C++ (C# & Java, too) concrete objects and essential application data

- *Conrad Weisert,*
  Information Disciplines, Inc.
  `www.idinews.com`

- For Chicago C++ Users' Group September 27, 2016

- **`cweisert@acm.org`**

Agenda

- Background: C and its successors
  - The problem
  - Why do we care?
- Elementary data items
  - 1. **Text** data
  - 2. **Numeric** data
  - 3. **Discrete** data
- Looking ahead;
  - **Composite** data;
  - Further discussions

+ bonus topic, if time permits

Background

- The **original C** language (c. 1979) lacked (*why?*) built-in data types for:
  - a. **text** (character-string) data items
  - b. **decimal** numeric data items
  - c. independent **discrete** data items
- But applications (esp. business/commercial) need to store and manipulate such data items.
- COBOL, PL/I, and other older programming languages supported **decimal** and **text** data as built-in types.

Was that a good idea? Is there a better way?

What could be done about that?

a. Within the **original C language**?

- clumsy, extremely error-prone standard library for **quasi-string** handling (arrays of **`char`** with null terminator)
  `strcpy(s1,s2)`, etc. *Forget it!*
  Often a separate statement for each operation, like assembly language programming.

  Another opinion *(What was he thinking?)*!

  "*One of the great strengths of C from its earliest days has been its ability to manipulate sequences of characters.*"
  -- P. J. Plauger, C/C++ Users Journal, July, 1995
- no standard **decimal** facility; just do your own scaling, as we did on early binary computers.

What could be done about that?

b. With **object-oriented languages** (**C++, Java, C#**) derived from C?

- Early realization that the facilities for defining O.O classes could also be used to define **string** and **decimal** types.
- Therefore, major breakthrough! We don't need to build those types into the language. Just provide library **classes**.
  (But that took a while and still isn't fully satisfactory.)

The trap

- The existence of **`string`** and **`decimal`** library **classes** misled naive programmers
  - They assumed that their data items declared as instances of those classes) were **object oriented**.
  - **Wrong!** Those **`string`** and **`decimal`** classes were just substitutes for *primitive* types that are built into other programming languages. They do not support the properties of most application data.
  - Data items that are just instances of those classes lack the automated power and error protection that we associate with the benefits of OOT.

Let's look at some examples . . .

Quick review of language-specific facilities for defining operators on objects

- **C++:** To define the behavior of a binary operator on one or both objects of a class **T** either:
  A. as a **member function** (left side implied; the object.)
  ```
  T  operator*(const T rs) const
  ```
  B. or as a **non-member function** (both operands explicitly declared):
  ```
  T operator*(const T ls, const T rs)
  ```
- They do the same thing, invoked by
  ```
  . . . obj1 * obj2 . .
  ```
  in an expression

When would you prefer each of those?

language-specific facilities for defining operators on objects

- **C++** (continued): For efficiency it is strongly recommended that we:
  - Define the compound-assignment operator as primitive:
    ```
    T&  operator*=(const T rs)
    ```
  - Define the *simple* arithmetic operator in terms of the corresponding *compound assignment* operator:
    ```
    T operator*(const T ls, const T rs)
        {T result = ls;
         return ls *= rs;
        }
    ```
    Why is this recommended?

language-specific facilities for defining operators on objects

- **C#:** You can't define the compound assignment operators, just the simple ones:
  - But users can still **use** the compound assigment operators. We hope the compiler figures out how to avoid creating a new object for `*=`
- **Java:** Hopeless!
  - You can't overload operators or use ordinary expression syntax for your own classes! You have to define named functions, so the user codes
    `. . obj1.multiply(obj2) . .`
    in an expression.
  - Such code is hard to read, but usually better than not defining objects at all.

Why doesn't Java support operator overloading?

- Java doesn't support *value* objects.
  `https://en.wikipedia.org/wiki/Value_object`
- Strange notions about *readability*:
- ". . . The language designers decided (after much debate) that overloaded operators were a neat idea, but the code that relied on them became hard to read and understand."—David Flanagan: Java in a Nutshell, 1996, O'Reilly & Associates, p. 35.
- "One of the major problems with operator overloading is that it gives the programmer the power to easily write code that is difficult to read."—Paul Tyma, Gabriel Torok, and Troy Downing: Java Primer Plus, 1996, Waite Group Press, p. 254

Part 1 (of 3) Text (or `string`) data items

- The standard **`string`** classes in C++, Java, and C#
- Examples:
  1. short fixed-length **identifiers**
  2. **names** of people
  3. book **titles**
- Other necessary **`string`** classes

Standard string classes

- C++, Java, and C# all provide a standard library **`string`** class:
  - The **good**:
    - Natural expression syntax for manipulating data
    - The usual (expected) operators and functions
    - Huge maximum length

    Good start, but is it sufficient?
  - The **bad**: (complicating programming)
    - No fixed-length objects
    - No contiguous objects within a containing structure.
    - Separate incompatible format & style for (Java) **`stringBuffer`**.

Text example 1: fixed-length identifiers

- Many data items used as identifiers (**`SSN, ISBN, UPC, AcctNo`**, etc.):
  - have either fixed length or varying length with a small maximum
  - may contain digits, letters, and some punctuation
- Declaring them instances of **`string`** is **extremely inefficient** and complicates programming:
  - non-contiguous storage complicates and slows I-O, comparisons, parameter passing, etc.
  - may accept illegal characters, absurd length, etc.
  - unnecessary length field wastes space
- *What can we do instead in C++?*

A C++ Solution:

- **Fixed-length** strings can occupy **contiguous storage** within a containing structure.

```
class  Book {
   Cstring<14> ISBN;
      .
      .
```

- So when we (shallow) move or copy a **Book** object, the ISBN moves right along with it.

What's a **`Cstring`**? (more later)

Fixed-length contiguous strings

- C++'s **class template** facility allows us to define a *constant-length string* class in which:
  - The string value is stored **inside** the containing structure, i.e. within a containing record. There are no pointers to non-contiguous memory.
  - That simplifies programming and yields efficient code.
- Downside: Compiler may generate near-duplicate code if you define multiple **`Cstring`** classes.
  - i.e. **`Cstring<8>`** is a separate class from **`Cstring<10>`**!

Text example 2: Names of people (international style)

- One of the most frequently needed data items in business applications.
- A common textbook solution uses three strings!
  -
```
string  last;
string  first;
string  middle;
   or
string[3] name;
```

What's wrong with that?

Text example 2: Names of people (continued)

- Common textbook solution uses three strings!
  - `string last;` `string first;` or `string[3] name;` `string middle;`
  - Each component
    - occupies non-contiguous storage pointed to from a containing record
    - can be between 0 and 65535 (**wow!**) characters!
    - carries a length field (2 bytes? 4 bytes?)
  - Can we tell **George Herbert Walker Bush** and **Cher** to change their names?
  - What if we need to store a Chinese name? **Sun Yat-sen, Mao Zedong**

One better solution: (There are others)

- **Single string** with comma following family name: `<familyname>,<given names>`
- No other restrictions on punctuation
  - Examples: `Bush,George Herbert Walker` `De Gaulle,Charles` `Mao,Zedong`
  - Need to specify **maximum** size for the **whole** name *(why? how big?)*
- *Advantages? Disadvantages?*

One better solution: *(continued)*

- Advantages:
  - Easy to sort and compare
  - Easy to reformat for polite envelope address, etc. `George Herbert Walker Bush`
  - Much less wasted storage than multiple strings.
  - Accommodates non-European-style names, as long as there's a **family name** having no embedded comma
  - Allows hyphens and other punctuation (except comma) within any component.
  - Flexible length of each component. Max. **size** applies to *whole* name, not to each component.
- Disadvantage:
  - Still non-contiguous with containing record (Are we stuck with that awkwardness?)

Text example 3: Book

(or other literary or performed work) **title**

- Why do we need a standard?
- Why not just use:
  - what the publisher provides?
  - whatever is printed on the cover?

Problems with titles

- Where would you expect to find in a *sorted* catalog:
  - "The Decline and Fall of the Roman Empire"
  - "Das Kapital"?
  - "A Tale of Two Cities"?
  - "La Traviata"?

For most purposes that list is already sorted (assuming that's the external representation)!

What's the absolute simplest acceptable standard?

Possible solutions for the internal representation of titles

a. Move the prefix article to the end, with a separator character that never appears within a title:

```
title = "Tale of Two Cities^A";
```

b. Or use two strings:

```
titlePfx  ="The";
titleBody
 ="Decline and Fall of the Roman Empire";
```

c. Again we must specify a maximum length.

Any other good ideas?

Summary: Classes for text data items: Do we need more than `string`?

- **`string`** is a relatively recent addition to the C++ standard class library (and thus to the language)
  - huge maximum size
  - non-contiguous memory
  - Java and C# emulated it, more or less
- What else could anyone possibly need? Doesn't that serve any conceivable requirement?

Shocking discovery in programs we've examined

- Many data items declared simply as instances of **`string`** shouldn't be!
- They should be instances of a specific **class** in order to enforce:
  - maximum (& minimum) length
  - formatting and editing rules
  - conversion rules
  - sorting sequence
  - default value, if any
- Those classes can **use** standard **`string`** internally when it's appropriate.

So, which data items are OK to declare as raw `string`?

- General text
  - No definite format or structure
  - Wide range of size
- Examples:
  - articles,
  - messages,
  - book chapters,
  - contracts,
  - laws,
  - correspondence,
  - . . .)

But not most names, titles, identifiers, etc.

Our (IDI) string classes

- Earlier (before std. **`string`**) we implemented **four** string classes, which we still find useful for localizing common kinds of processing:
  - **`Dstring`**: **Dynamic**
    - like what became the standard library **`string`** class.
  - **`Fstring`**: **Fixed length**
    - truncate if too long, pad if too short
  - **`Vstring`**: **Varying length**
    - up to a maximum like PL/I **`varying`**
    - Avoids re-allocation upon **`+=`** (non **`const`**)
  - **`Cstring`**: **Constant length**
    - contiguous with containing **`struct`** (mostly short)

Not only useful but needed!

Can we get those four string classes?

- Sure but not immediately:
  - We're revising the C++ version to substitute standard library **`string`** for our original **`Dstring`**
  - That's easy but the other three classes all interact with **`Dstring`**, so they have to be changed and thoroughly tested, too.
- When time permits, we'll do the same for the C# equivalents.
- If you're interested, contact me at

`cweisert@acm.org`
`(773) 736-9661`

Part 2 of 3
Numeric data items

- Issues with decimal scaling
  - Do application programs really need internal (member) **`decimal`**? If so, why?
- Examples:
  - **`Money`**
  - **`Date`**
  - **`Angle`**
  - **`. . .`**

Built-in decimal (history)

- Packed decimal format (2 digits/byte, 1963)
  - Started with COBOL (IBM extension)
    `. .PICTURE . . USAGE IS COMPUTATIONAL-3`
  - Supported by PL/I `. . . FIXED DECIMAL(m,n)`
- Not needed in object-oriented languages
  - **Integers** of various sizes are built-in.
  - Java, C#, and proposals for C++ provide "standard" numeric **decimal** library **classes**
    - Not compatible with **packed decimal** format (still found in old files)
    - C# `decimal` is a pair of `long` (128 bits!)
- Slower than binary on most computers, but may avoid scaling issues.

Properties of Numeric Data

- Do we *ever* need (or prefer) only internal **decimal** for concrete data items that have:
  - a **unit** of measure (internal),
  - a specified **range** (min & max),
  - and a definite **precision** (or accuracy)?
  - If so, for what? and why?
    - `Money?`
    - `Distance?`
    - `Temperature?`
    - `Elapsed time?`
    - `Time of day?`
    - `Weight?`
    - `Date?`

Some arithmetic operations are needed and others are illegal nonsense. (We showed those design patterns 4 years ago.)

- Answer: Hardly ever!
  (But we'll keep it in mind as an option.)

The argument for internal decimal

- Except for negative powers of 2 (1/2, 1/16, etc.) fractional quantities, such as 1/10, are non-terminating in binary representation.
  - Careless programmer may generate `$44.99999999`, unacceptable to end users and fussy auditors.
- Let's look first at amounts of **money**, since money has received considerable attention (and misinformation) in articles, textbooks, courses, and even language design.

Standard library decimal classes

- Standard library classes
  - C++ Proposal to add **Decimal** classes
    `decimal32, decimal64, decimal128`
    (See next page)
  - Java `BigDecimal` class
    (Precludes ordinary expression syntax in computations)
  - C# a `decimal` item consumes 16 bytes!
    (This is a `.Net` standard representation)
- NOTE: **None** of those is compatible with IBM 360 **packed decimal** (2 digits per byte), common in old files.

So, when do we need them?

"Proposal to Add Decimal Floating Point Support to C++"

`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3407.html`

- "In many areas, especially in finance, exact values need to be processed and the inputs are commonly decimal. Unfortunately, decimal values cannot, in general, be represented accurately using binary floating points even when the decimal values only uses a few digits. Instead, the values become an approximation. As long as the values are carefully processed the original decimal value can be restored from a binary floating point (assuming reasonable restrictions on the number of decimal digits). However, computations and certain conversions introduce subtle errors (e.g. double to float and back to double, even if float is big enough to restore the original decimal value). As a result, the processing of exact decimal values using binary floating points is very error prone."
- **Wrong!** Binary arithmetic is perfectly accurate with **integer** scaling. Just pick the right **units**!

The counter argument

- But avoiding such cases is easy and natural without internal decimal
- Define a **`Money`** class (or **`USMoney`**, etc.)
  - Just represent the object in **integer cents** (or whatever the finest increment will be)
  - Apply scaling at input time and output (display) time
  - Addition and subtraction of such money items work fine in a natural way.
  - So does multiplication or division by a pure number, e.g. a percentage or quantity calculation.
  - That was easy *(and commonly done 50 years ago) without* OOP. It's even easier *with* OOP.

Why wouldn't we do that?

Popular mythology and a C# "solution"

- From the stackoverflow web site:
  - `http://stackoverflow.com/questions/316727/is-a-double-really-unsuitable-for-money`
- This question was posed:

"I always tell in c# a variable of type **`double`** *is not suitable for money. All weird things could happen. But I can't seem to create an example to demonstrate some of these issues. Can anyone provide such an example?"*

What were some of the interesting replies?

Some early replies

1. "Very, very unsuitable. Use **`decimal`**.

```
double x = 3.65, y = 0.05, z = 3.7;
Console.WriteLine((x + y) == z); // false  "
```

2. "You will get odd errors effectively caused by rounding. In addition, comparisons with exact values are extremely tricky - you usually need to apply some sort of epsilon to check for the actual value being "near" a particular one.
   Here's a concrete example:

```
class Test
{  static void Main()
  {    double x = 0.1;
       double y = x + x + x;
       Console.WriteLine(y == 0.3); // Prints False
  }
}                      "
```

What was the real problem here?

And some utter confusion

3. "Yes it's unsuitable.

   If I remember correctly **double** has about 17 significant numbers, so normally rounding errors will take place far behind the decimal point. Most financial software uses 4 decimals behind the decimal point, that leaves 13 decimals to work with so the maximum number you can work with for single operations is still very much higher than the USA national debt. But rounding errors will add up over time. If your software runs for a long time you'll eventually start losing cents. Certain operations will make this worse. For example adding large amounts to small amounts will cause a significant loss of precision.

   You need fixed point datatypes for money operations, most people don't mind if you lose a cent here and there but accountants aren't like most people."

Finally: a few sane clarifications

4. "Depending on where you live, using 64 bit integers to represent cents or pennies or kopeks or whatever is the smallest unit in your country will usually work just fine. For example, 64 bit signed integers representing cents can represent values up to 92,223 trillion dollars. 32 bit integers are usually unsuitable."

5. "My understanding is that most financial systems express currency using integers -- i.e., counting everything in cents. IEEE double precision actually can represent all integers exactly in the range **-2^53** through **+2^53**. (Hacker's Delight, pg. 262) If you use only addition, subtraction and multiplication, and keep everything to integers within this range then you should see no loss of precision."

Why not division?

And the last word?

6. "Actually floating-point **double** is perfectly well suited to representing amounts of money as long as you pick a suitable unit.

   See **`http://www.idinews.com/moneyRep.html`**

   So is fixed-point **long**. Either consumes 8 bytes, surely preferable to the 16 consumed by a **decimal** item." **– Conrad Weisert, June 8, 2015**

But with a final dissent!

"Linking an article you wrote that disagrees with decades of common practices and expert options that floating-point is unsuitable for financial transaction representations is going to have to have a little more backup than a single page."

– MuertoExcobito June 8 , 2015 at 15:42

No further discussion has been posted after a year!

`Money` Conclusion

- The issue wasn't whether floating point is "suitable" for representing money.
- The issue was simply the choice of **unit** for the internal representation.
  - Either **`double`** or **`long`** works just fine for representing integer **pennies** (or whatever the smallest unit is),
  - and either of them also
    - consumes much **less space** (half) than the popular **`decimal`** class objects
    - yields **fast computation** on most computer architectures.

So what good are those standard library `decimal` classes?

- So far, organizations I've worked with have found **no use** for them in representing application data in ordinary business or scientific applications.
- They may be useful in theoretical *number-theory* research studies.
  - *Anything else?*

Numeric example 2: Date

- C++, Java, & C# provide standard **`Date`** classes.
  - Some combined with **`TimeOfDay`**
    (*Is that a good idea?*)
  - Some impose weird representation conventions (e.g. January is month **0**, 2000 was year **100**)
  - and amateurish function names
    (**Date** *is both the day-of-month function and the class name!*)
  - Few provide necessary operations
    (*Which ones are necessary?*)
- Don't even think about using those ugly *standard library* classes in a serious program.

Operations on standard (C++) library `Date` (common confusion)

Q1: What's the result of **adding** one date to another date?

- I own two textbooks that define overloaded **+** for two **Date**s yielding a **Date** result!
- So what **Date** is
  `July 4, 1776 + September 7, 1941` ?

Q2: What's the result of **subtracting** one date from another date?

- Standard **Date** classes don't support this operation, which should yield a **Duration** (no. of days)**.**
- So what is
  `January 1, 2017 - December 25, 2016` ?

Operations on standard library `Date` (continued)

Q3: How does a program compare two **`Date`**s?

- **`d1<d2`** is not supported!
  (needed by library **sort** routines)
- neither is **`d1.lessThan(d2)`**
- use **`d1.before(d2)!`** (Java)
  (not used anywhere else.
  Undermines generic programming)

- How did that happen?
- What should a programmer do?

Conclusion for Dates

- Applications need a **pair** of related interacting classes:
  - **Date** (*point*-in-time) (See note below)
  - **Days** (*extent*-of-time or duration)
- **Date** range must span the Gregorian (or an alternative) calendar and beyond.
- See **`http://www.idinews.com/NoDate.html`** for more details.
- **Note:** Whether to combine **date** and **time of day** in a single value is controversial (and messy) and depends upon the application. We needn't discuss it here.

Numeric example 3: Angle

- **`Angle`** follows the same pattern as **`Money`**, **`Weight`**, **`ElapsedTime`**, except for one minor issue.
  - It's the **additive design pattern**, but the result of any operation must be between **-pi** and **+pi**
- Q: Why would a programmer using a language that supports OOP choose *not* to use an **`Angle`** class for any application that deals with plane angles?
  - Why don't C++, Java, and C# standard libraries, which already provide all the **trigonometric functions**, provide those classes?

Anti-OOP libraries

- The three C-like languages provide the usual trigonometric fuctions as library routines, but no standard **class** to define the angles themselves!
  - It's as if they wanted to discourage the use of the object paradigm.
  - Java is (as expected) the worst or the three, because it doesn't support expression syntax for objects.
  - That's like coding in **Fortran**.

Part 3 of 3
Discrete data items

- Definition
- Small examples:
- Large examples:

Discrete (or coded) data items

- Possible values:
  - May be a *small* stable list (**`maritalStatus`**) or a *large* (even unlimited) frequently changing one (**`telephoneNumber`**),
  - Never enter into arithmetic calculations,
  - But some may be compared for sequence (with the relational operators) or sorted in lists,
  - Some may be used as **identifiers** (Part 1 of this presentation), usually internally **`string`** or **`Cstring.`**

Two broad categories:

- **Small** number of possible values:
  - **`sex`** (or gender) of a person or animal
  - **`maritalStatus`**
  - **`color`** of a product
- **Large** number of possible values:
  - **`telephoneNumber`**
  - **`postalCode`** (ZIP)
  - **`employeeNumber`**

"Number" in the data name doesn't make it a numeric item. What would?

- We never do arithmetic on them, but we may sort them.

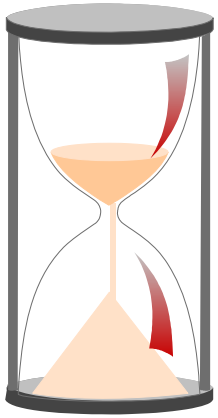
Why bother with discrete item classes?

- Assures compliance with standard forms:
  - Is the customer's phone number
    - **`(202) 393-1200?`**
    - **`202 393-1200?`**
    - **`202 3931200?`**
    - **`(202) EXecutive-3 1200?`**
    - **`(202) EX3-1200?`**
- Why do we care?
  - Can we produce a sorted list?
  - Will we avoid errors? **`202 393120`**
  - Can we avoid mischievous input
    `(919) 393-1200?`

An obvious construct for small number of values that don't change often

- **`maritalStatus`** is a good example
  - **`unknown`**
  - **`single`**
  - **`married`**
  - **`divorced`**
  - **`widowed`**
- The **`enum`** type is obvious for internal representation.

Conclusions



Summary questions

- Q1: The O.O. languages support primitive (non-O.O., built-in, inherited from C) data. Is it **wrong** to declare application data items as instances of those primitive types?
- Q2: What about instances of **`string`** and **`decimal`**?
- Q3: What about Java? Manipulating **primitive** data versus **objects** is almost two separate languages, and expression syntax is ugly.

I hope you're convinced that with object-oriented technology:

- Every **numeric data item** that has a specified:
  - unit of measure,
  - range,
  - precision
- and every **text data item** that identifies (or helps to identify) an object
- and every **discrete data item** that gives the value of an option property

should be an instance of a well-defined error-free, simple, and flexible **class**.

Looking ahead: For another session?

- Tonight we've focused on **elementary** (text, numeric, & discrete) data items and we've drawn useful conclusions.
- What about **composite** data (entities, records, structures). Which principles that we just discussed apply to them, too, and how? In particular:
  - The notorious **`Person`** class
- What about **container** classes? (Java calls them "collections")

BONUS TOPIC
(optional)

- I've been asked to show again the electric circuits example that we examined three years ago:
  - It illustrates the power of OOP in simulating real-world systems.
- It will take about six minutes
- Those who aren't interested are free to leave now, without hurting the speaker's feelings.

Direct Current Electrical Quantities

- Some types (classes) required:

Data type	Unit of measure
Potential	Volts
Current	Amperes
Power	Watts
Resistance	Ohms

- Some operations:
  - power = voltage * current
  - voltage = current * resistance

Defining the required operators

- Ohm's law, `V = I * R`
  demands **4** operator definitions for
  - `V = I * R`
  - `V = R * I`
  - `I = V / R`
  - `R = V / I`
- So does `P = I * V`
- That's awfully tedious to prepare

Why not just define the ones our own program needs? **("YAGNI")**

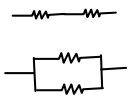
Is it worth the bother?

A troublesome operator issue

- Electrical engineers often want to combine the two earlier formulas
  `V = I * R` and `P = I * V`
  into
  `P = I * I * R`
- What would we need to define to support that? What problems arise?
  - No problem if the user-programmer parenthesizes
    `P = I * (I * R)`
    but must we require that?

A special operator opportunity

- In modeling a circuit we'd like to express combining resistances both in series and in parallel.
  - Series is just addition
    R = R1 + R2
  - Parallel is
    R = (R1 * R2) / (R1 + R2)



- C's *bitwise* boolean operators suggest a simpler notation
  - Series combination: **R1 & R2**
  - Parallel combination: **R1 | R2**

We can build that. Should we?

A raging controversy

among OOP insiders

- Never define any operator to mean anything other than what it originally meant in C
- If you do, it will confuse the reader of a program and impair readability.

- Nonsense! Operator notation is simple and natural.
- There's precedent in standard C++ *Where?*
- There's no chance of anyone's misinterpreting **|** or **&** between resistances *Why?*

Which point of view is more sensible?

Code fragment and output #1

```
cout << endl << "1:  Simple voltage, current, and power" << endl;

Potential  v1 = 115.0;    display(v1);
Current    c1 =  15.0;    display(c1);
Power      w = v1 * c1;   display(w);
display(w / c1);
display(w / v1);
```

What is `display(x)`*?*

```
1:  Simple voltage, current, and power
  v1 = 115 volts
  c1 = 15 amperes
  w = 1725 watts
  w / c1 = 115 volts
  w / v1 = 15 amperes
```

Code fragment and output #2

```
cout << endl << "2:  Resistances in series and in parallel" << endl;
Resistance r1 = 6.0;      display(r1);
Resistance r2 = 4.0;      display(r2);
Resistance r3 = r1;       display(r3);
display(r1 & r2);
display(r1 | r2);
display(r1 & (r2 | r3));
```

```
2:  Resistances in series and in parallel
r1 = 6 ohms
r2 = 4 ohms
r3 = 6 ohms
r1 & r2 = 10 ohms
r1 | r2 = 2.4 ohms
r1 & (r2 | r3) = 8.4 ohms
```

Code fragment and output #3

```
cout << endl << "3:  More complicated computations" << endl;
cout << v1 << " across "  << r2  << " gives "    << v1 / r2 << endl;
cout << c1 << " through " << r2  << " requires " << c1 * r2
          << " and uses "<< c1 * (c1 * r2) << endl;
cout << "4 60-watt bulbs use "    << 4 * Power(60)
            << " and draw "      << Power(60) / v1 * 4
            << " at "            << v1 << endl;
cout << "A " << v1 << ", " << c1  << " circuit can support "
            << int(v1 * c1 / Power(60)) << " 60-watt light bulbs.";
```

```
3:  More complicated computations
115 volts across 4 ohms gives 28.75 amperes
15 amperes through 4 ohms requires 60 volts and uses 900 watts
4 60-watt bulbs use 240 watts and draw 2.08696 amperes at 115 volts
A 115 volts, 15 amperes circuit can support 28 60-watt light bulbs.
```

Thank you

- *Conrad Weisert*
- **cweisert@acm.org**
- **(773) 736-9661**
- **www.idinews.com**
  (usually updated monthly)
- **Information Disciplines, Inc.**
  4620 N. Austin Avenue
  Chicago 60630

