Programming paradigms

Michael R. Nowak

Texas A&M University

October 12, 2017

Programming paradigm

- A **programming paradigm** is a style of programming
  - Imperative programming
    - Control flow is *explicit*
    - Commands show how the computation is to take place
      - C++ is an example
  - Declarative programming
    - Control flow is implicit
    - Commands focused on getting result, not on how to obtain it
      - SQL is an example
  - Functional programming
    - Control flow expressed as combination of functions
    - Programming without assignment statements; one just applies functions to arguments

Abstraction

- Abstraction | dictates that some information is more important than other information, but does not specify a specific mechanism for handling the unimportant information
  - As a process, denotes the extraction of the essential details about an item, or group of items, while ignoring the inessential details
  - As an entity, denotes a model, view, or some other focused representation for an actual item

Information hiding

- Information hiding | accomplished by restricting access to data, functions, types, etc. in order to ”hide” implementation details, while providing the user with an interface detailing *what the object does* instead of *how it does it*
  - At the language level, we will see that C++ provides public, private, and protected class members

Abstraction and information hiding

- The process of abstraction can be used as a technique for identifying which information should be hidden; for example,
  - Functional abstraction
    - It is important to add items to list, but the details of how that is accomplished are not of interest and should be hidden
  - Data abstraction
    - A list is a place where we can store information, but how the list is actually implemented (e.g., as an array, vector, etc.) is unimportant and should be hidden

Encapsulation

- Encapsulation |
  - As a process, the act of enclosing one or more items within a (physical or logical) container
  - As an entity, refers to a package or an enclosure that holds (contains, encloses) one or more items
  - In programming languages,
    - Functions, arrays, and structured types (classes, etc.) are common examples of encapsulation mechanisms

Programming paradigms

C++ supports many different programming paradigms; today we will talk briefly about:

- Procedural programming
- Modular programming
- Data abstraction (and user-defined types)
- Object oriented programming (OOP)
- Event-driven programming
- Generic programming

Procedural programming

Decide which procedures you want;
use the best algorithms you can find

- The focus is on the processing – the algorithm needed to perform the desired computation
  - Imperative programming with procedure calls
  - The steps, actions, and functions; data is subordinate to functions
- Languages support this paradigm by providing facilities for passing arguments to functions and returning values from functions
- Functions are used to create order in a maze of algorithms
- Algorithms are written using function calls and other language facilities

Procedure programming : example

- A typical example of “good style” is a square-root function
  - Given a double-precision floating-point argument, the square-root of that argument is calculated and returned

```
double sqrt (double arg)
{
    // code for calculating a square root
}

void f()
{
    double root2 = sqrt(2);
    // ...
}
```

Modular programming

Decide which modules you want; partition the program so that data is hidden within modules

- A module is a set of related procedures with the data they manipulate
- Emphasis in the design of programs has shifted from the design of procedures towards the organization of data
  - The techniques for designing "good procedures" are now applied for each procedure in a module
- While there are ways to use modules to provide a *form* of user-defined types, the resultant "fake types" don't behave like built-in types

Modular programming : stack example

- The most common example of a module is the definition of a stack
  - The main problems that have to be solved are:
    1. Provide a user interface for the stack (e.g., functions `push()` and `pop()`)
    2. Ensure that the representation of the stack (e.g., the array of elements) can only be accessed through this user interface
    3. Ensure that the stack is initialized before its first use
- C++ provides a mechanism for grouping related data, functions, etc., into separate namespaces; we can use namespaces to construct our stack module

Namespaces

- A namespace is just a named scope
  - The usual scope rules hold for a namespace, so if a name is previously declared in the namespace or in an enclosing scope, it can be used
- Members of a namespace are introduced using this notation:

```
namespace namespace_name {
    // declaration and definitions
}
```

- A member can be declared within a namespace definition and defined later using its qualified name `namespace_name::member_name`
- A member of a namespace can be accessed using its qualified name `namespace_name::member_name`

Namespaces

- When a name is frequently used outside its namespace, we may not want to have to repeatedly qualify it with its namespace name
- using-declaration introduces a local synonym
  - using std::cout; // use std's cout
  - using std::cin; // use std's cin
- using-directive makes all names available almost as if they had been declared outside their namespace
  - using namespace std;
    - // can use any name in std (such as cout) without having to qualify it

Modular programming : stack example

- The most common example of a module is the definition of a stack
  - The main problems that have to be solved are:
    1. Provide a user interface for the stack (e.g., functions `push()` and `pop()`)
    2. Ensure that the representation of the stack (e.g., the array of elements) can only be accessed through this user interface
    3. Ensure that the stack is initialized before its first use
- Let's construct this module with a namespace
  - We will separate out the interface and implementation details for our stack
    - The interface of the stack will be contained in a header file, which will be included in the source files wishing to use our stack
    - The definition of the stack will be provided in a separately-compiled cpp file

Modular programming : stack example

Stack.h

```
#ifndef STACK_H
#define STACK_H

namespace Stack { // declarations
that specify the interface to the
Stack module
    void push(char);
    char pop();
    class Overflow {};
    class Underflow {};
}

#endif
```

Stack.cpp

```
#include "Stack.h"

namespace Stack { // representation
    const int max_size = 200;
    char v[max_size];
    int top = 0;
}
void Stack::push(char c) {
    if (top == max_size) throw Overflow();
    v[top] = c;
    top += 1;
}
char Stack::pop() {
   if (top < 0) throw Underflow();
   top -= 1;
   return v[top];
}
```

Modular programming : stack example

- The key point about our stack module is that the user code is insulated from the data representation of our stack by the code implementing `Stack::push()` and `Stack::pop()`
  - Through the module's interface, the user will have the information on *what the object does* instead of *how it does it*
  - The data members that maintain the stack along with the member functions that operate on it are placed in an implementation file
    - Such files can be compiled separately and distributed to users as object (.o) files instead of source files (.cpp)
    - Information hiding: by making our data, the names of functions, types, etc. local to a module provides a means of sorts to "hide" its implementation details

Modular programming

- The modules in the form of our stack are not sufficient to express complex systems clearly
  - There are ways to use modules to provide a *form* of user-defined types, but these "fake types" don't behave like built-in types
  - Furthermore, what if we wanted many stacks, rather than a single one provided by our stack module?

User-defined types

Decide which types you want; provide a full set of operations for each type

- User-defined types follow nearly the same rules for naming, scope, allocation, lifetime, etc., as does a built-in type such as `int` or `char`
  - User-defined types address the shortcomings of pseudo-types that can be created through modular programming
- A user-defined type is also known as an abstract data type (ADT)
  - Stroustrup prefers the term user-defined type as a reasonable definition of an abstract data type would require a mathematical "abstract" specification
    - For the moment, let's not worry about this notion of a mathematical "abstract" specification though

User-defined types : stack example

Stack.h

```
#ifndef STACK_H
#define STACK_H

class Stack {
public:
    class Underflow{};
    class Overflow{};
    void push(char c);
    char pop();
private:
    static constexpr int max_size = 200;
    char v[max_size];
    int top = 0;
};

#endif
```

Stack.cpp

```
#include "Stack.h"

void Stack::push(char c) {
    if (top == max_size) throw Overflow();
    v[top] = c;
    top += 1;
}

char Stack::pop() {
   if (top < 0) throw Underflow();
   top -= 1;
   return v[top];
}
```

Object-oriented programming

Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance

- Classes can be designed specifically as building blocks for other types, and existing classes can be examined to see if they exhibit similarities that can be exploited in a common base class
- The amount of commonality between types that can be exploited by using inheritance and virtual functions is the litmus test of the applicability of object-oriented programming to a problem
  - Where there is no such commonality, data abstraction suffices
- The main focus is on message passing between objects
  - Objects respond to messages by performing some behavior
  - Important to note: each object has its own internal state

Event-driven programming

- The program is a continuous loop which responds to events as they are generated
  - Asynchronous interaction between initiators and listeners of actions
- The event framework controls the flow of the program

Generic programming

Write code that works with a variety of types presented as arguments, as long as those argument types meet specific syntactic and semantic requirements

- We can parameterize our functions such that the resulting function template (parameterized function) can work for a variety of suitable types and data structures

- We can also parameterize classes, resulting in class templates (parameterized types) such that they can work with a variety of types