Software Engineering Concepts: Abstraction
Operator Overloading
Software Engineering Concepts: Assertions

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, August 28, 2024

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
Advanced C++ & Software Engineering Concepts

Topics: Advanced C++

- ✓ Expressions
- ✓ Parameter passing I
- Operator overloading
- Example class
- Parameter passing II
- Invisible functions I
- Managing resources in a class
- Containers & iterators
- Invisible functions II
- Error handling
- Using exceptions

Topics: S.E. Concepts

- Abstraction
- Assertions
- Testing
- Invariants

Review

Review
Expressions [1/3]

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

Every expression has a **type**.

```
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>
```

Review
Expressions [2/3]

A C++ expression is either an Lvalue or an Rvalue—never both.

An **Lvalue** has a value that persists. Variables, what pointers point to, and parts of Lvalues are all Lvalues.

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

We can take the address of an Lvalue. If it is non-const, then we can pass it by reference.

Review
Expressions [3/3]

An **Rvalue** is an expression that is not an Lvalue.

```
42.7        // Rvalue
abc + 34    // Rvalue

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

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

We expect that an Rvalue is *about to go away*. So we can “mess it up” without causing problems. This can lead to faster code.

Review
Expressions — TRY IT (Exercises)

Consider the following C++ code.

```
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"`

Review
Expressions — TRY IT (Answers)

Consider the following C++ code.

```
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***

Review
Parameter Passing I [1/3]

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
```

Review
Parameter Passing I [2/3]

	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 ☺

For many purposes, *when we pass objects*, reference-to-const combines the best features of the first two methods.

Review
Parameter Passing I [3/3]

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.

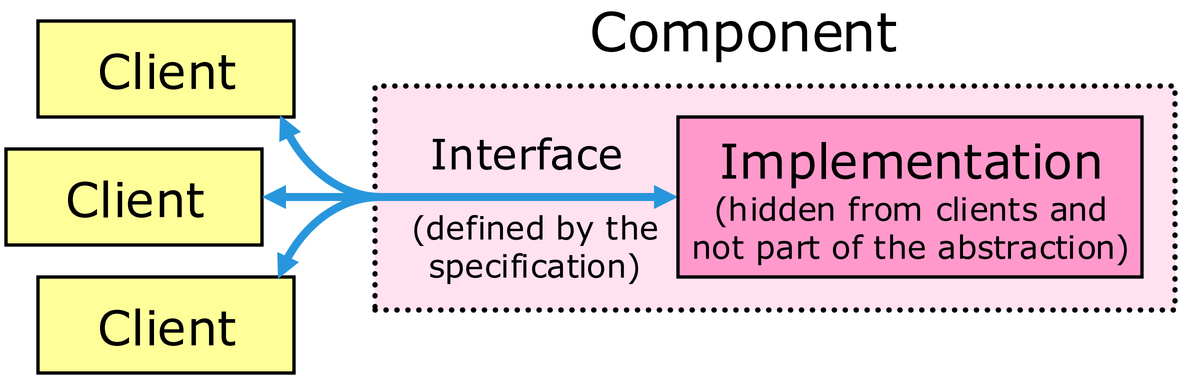
But there are special cases where we may use other methods.

- 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.

Software Engineering Concepts: Abstraction

Software Engineering Concepts: Abstraction Definitions

Abstraction: Considering a software component in terms of *how* and *why* it is used—what it looks like from the *outside*—separate from its internal implementation.



Here, "**component**" is just a general term for a *thing*: function, class, package, etc.

We use the term "**client**" for *code* that makes use of a component.

In this course, a client is code. A **user** is a person.

Software Engineering Concepts: Abstraction
Functional Abstraction [1/2]

Functional abstraction: applying the idea of abstraction to functions. So, dealing with functions in terms of how and why they are used—what they look like from the outside—separate from their internal implementation.

You have certainly used this idea—even if you were not familiar with the term “functional abstraction”. *See the next slide for an example.*

Software Engineering Concepts: Abstraction
Functional Abstraction [2/2]

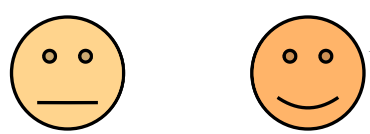
```
void printIntVec(const vector<int> & data)
{
    for (size_t i = 0; i != data.size(); ++i)
        cout << data[i] << " ";
    cout << "\n";
}
```

Functional abstraction: what it does, but not how it does it

Describe this function, in detail.

Function `printIntVec` is given a `vector` of `int`s called `data`, passed by reference-to-const. It executes a `for` loop in which local `size_t` variable `i` is initialized to `0`, the loop continues as long as "`i != data.size()`" evaluates to `true`, and `i` is pre-incremented after each loop iteration. Inside the loop, a reference to an item in `data` is retrieved using the bracket operator, with parameter `i`, and then inserted into `cout`, using overloaded `operator<<`, followed by an array of `char`s of size 2, which contains a blank and a null `char`. After the loop, stream manipulator `endl` is inserted into `cout`. The function then terminates.

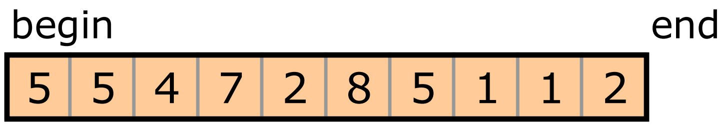
Function `printIntVec` prints a given `vector` of `int`s to `cout`. Items are separated by blanks, and followed by a blank and a newline.



Software Engineering Concepts: Abstraction
Data Abstraction

Data abstraction: applying abstraction to the structure of data. Consider the form of the data without regard to how it is stored.

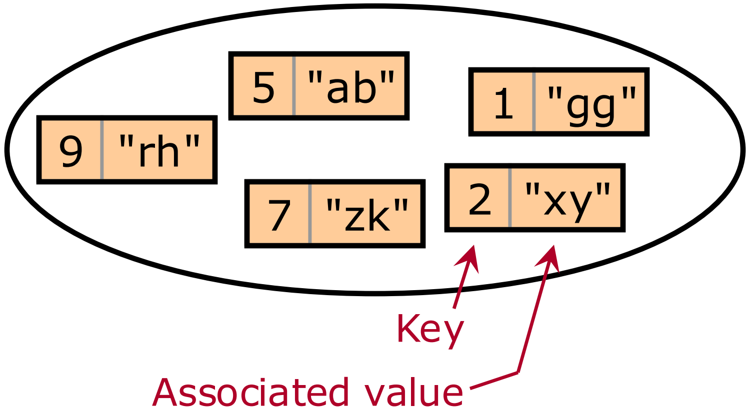
For example, a dataset may be a sequence of items, in some order.



Q. What value lies in position 3 (start at 0)?

A. 7.

Or it may be a collection in which we look up values by **key**.



Q. What value is associated with the key 1?

A. "gg".

We look at data abstraction in the second half of the semester.

Operator Overloading

Operator Overloading Basics [1/2]

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-`.

> **Overload**: use the same name for two things.
>
> **Arity**: number of operands.

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

> **Global**: declared neither inside a class nor inside a function.

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

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

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

> **Member**: Declared inside a class.

Operator Overloading Basics [2/2]

Operators with the same symbol are distinguished by parameters.

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

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

```
class Num {
public:
    Num & operator++();          // ++a
    Num operator++(int dummy);   // a++
```

Why are different return methods used here?

Operator Overloading
Stream Operators [1/2]

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:

```
cout << a << b;  // Same as (cout << a) << b;
```

Operator Overloading
Stream Operators [2/2]

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.

```
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.

Operator Overloading
Global vs. Member [1/2]

Global function:

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

Member function:

Which is better: global or member?

```
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. ☹

Operator Overloading
Global vs. Member [2/2]

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.

Software Engineering Concepts: Assertions

Software Engineering Concepts: Assertions Introduction [1/2]

An **assertion** is a statement made in code that something must be true—or else the code is not working properly.

Many programming languages allow for assertions, often via a function or function-like thing named "`assert`".

For example, here is Python.

```
def get_first_item(mylist):
    assert len(mylist) > 0
    return mylist[0]
```

Typically, `assert` is given a **Boolean expression**—something true or false. If the expression is true, then `assert` does nothing; otherwise, `assert` crashes with an explanatory message.

Software Engineering Concepts: Assertions
Introduction [2/2]

We want our code to crash? Why?

If we find a bug while developing it, then yes, we do, because of an engineering concept called **fail-fast**. A fail-fast system that detects a flaw in itself will cease operating so that the flaw can be fixed, rather than continuing a flawed process.

We generally apply the fail-fast idea to software systems when they are under development.

Software Engineering Concepts: Assertions
assert in C++ [1/3]

The C++ Standard Library includes `assert`. It is defined in the header `<cassert>`.

```
# include <cassert>  // For assert
```

`assert` looks like a function, but it is actually a preprocessor macro, so there is no "`using`".

`assert` takes an expression of type `bool`, which it evaluates. If the result is `true`, then `assert` does nothing more; otherwise, it crashes with a message, which typically gives the source file, line number, and the expression that was `false`.

```
assert(n >= 0);
```

Software Engineering Concepts: Assertions
assert in C++ [2/3]

The behavior of `assert` that was described is what happens if the preprocessor symbol `NDEBUG` is not defined. If this symbol is defined, then `assert` does nothing at all; it does not even evaluate the given expression.

If you use a C/C++ integrated development environment (IDE) that has debug and release builds, then it is likely that `NDEBUG` is not defined in debug builds, but is defined in release builds. We generally do not apply fail-fast to code that is released or shipped to customers.

Software Engineering Concepts: Assertions
assert in C++ [3/3]

We use `assert` by placing it as a statement in normal code. It indicates something that must be true, or else there is a bug in the code.

So an appearance of `assert` serves as a kind of active comment. It tells the reader something, but it also *does* something.

TO DO

- Write some C++ assertions using `assert`.

See `assertion.cpp`.