Programming Paradigms

Robert M. Dondero, Ph.D.
Princeton University
Objectives

You will learn/review:

- The "unstructured" paradigm
- The structured paradigm
- The abstract object paradigm
- The abstract data type paradigm
- The object oriented paradigm
- The generic paradigm
- The functional paradigm
- The logic paradigm

Using C++
Goals

- **Primary goal:**
  - Give historical overview of programming
  - Describe programming paradigm shifts through time

- **Secondary goal:**
  - Describe seminal publications
  - Available via Blackboard

- **Another secondary goal:**
  - Describe C++
The C++ Language

- Who: Bjarne Stroustrup
- When: 1979
- Why: ...
C++ Design Goals

- Be a **statically typed**, general-purpose language
- Be as **efficient** and **portable** as C
- Support **multiple programming styles** (procedural programming, data abstraction, object-oriented programming, and generic programming)

Bjarne Stroustrup. 
*The Design and Evolution of C++.*
Addison-Wesley, Reading, MA 1994.
C++ Design Goals

- Give the programmer choice, even if this makes it possible for the programmer to choose incorrectly.
- Be as compatible with C as possible, therefore providing a smooth transition from C.
- Avoid features that are platform specific or not general purpose.

Bjarne Stroustrup. 
The Design and Evolution of C++. 
Addison-Wesley, Reading, MA 1994.
C++ Design Goals

- Do not incur overhead for features that are not used (the "zero-overhead principle")
- Function without a sophisticated programming environment

Bjarne Stroustrup.  
*The Design and Evolution of C++.*  
Addison-Wesley, Reading, MA 1994.
Scientific Paradigms

Thomas Kuhn
"A paradigm is what members of a scientific community, and they alone, share"

- Anomalies => state of crisis => new paradigm
- Science makes modest progress within paradigms
- Science makes revolutionary progress via paradigm shifts

Reality (more or less):

- Unstructured
- Structured
- Object based
- Abstract data type
- Object oriented
- Functional
- Logic
- Generic
Programming Paradigms

In this lecture:

- Unstructured
- Structured
- Object based
- Abstract data type
- Object oriented
- Generic
- Functional
- Logic
"Unstructured" Programming

- Title in retrospect!
- Example languages:
  - Machine languages
  - Assembly languages
  - FORTRAN (Formula Translator)
  - BASIC (Beginner's All-Purpose Symbolic Instruction Code)
"Unstructured" Programming

- Example program:
  - POLLY.BAS
Toward Structured Programming

- What's wrong?
  - From programmer's viewpoint?
Toward Structured Programming

Edsger Dijkstra
"My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the 'making' of the corresponding process is delegated to the machine."

"My second remark is that our intellectual powers are rather geared to master **static** relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to **make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.**"

Use of the **goto** statement makes the correspondence between the program and the process non-trivial

In my words...

A program:
- Is a *static* entity that specifies a process
- Has no time dimension

A process:
- Is a *dynamic* entity
- Has a time dimension
- Can be understood only in terms of its time dimension
People understand **static** things better than they understand **dynamic** things.

So the **static** structure of a program should be similar to its **dynamic** structure.
Toward Structured Programming

- Or, in other words...
- Suppose:
  - We have program written on paper 1
  - Each time computer executes a statement, we write that statement on paper 2
    - Thus generating a dynamic representation of the program for a particular set of data
Then consider the correspondence between paper 1 and paper 2

- Conditionals interfere
  - But only slightly
- Procedures interfere
- Iterations interfere

Nevertheless, for the sake of understandability...
Paper 2 should be similar to paper 1
- The **dynamic** rep of the pgm should be similar to the **static** rep of the pgm

And secondarily...
- If the static rep of the pgm contains **goto** statements, then paper 2 will be dissimilar to paper 1
- So **avoid goto** statements
Toward Structured Programming

Corrado Bőhm
Every computable function can be expressed as a nesting of only 3 control structures:

(1) Sequence

statement1

statement2
Toward Structured Programming

(2) Selection

TRUE

condition

FALSE

statement1

statement2
(3) Iteration

```
TRUE
\[\text{condition}\]
FALSE
\[\text{statement}\]
```
Toward Structured Programming

- Bőhm & Jacopini:
  - Any program *can* be expressed as the nesting of only 3 control structures
- Bőhm & Jacopini + Dijkstra
  - Any program *should* be expressed as the nesting of only 3 control structures
Structured Programming

- Key ideas:
  - Programming using only the nesting of the 3 elementary control structures: sequence, selection, iteration
  - Occasional exceptions are OK
  - See paper by Frank Rubin

Structured Programming

- Example languages:
  - ALGOL (*Algorithmic Language*)
  - C
  - Pascal
  - (C++)
Structured Programming in C++

Examples:
- polly.cpp
- SortedSeqSp/sort.cpp
Structured Programming in C++

- Notes on example program:
  - C++ I/O facilities
  - C++ "bool" data type
  - C++ references
Toward Abstract Obj Pgmmming

- What's wrong?
  - From programmer's viewpoint?
- Think about:
  - Design decisions
  - Modularity
Toward Abstract Obj Pgmmming

David Parnas
"In the first decomposition the criterion used was to make each major step in the processing a module. One might say that to get the first decomposition one makes a flowchart. This is the most common approach to decomposition or modularization."

David Parnas
"On the Criteria to be Used in Decomposing Systems into Modules."
"The second decomposition was made using 'information hiding' as a criterion. The modules no longer correspond to steps in the processing... Every module in the second decomposition is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings."

David Parnas
"On the Criteria to be Used in Decomposing Systems into Modules."
Abstract Obj Pgmming

- Key ideas:
  - Split programs into modules
  - Make modules abstract
    - Design modules to encapsulate important design decisions
    - Design modules to hide info from clients
Abstract Obj Pgmming

- Example languages:
  - Ada, (C++)
Abstract Obj Pgmming

- Example program: SortedSeqAop/
  - `sort.cpp` (client)
  - `sortedseq.h` (interface)
  - `sortedseq.cpp` (implementation)
Notes on example program:

- Major design decisions encapsulated
  - Which data structure?
  - Which sort algorithm?
  - When to sort?
Toward ADT Programming

- What's wrong?
  - From programmer's viewpoint?
Toward ADT Programming

Barbara Liskov
"An **abstract data type** defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type."

"We believe that the above concept captures the fundamental properties of abstract objects. When a programmer makes use of an abstract data object, he is concerned only with the behavior which that object exhibits but not with any details of how that behavior is achieved by means of an implementation."

"Abstract types are intended to be very much like the built-in types provided by a programming language. The user of a built-in type, such as integer or integer array, is only concerned with creating objects of that type and then performing operations on them. He is not (usually) concerned with how the data objects are represented, and he views the operations on the objects as indivisible and atomic when in fact several machine instructions may be required to perform them."

Barbara Liskov and S. Zilles. 
"Programming with Abstract Data Types." 
Key ideas:

- A module should be **abstract**
  - As in object-based programming
- A module can/should be a **data type**!!
  - Data type consists of data and operators applied to those data
  - Program can create as many objects of that type as necessary
- And incidentally... A lang should support exceptional control flow
ADT Programming

- Example languages:
  - CLU, (C++)
Example program: SortedSeqAdt1/

- `sort.cpp` (client)
- `sortedseq.h` (interface)
- `sortedseq.cpp` (implementation)
ADT Programming in C++

- Notes on example program:
  - C++ Classes
    - Data members (alias fields)
    - Member functions (alias methods)
C++ "new" and "delete" operators
  - Objects in the heap ("free store")
C++ constructors and destructors
  - Consider this code with (1) default destructor and (2) given destructor

```cpp
{
    SortedSeq seq = new SortedSeq();
    ...
    delete seq;
}
```
ADT Programming in C++

- C++ exceptions
  - Creating; throwing; catching
Example program: SortedSeqAdt2/

- sort.cpp (client)
- sortedseq.h (interface)
- sortedseq.cpp (implementation)
Notes on example program:

- C++ objects on the stack
  - Impossible in Java, Python
C++ assignment operators

Consider this code using (1) default asgt op and (2) given asgt op:

```cpp
{
    SortedSeq seq1;
    seq1.add(...);
    ...
    SortedSeq seq2;
    seq2 = seq1; // Calls asgt op
    ...
} // seq2 and seq1 go out of scope
```
C++ copy constructors

Consider this code using (1) default copy constructor and (2) given copy constructor

```cpp
void f(SortedSeq s) {
    ...
} // s goes out of scope
...
SortedSeq seq;
f(seq); // Calls copy ctr
...
ADT Programming in C++

- Proper constructor and copy constructor and asgt operator and destructor
  - => True value semantics for objects
- Improper constructor or copy constructor or asgt operator or destructor
  - => Incredible mess
What's wrong?
- From programmer's viewpoint?
Toward OOP

C.A.R. (Tony) Hoare
"In the preceding sections we have seen how the class mechanism is capable of modelling certain simple concepts, by specifying data structures and defining operations over them. In this section, we develop a method by which more elaborate concepts can be constructed on the basis of simpler ones. This will establish potential hierarchies of concepts, with complex concepts subordinate to the more simple ones in terms of which they are defined. The structuring technique gives a new method of composing a program from its constituent parts, and is known as concatenation."

"Hierarchical Program Structures."
Toward OOP

- Observations:
  - Real world objects can be classified, often hierarchically
  - Software objects can (and often should) model real world objects
  - So software objects can (and often should) be classified hierarchically

- Dahl & Hoare:
  - Concatenation (inheritance) via Simula lang
Key idea:

OOP = ADT + inheritance + dynamic binding
- **Inheritance**
  - A class can **inherit** the data (fields/attributes) and behaviors (functions/methods) of another class (or classes)

- **Dynamic binding**
  - Function calls bound to function definitions at **run-time** (instead of at **compile-time**)
Example programming languages:
- Simula, Smalltalk, C++, Objective-C, Java, Python, PHP, JavaScript
- And many others
OOP in C++

- Example program: SortedSeqOop1/
  - sort.cpp (client)
  - seq.h (interface)
  - seq.cpp (implementation)
  - sortedseq.h (interface)
  - sortedseq.cpp (implementation)
  - sortparallel.cpp (another client)
OOP in C++

- Notes on example program:
  - C++ inheritance
  - C++ protected data members and member functions
  - C++ static binding
Example program: SortedSeqOop2/

- `sort.cpp` (client)
- `seq.h` (interface)
- `seq.cpp` (implementation)
- `sortedseq.h` (interface)
- `sortedseq.cpp` (implementation)
- `sortparallel.cpp` (another client)
OOP in C++

- Notes on example program:
  - C++ *dynamic* binding, via...
  - C++ *virtual member functions*
Editorial

- "Inheritance is the GOTO statement of the 1990s"
- Recall Dijkstra's argument!
- Beware of deep inheritance hierarchies
Toward Generic Programming

- What's wrong?
  - From programmer's viewpoint?
"As mathematicians learned to lift theorems into their most general settings, so I wanted to lift algorithms and data structures. One seldom needs to know the exact type of data on which an algorithm works since most algorithms work on many similar types. In order to write an algorithm one needs only to know the properties of operations on data. I call a collection of types with similar properties on which an algorithm makes sense the underlying concept of the algorithm."

Generic Programming

- Key idea:
  - Using strongly-typed langs...
  - Define functions and classes that can work with multiple data types
- Non-issue in weakly-typed langs
Generic Programming

- Example languages:
  - Ada, C++, Java (recent versions), ...
Generic Programming in C++

- Example program: SortedSeqGen/
  - sort.cpp (client)
  - seq.h (interface & implementation)
  - sortedseq.h (interface & implementation)

- sortstr.cpp (another client)
Notes on example program:

- **C++ class templates**
  - From which the compiler generates **template classes**

- **C++ function templates**
  - From which the compiler generates **template functions**

- Must place templates in module interface!!!
Aside: Polymorphism

- **Polymorphism** ("many forms")
  - The ability of an object reference to refer to objects of different types, or...
  - The ability of a function to work with objects of different types

- Common theme:
  - How to achieve polymorphism in a strongly typed lang (such as C++)?
**Inclusion Polymorphism**

- **Inclusion** polymorphism
  - C++ mechanism: inheritance

```cpp
class T1
{
    void f(){...}
    ...
}

class T2: public T1
{
    ...
}
```

- `f()` works on objects of type T1 and T2
- … Because the set of objects of type T1 includes the set of objects of type T2
**Ad hoc polymorphism**

- **C++ mechanism: function overloading**

```cpp
class T1
{  void f() {...}
   ...
}
class T2: public T1
{  void f() {...}
   ...
}
void f(T1 t){...}
void f(T2 t){...}
```

- `f()` works on objects of type T1 and T2
- … But maybe in different *(ad hoc)* ways
Parametric Polymorphism

- **Parametric** polymorphism
  - C++ mechanism: function/class templates

    template <class T>
    void f(T t){...}

- $f()$ works on objects of any type that supports the operations performed by $f()$ on $t$
- ... And in the same *(parametric)* way
Non-Mainstream Paradigms

- Some non-mainstream paradigms:
  - Functional programming
  - Logic programming
Functional Programming

- Key ideas:
  - Treat computation as the evaluation of mathematical functions
  - Avoid state and mutable data
- Example languages: LISP, ML
- Example program
  - See sort.lisp
Logic Programming

- **Key ideas:**
  - Treat computation as logic
  - Consequent ← antecedent1 & antecedent2
  - To prove consequent, prove antecedent1 and antecedent2
  - Declarative, not imperative

- **Example language:** Prolog

- **Example program:**
  - See `sort.prolog`
Summary

- We have covered:
  - The "unstructured" paradigm
  - The structured paradigm
  - The abstract object paradigm
  - The abstract data type paradigm
  - The object oriented paradigm
  - The generic paradigm
  - The functional paradigm
  - The logic paradigm