# A `Person` class

Is it

- possible?
- practical?
- worth the effort?

Chicago C++ Users' Group
November, 2013

# Why is this interesting?

- **Person** objects occur in many applications.
- Some components (member data) of a **Person** object are themselves candidates for O.O. treatment.
- Existing application systems are littered with badly flawed examples. So are textbooks and courses
- Balancing our agenda:
  - Recent CPPUG programs have been about **new** language features.
  - These C++ capabilities have been exploited for over a decade.

# What's wrong with this?
## (two serious flaws)

```
class Employee {
    string firstName,
           familyName;
    char   middleInitial;
        .
        .
        .
}
```

- Bjarne Stroustrup: *The C++ Programming Language*, Third Edition, p. 304

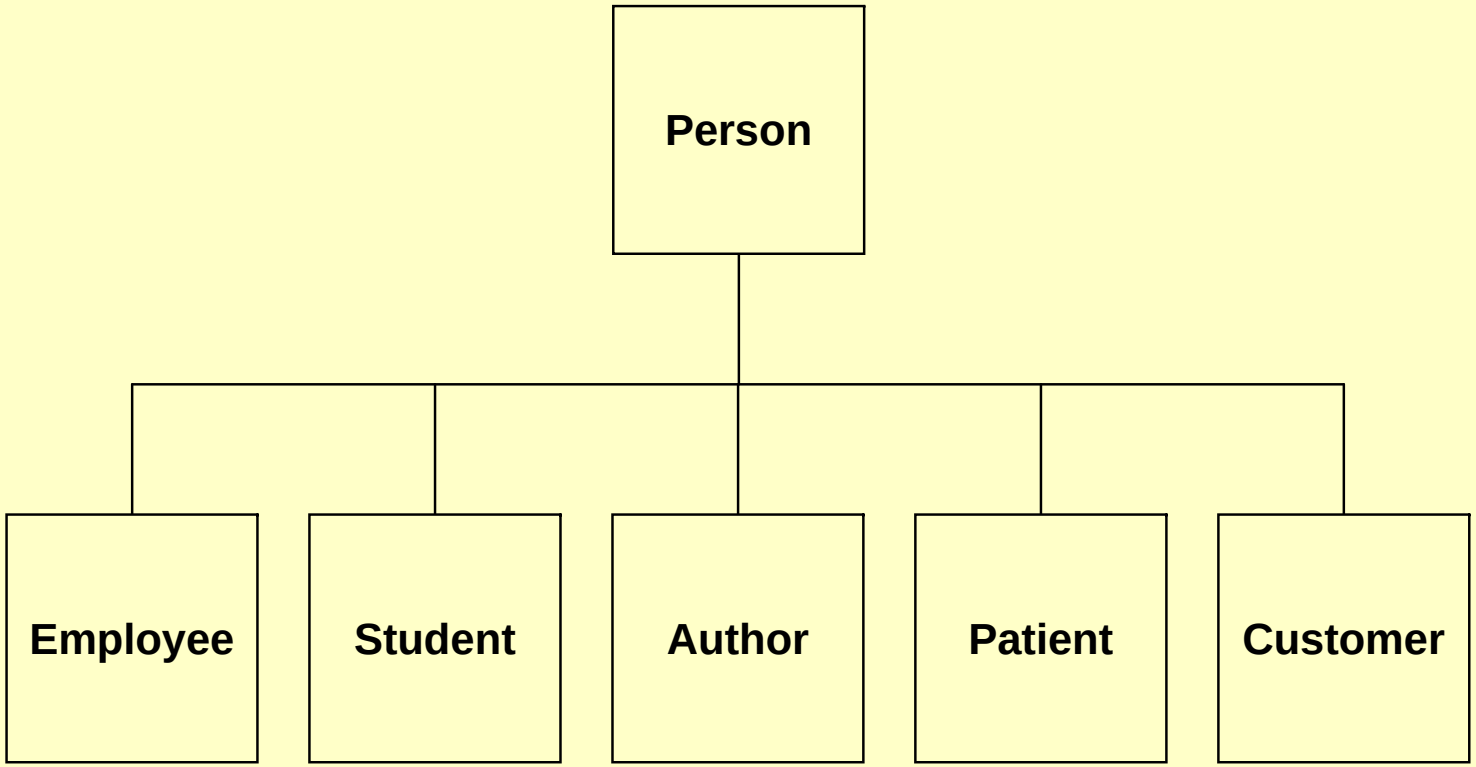
# Background

- Many applications deal with records representing individual *people*;
- For example:
  - Personnel systems (**`Employee`**)
  - Medical systems (**`Patient`**)
  - Academic systems (**`Student`**)
  - Library systems (**`Author`**)
  - Billing systems (**`Customer`**)

# Question

- Is it possible to define a **`Person`** class that will support *all* those application systems?
  - If so, is it desirable?
  - worth doing?
- Some say: ***No.***
  - The needs of each application area are unique.
  - You'll just distort the application if you try.
- Others say: ***Of course.***
  - Otherwise what good is OOP?

# Can (should) we exploit this kind of inheritance?




An **`Employee`** is a **`Person`**, etc.

# Obstacles to inheritance

- Suppose the same Person is both a **`Student`** and an **`Employee`**
  - Then we'd have two objects
  - Would they be equal? Should they be?
- How would we specify accessibility?
  - Number of **dependents** should be **`public`** for an **`Employee`**,
  - But none of anyone's business for a **`Customer`**
- Those "subclasses" aren't really special *kinds* of **`Person`** (**is-a** relation)
  - They're **roles** that a Person may play.
  - **Role** is another kind of object, another **class**.

# Relationship between Role and Person

- Natural language (esp. English) is misleading.
  - It sounds reasonable to say that a **Student** **is a** **Person**.
- But actually:
  - A **Person** ***plays*** a **Role** (maybe more than one)

    or
  - A **Role** ***is assigned to*** a **Person** or *vice-versa*.
- In OOP those are better represented by a **has-a** (membership) relationship, even though that departs from common English usage.

*Which way?*

# Part 1: a basic Person class

- Let's first focus on designing a **`Person`** class without having to be concerned about roles.

- A **`Person`** object should represent properties that are common to *all* Persons and are relatively stable.

  *What are those properties*

# Basic `Person` properties

**Permanent** properties

- **`dateOfBirth`**
- **`name`**
- **`sex`**

Change is extremely rare

**Stable** properties

- **`maritalStatus`**
- **`citizenship`**

Change is infrequent

**Transient** properties

- **`mailingAddress`**
- **`telephoneNumber`**

Change may be frequent

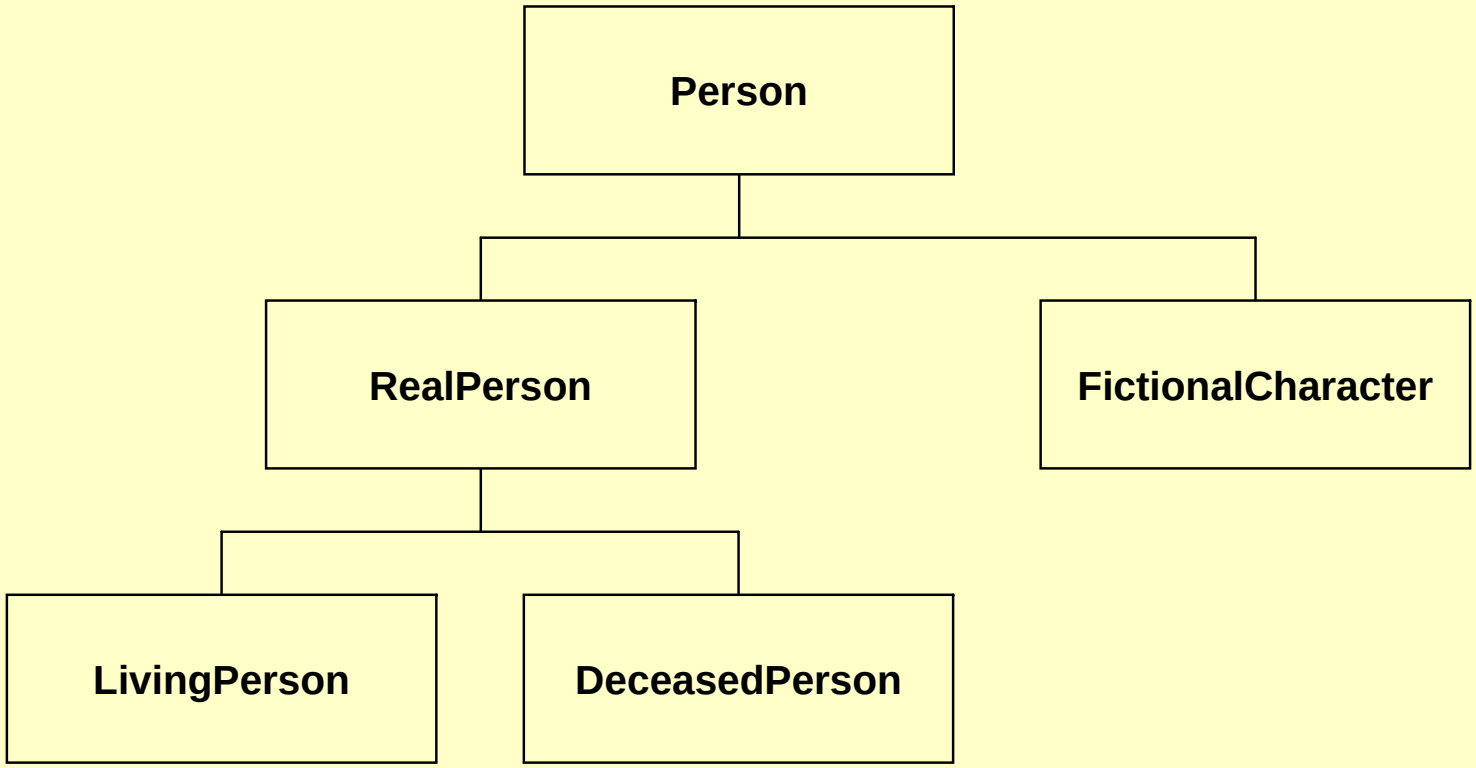
*What else?*

# Let's focus first on the permanent properties

- The other properties can be added later or delegated to
  - subclasses
  - role classes
  - databases
- A general **`Person`** class can be useful in all applications.

# Another hierarchical issue

*Should we be concerned about this?*



# Let's focus on `RealPerson`

- Every **`LivingPerson`** eventually becomes a **`DeceasedPerson`**
  - It's cumbersome to have to destroy one object and create another one when someone dies.
- So let's not bother with a cumbersome subclass arrangement for those.
  - Just include a **`dateOfDeath`** field (**`null`** for a **`LivingPerson`**).

# The basic `Person` class

- Member data items (the permanent properties)
  - **`name`**
  - **`dateOfBirth`**
  - **`dateOfDeath`**
  - **`sex`**

- Questions:
  - *What's missing?*
  - *What data type is each of those members?*

# The identity problem

- Suppose we have
  - **personA**
    - **name = "George Jefferson"**
    - **sex = male**
    - **dateOfBirth = June 14, 1964**
    - **citizenship = USA**
  - **personB**
    - **name = "George Jefferson"**
    - **sex = male**
    - **dateOfBirth = June 14, 1964**
    - **citizenship = USA**
- *What should*

**personA == personB**

*return?*

# The identifier problem

- The Boolean **`equals`** predicate (Java) or the **`==`** operator (C++) must return:
  - **`true`**, or
  - **`false`**
- It may *not* return
  - **`maybe`**,
  - **`probably`**, or
  - **`possibly`**
- Therefore each **`Person`** object should have a unique **`identifier`**
  - Who assigns them?
  - What's their scope?

# Identifier

- Many organizations assign their own, often specific to an application:
  - **`customerNumber`**
  - **`studentNumber`**
- But we still can't tell if a **student** and a **customer** are the same person.
- In the United States the **social security number** (SSN) is the only universal ID
  - It's not to be used for other purposes.
  - But everyone does, anyway.
  - And you have to *apply* for one



copyright Information Disciplines, Inc.

# So let's assume:

- A **`Person`** object contains an **`ID`** field
- The **`ID`** can be *anything*, as long as it's unique within the scope of the applications that encounter it.
- Determining equality may be unreliable.
- Some versions may allow *two* **`ID`** fields
  - An **interim** one (usually a sequence number), if the permanent one is unknown at the time the object is first created.
  - A **permanent** one, once we know it.

# A C++ Person class

(rough version)

```
public class Person {
   ????    ID;
   ????    name;
   ????    dateOfBirth;
   ????    dateOfDeath;
   ????    sex;

 //  Constructors
    .
    .
 //  Accessors
    .
    .
 //  Relational operators
    ..  etc.
 }
```

*What are the missing type codes?*

# A naive version

```
public class Person {
   long     ID;
   String   name;
   Date     dateOfBirth;
   Date     dateOfDeath;
   bool     sex;

 //  Constructors
    .
    .
 //  Accessors
    .
    .
 //  Relational operators

 }
```

*What's wrong with that?*

# We're trying to use the object-oriented paradigm

```
class Person {
   ????   ID;
   ????   name;
   ????   dateOfBirth;
   ????   dateOfDeath;
   ????   sex;

 //  Constructors
    .
    .
 //  Accessors
    .
    .
 //  Relational operators

 }
```

*Which of those members should be objects?*

# Consider a Person's name

- What components does it have?
- What format is it usually in?
  - English (**`first middle last`**)
  - Directory (**`last, first middle`**)
  - Other
- How long may it be?
  - Will it fit on a mailing label?
- Can a raw **string** satisfy those criteria?
- Can a Person's name appear elsewhere (not in a **`Person`** object)? Examples?

# Conclusion

- We need a **`PersonName`** class in order to:
  - standardize and enforce internal representation
  - facilitate data entry and retrieval
- It must be a top level class, independent of **`Person`**.
  - Then **`PersonName`** objects can appear in contexts other than member of a **`Person`** object, e.g. in a directory listing or on a mailing label.
  - A **`Person`** object **has a** **`PersonName`**.
- What about international names?

# An obvious OOP generalization about names

- A **thing** is different from the **name of a thing**. Thus:
  - A **`Person`** is not the same as a **`PersonName`**
  - A **`City`** is not the same as a **`CityName`**
  - A **`Company`** is not the same as a **`CompanyName`**

- *But is that obvious to everyone?*

# What about Sex? or Gender

- *Three* possible values:
  - male
  - female
  - unknown

  *That rules out* **`bool`!**

  *What about an integer?*

  *a single character* **`(M, F, U)`**?
- Must it be standard
  - for an organization?
  - within a single application system?
  - throughout the C++ (etc.) world?

*Why not?*



Copyright Information Disciplines, Inc.

# Another class?

- It looks as if we'll need another class for **`Sex`** (or **`Gender`**).
- Since it has only three possible values and they're usually immutable, we should consider the **`enum`** facility, with three constant objects.

# Are we done yet?

- What about the **identifier** field?
  - We tentatively chose **`long`**, but some identifiers contain non-numeric characters.
  - We can use a **`String`**, but then there's no central format control or enforcement
- So we need a **`PersonID`** class.
- Aren't we proliferating too many classes?
- No, that's what OOP is about.

# An improved version

```
class Person {
   PersonID     ID;
   PersonName   name;
   Date         dateOfBirth;
   Date         dateOfDeath;
   Sex          sex;

 //  Constructors
    .
    .
 //  Accessors
    .
    .
 //  Relational operators

 }
```

*We've got work to do developing those lower-level classes*



Copyright Information Disciplines, Inc.

# Bad news for Java programmers!

- **`Date`** and **`Calendar`** are among the few numeric classes in the standard Java library. We'd like to use them if we can.
- But those are among the *worst* designed components of the standard Java library! They're extremely awkward, inflexible, and error prone. Some experts call them an "abomination".

*In what ways?*

- In any Java (maybe others, too) application where date manipulation and calculation are important, therefore, we prefer to use our own **`Date`** class.

# The bottom line (work plan)

- We need to develop (design, code, and test) classes, probably in bottom-up sequence.
  - **`PersonID`**
  - **`PersonName`**
  - **`Date`**
  - **`Sex`**
  - and finally **`Person`**.
- Isn't that an awful lot of work?
  - Yes, but *all* of those classes are potentially **reusable**. That justifies the investment.

# PersonName internal representation: popular alternatives

1. Separate fields

```
string firstName;
string middleName;
string lastName;
```

*The choice is clear! (but incomplete)*

2. Polite form:

```
string name;    //  First Middle Last
```

3. Directory form:

```
string name;    //  Last,First Middle
```



copyright Information Disciplines, Inc.

# Person Name: eliminating the bad choices

- Separate fields form:

```
string firstName;
string middleName;
string lastName;
```

*Why did we eliminate those?*

- Polite form:

```
string name;    //  First Middle Last
```

- Directory form:

```
string name;    //  Last,First Middle
```

*What more is needed?*

# What's wrong with this

- `string  customerName = "Konerko,Paul";`

- The C++ **`string`** class by itself is too general, too broad, too permissive. (Java, C#, too).
- Persons' names must have a *maximum* **length** *Why? How do we pick it?*
- We may need to impose some restrictions on **format** or legal characters. *Why?*

(More later if time permits)

# Appending non-permanent (but usually stable) attributes

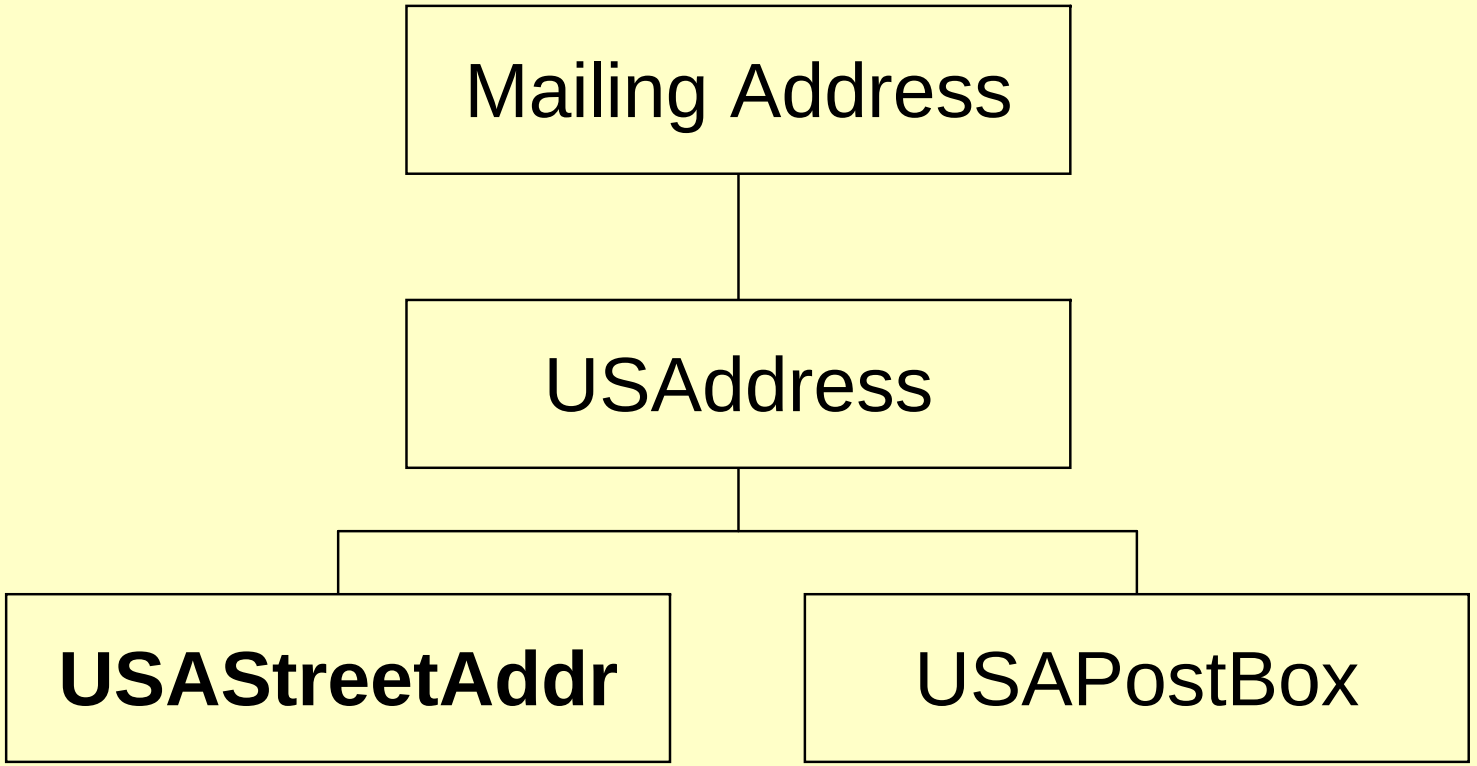
```
class Person {
   PersonID      ID;
   PersonName    name;
   Date          dateOfBirth;
   Date          dateOfDeath;
   Sex           sex;
   MaritalStat   mstat;
   Address       addr;
```

*What else?*

*Are these classes, too?*

*Let's tackle* **Address**

# The address hierarchy




*We'll focus on this. We can do others later.*

# USStreetAddr (another composite class)

- **Constraint:** Must fit on a standard mailing label. (about 40 12-point characters wide)

- **Components:**
  - streetAddress (1 or 2 lines X up to 40 characters)
  - cityName (up to 18 characters) *Why the limit?*
  - postalCode (ZIP) *String or numeric?*

*What happened to the state?*

# An important conclusion

- We've seen several text data items
  **`PersonName`**
  **`CityName`**
  **`StreetAddress`**
  for which a raw **`string`** class item raises problems or complications:
  - Decentralized knowledge and control of the data-item's length:
    - Truncating too-long results.
    - Blank-padding too-short results.
    - Complicated comparison logic.

*What's the solution?*

# What about marital status?

- A discrete data item with limited values:
  - single
  - married
  - divorced
  - widowed
- Another obvious candidate for an **`enum`** representation.

# Part 2: The too-popular `string` class

- We've seen several classes for which a character string is all or part of the obvious internal data representation.
  - **`PersonName`**
  - **`Address`**
  - **`CityName`**
  - etc., etc.
- **`string`** may be the most overused standard library class in C++ (also in C# and Java)
  - It's too general, too low-level for many kinds of object.

# Background

- C++ was available several years before the library **`string`** class.
- Therefore, rather than use C's crude character-handling facilities, organizations developed their own character string classes
  - Some of those were badly flawed, even those from major vendors.
  - Some were well-conceived and survive today.

# IDI's character-string classes: A: What we got right

- We developed and used four string classes:
  - **`Dstring`** (Dynamic) like the eventual standard library string class
  - **`Fstring`** (Fixed-length) truncates or pads with blanks as needed.
  - **`Vstring`** (Varying) pre-allocates space to avoid copying (cf. stringBuffer, StringBuilder)
  - **`Cstring`** (Contiguous)--Characters are inside the object. Avoids serializing, deep copying, etc.

*Why are all those needed?*

- **`Fstring`** and **`Vstring`** were inspired by PL/I.

# IDI's character-string classes: B: What we got wrong

- We retained C's null terminator for the internal representation
  - We could then use some of the C library functions in our methods, greatly reducing the implementation effort.
  - Programs could easily convert to and from C-style strings
- We regret that choice and are now engaged in re-doing those classes.
  - **`Dstring`** is no-longer needed, since standard library **`string`** duplicates its functionality and is surely more efficient (reference counting).
  - We still need the other three.

# Part 3: Associating persons with roles (relatively stable data)

- ```
  class Employee {
      Person    emp( . . . . );
      Date      dateHired;
      Position  ???;
      Money     Salary;
      short     noOfDependents;
    }
  ```
- or . . . ?

*What else?*
  ```

# Data-base records

- We can append many more data items to the **`Employee`** object.
  - Payroll data
    - hoursWorked
    - noOfDependents
    - deductions
    - vacationDaysDue
    - YTD stuff

*Similarly for the other role classes*

- Or we can just put them in a relational database
- But the important thing is that *none* of that has an impact on the definition of our stable **`Person`** class, which remains useful elsewhere.

# Epilog: Component library
## (A mission for our group)

- In the next few days I'm likely to get requests for some of the actual code I've described tonight.
- I'm happy to share anything, but instead of a bunch of two-way communications, can't our C++ User's Group establish a simple web-based **component library**?
  - Members would contribute their classes and other components.
  - Other members could download them at any time without bothering anyone.

*Should we monitor quality,*

*prevent utter garbage?*

# Contact


- Conrad Weisert
  Information Disciplines, Inc.
  4620 N. Austin Avenue
  Chicago 60630

- (773) 736-9661

- **cweisert@acm.org**

- **www.idinews.com**