;   // dd is an Lvalue
```

If something is an Lvalue, then parts of it are also Lvalues. And something pointed to by a pointer is an Lvalue.

```
vv[3]             // vv is an Lvalue, and so is vv[3]
x.qq              // x is an Lvalue, and so is x.qq
*p               // *p is an Lvalue
```

## Expressions

### Lvalues & Rvalues [2/4]

---

An Lvalue has a location in memory. We can take its address.

```
int * p = &abc;    // Legal because abc is an Lvalue
```

We can also pass an Lvalue by reference—if it is non-const.

```
void incr(int & n)
{ ++n; }
```

```
incr(abc);    // Legal because abc is a non-const Lvalue
```

Historically, “Lvalue” comes from the idea that we can put it on the left-hand side of an assignment operator (=). “L” stood for “left”. But note that, in C++, a const variable is still an Lvalue.



## Expressions

### Lvalues & Rvalues [3/4]

---

An **Rvalue** (say “ARR value”) is an expression that is not an Lvalue.

```
42.7          // 42.7 is an Rvalue
```

```
abc + 34      // abc + 34 is an Rvalue
```

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

```
add(6, 8)     // add(6, 8) is an Rvalue
```

Why do we care? An Rvalue is something that is *about to go away*.  
That means that we can “mess it up” without causing problems.

## Expressions

### Lvalues & Rvalues [4/4]

---

We cannot take the address of an Rvalue.

```
int * p2 = &(abc+34); // DOES NOT COMPILE!
```

We cannot pass an Rvalue by reference.

```
incr(6); // DOES NOT COMPILE!
```

A C++ expression is either an Lvalue or an Rvalue, but *never both!*

---

# Parameter Passing I

# Parameter Passing I

## Overview

---

C++ provides three primary ways to pass a parameter or return a value.

### **By value:**

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

### **By reference:**

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

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

```
void p3(const Foo & x);    // Pass x by reference-to-const
const Foo & r3();          // Return by reference-to-const
```

## Parameter Passing I

### Details — By Value [1/2]

---

```
void p1(Foo x);  
Foo r1();
```

Passing by value means that a **copy** is made.

- Below,  $x$  (in  $p1$ ) is a copy of  $y$ . Modifying  $x$  does nothing to  $y$ .

```
Foo y;  
p1(y);
```

The copy is made with an **implicit** function call to the Foo **copy constructor** or **move constructor**.

- This may be slow, if  $y$  is a large object.
- And if Foo has no copy/move constructor, then it is impossible.

## Parameter Passing I

### Details — By Value [2/2]

---

Passing by value does *not* allow for proper handling of derived classes, including calling of virtual functions.

```
class Base { ... };  
class Derived : public Base { ... };  
  
void ff(Base bb);  
  
Derived dd;  
ff(dd); // This might cause problems
```

## Parameter Passing I

### Details — By Reference [1/2]

---

```
void p2(Foo & x);  
Foo & r2();
```

When passing by reference, no copy is made.

- The original and passed versions are the *same object*.

```
Foo y;  
p2(y); // Modifying x inside p2 will modify y
```

Be careful when returning by reference.

- Do not return a value that goes away when the function ends.

```
int & squareThis(int n)  
{ int square = n * n; return square; }
```

BAD! ☹️

## Parameter Passing I


### Details — By Reference [2/2]

---

Passing by reference *does* allow for proper handling of derived classes, including calling of virtual functions.

```
class Base { ... };  
class Derived : public Base { ... };
```

```
void ff(Base & bb);
```



Note

```
Derived dd;  
ff(dd); // No problem
```

Only non-const Lvalues can be passed by reference.



## Parameter Passing I

### Details — By Reference-to-Const [1/2]

---

```
void p3(const Foo & x);  
const Foo & r3();
```

When passing by reference-to-const, no copy is made.

- Instead, the original and the passed version are the same object ...
- ... *unless* they are of different types; implicit type conversions may be done.

```
void h(const double & z);  
const double dd;  
const int ii;  
h(dd); // z is dd  
h(ii); // Legal, but x is not ii
```

As before, be careful when returning by reference-to-const.

## Parameter Passing I

### Details — By Reference-to-Const [2/2]

---

Like passing by reference, passing by reference-to-const allows for proper handling of derived classes, including calling of virtual functions.

Const variables may be passed by reference-to-const. The passed version is not modifiable.

In fact, *any value at all* may be passed by reference-to-const.

# Parameter Passing I

## Details — Summary of the Three

	By Value	By Reference	By Reference-to-Const
Makes a copy	YES ☹️*	NO 😊	NO 😊
Allows for polymorphism	NO ☹️**	YES 😊	YES 😊
Allows implicit type conversions	YES 😊	NO ☹️	YES 😊
Allows passing of:	Any copyable value 😊	Non-const Lvalues ☹️?***	Any value 😊

\*This is a problem when we pass values that take time to copy, like large objects.

\*\*This is a problem when we use inheritance, as we often do with 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 many purposes, *when we pass objects*, reference-to-const combines the best features of the first two methods.

## Parameter Passing I

### Usage — Normal

---

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.

```
template<typename T>  
void f1(const T & x);    // x might be a large object (?)
```

We normally return by value.

```
Foo f2();
```

There are special cases where we may use other methods ...

## Parameter Passing I

### Usage — Special [1/2]

---

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

```
// Convert seconds after midnight to hrs, mins, secs.  
void secsToHMS(int secs, int & h, int & m, int & s);
```

## Parameter Passing I

### Usage — Special [2/2]

---

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.
- Idea: You are returning an Lvalue to the caller.

```
class BunchOfInts {  
public:  
    int & operator[](std::size_t index)  
    { return _theInts[index]; }  
    const int & operator[](std::size_t index) const  
    { return _theInts[index]; }  
  
private:  
    int _theInts[100];  
};
```