Structuring the Program

1. Objectives

- How to structure large programs?

- How to create large applications from small program units?

- So far we have covered requirements for simple program units (Dealing with programming in the small).

- What are the requirements for developing large applications (programming in the large)?
  - Mechanisms for multiple logical units
  - Multiple programmers, and
  - Translating design into implementation. In many cases, the approaches that are used for small programs do not scale up to solve large problems

- We can use **abstraction** and **modularity** to assist us:

  - **Abstraction**
    - Allows us to concentrate on the major, important aspects of the problem, and leave the details for later
    - During problem analysis, we discover and invent abstractions that allow us to understand the problem
  - **Modularity**
    - Allows us to build programming pieces that can be assembled to construct larger programs

- During the design phase, we try to discover a modular program structure

2. Software Design Methods

- Find the appropriate modular decomposition of the desired system.

- Complexity requires us to systematically design small units called modules to construct large programs.

- Procedure and Functions are examples of Modules

- Modules:
  - Should interact in well-defined and controlled ways
  - Can be designed, understood, and validated independently
  - We need a way to describe the modules, and their interfaces
  - Modules should be as independent as possible

- Good Modularity can be achieved using information hiding:

  - Each module hides a design decision as its secret, and
  - Provides methods for clients to manipulate its abstraction

- Maintainability:

- The implementation can change without affecting the clients

3. Support of Modularity

- How to support efficient modularization?

- Different languages use different techniques for building modules
  - Class: C++, Java
  - Package: Ada, Modula-2
  - Module: C, Pascal

- Module Relationship:
  - A module is a server or service provider and the caller is the client.

- Encapsulation:

  - A unit is said to encapsulate the service
  - Combine program components that are necessary to provide the service
  - Only make those aspects visible to clients that are required
  - Language mechanisms provide a means for information hiding

- A unit is clearly defined by two parts:
  - The interface describes what services are exported by the module, and imported by the client

- The implementation describes the module's internal structure that provide the service

- Example: A dictionary data structure
  - Clients may store and retrieve pairs of the form <name, id>Operations include
    - Inserting a pair <"Pat", 23>
    - Retrieving elements by supplying the string component "Pat"
    - Deleting a pair based on a string

  - Encapsulation helps to hide the implementation (array, linked list, etc.)

- Language Example:

  - Built-in Types:
    - How float type is implemented

  - Java:
    - How vector or set data types are implemented?

4. Separating Interface and Implementation

- In order to use a module, we only need its interface and not its internal implementation.

- This clean separation contributes to the independence of the module.

- C++:

- The unit of encapsulation in C++ is a class.The interfaces consist of all class member functions, types, and variable declarations that can be accessed by clients.

- Ada:
  - Treats the separation of interface and implementation more strictly
  - Unit of encapsulation is the package:
    - Package specification
    - Package body
    - Compiled separately

5. Separate and Independent Compilation

- Modules can be developed and tested independently of the whole application.
- This is independent compilationSeparate compilation means that modules can be compiled separately, but might have some ordering constraints
- Example:
  - C supports independent compilation
  - Ada supports separate compilation

- C Example:
  - compute.c
  - calc_amount.c
  - get_rate.c

- Programming Languages must provide the following:

  - Unit of compilation
  - Order of compilation

        - Amount of inter-module checking

6. Libraries of Modules

- To control the complexity of very large programs, libraries were introduced
- Common modules that are used by many other modules can be grouped into standard libraries–Avoid name clashes
- Avoid the need to recompile when an unused part of a library changes

7. Basic Definitions

- Now we look at how specific languages support programming in the large.

- **Logical modules**: Modules identified at the design stage

- **Physical modules**: Implementation modules
    - A logical module may be implemented by one or more physical modules

- Each language will be discussed based on the following points:
        - **Module encapsulation**: What is the unit of modularity and encapsulation?
        - **Separation of interface from implementation**:
            - How does a client module gain access? What entities are exported/imported?

- **Program organization and module grouping**:
  - How independently can modules be compiled?
  - What are the visibility and access control mechanisms?

8. C Language

- Decomposition:
  - Functions are the unit of decomposition
  - Unlike Pascal, functions may not define nested functions

- Encapsulation
  - The file is the unit of encapsulation
  - Can access variables and functions from other files by declaring them as **extern.**
  - Variables and functions can be protected from export using the **static** key word.
  - Variables declared within functions have local scope

- Interface and implementation
  - The unit of physical modularity is the file
  - No language support for separating an interface from an implementation
  - ".h"-files are headers or include files (interface)
  - ".c"-files are code (implementation)
  - A client wishing to use a module includes its header
    - –**#include <myADT.h>**
  - Header files can contain constants, type definitions, variables, and functions

- Only the signature of a function (its **prototype**) is given in the header
- The implementation is given in the code file
- Header files allow the compiler to compile modules independent of the implementation

- General program structure

  - There is no nesting

  - Structure:

```
#include …various files…
global variable declarations
functions declarations
int main( int argc, char *argv )  {
  …Only one main function is allowed
}
```

9. C++ Language

- C++ is a proper superset of C

- The file is the unit of physical modularity

- It provides **classes** in support of programming in the large

  - Provide encapsulation and interface control
  - Classes may be nested

- It provides **templates** in support of generic programming

- Encapsulation

- The class is the unit of logical modularity
- The class is used:
    - To define user-defined types
    - To group entities together

- Entities are either
    - **Public:** exported to clients
    - **Private:** hidden from other applications.

- Grouping of units
    - Friend
    - Namespaces

- ***"friend"*** classes
    - There is no "friend" relation in Java (Java uses Default access modifier or Also know as "Package Visibility")
    - A *friend* class is one that has access to the private and protected elements (data and methods) of another class.

    - Example: www.cplusplus.com

```
// friend functions
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area (void) {return (width * height);}
    friend CRectangle duplicate (CRectangle);
};

void CRectangle::set_values (int a, int b) {
```

```
  width = a;
  height = b;
}
// Note that "duplicate" function is not a member of CRectangle
CRectangle duplicate (CRectangle rectparam)
{
  CRectangle rectres;
  rectres.width = rectparam.width*2;
  rectres.height = rectparam.height*2;
  return (rectres);
}

int main () {
  CRectangle rect, rectb;
  rect.set_values (2,3);
  rectb = duplicate (rect);
  cout << rectb.area();
}
```

- Another Example: www.cplusplus.com

```
// friend class
#include <iostream.h>

class CSquare;

class CRectangle {
    int width, height;
  public:
    int area (void)
      {return (width * height);}
    void convert (CSquare a);
};

class CSquare {
  private:
    int side;
  public:
    void set_side (int a)
      {side=a;}
```

```
    friend class CRectangle;  // So CRectangle can access the protected
                              // and private members if CSquare.
};

void CRectangle::convert (CSquare a) {
  width = a.side;        //Note that convert method of CRectangle
  height = a.side;       //uses the private variable side of CSquare

}

int main () {
  CSquare sqr;
  CRectangle rect;
  sqr.set_side(4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

- Namespaces
  - Namespaces allow us to group a set of global classes, objects and/or functions under a name.
  - How to refer to a class that has the same name in two different libraries?
  - Namespaces allow the user to uniquely specify an element:

```
namespace MyLibrary {
  class MyClass …
};

MyLibrary::MyClass.myMethod( );
```

  - Example: www.cplusplus.com

```
// using namespace example
#include <iostream.h>

namespace first
```

```
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << var << endl;
  }
  {
    using namespace second;
    cout << var << endl;
  }
  return 0;
}
```

10. Ada Language

- Originally designed to support programming in the large
    - Modules
    - Encapsulation
    - Interfaces

- Encapsulation:
    - The package is Ada's unit of modularity
    - Explicit distinction between specification and implementation
        - Module specs are developed first
        - Implementation may be done independently

- Interface and Implementation:
    - The interface of an Ada unit consists of the "**with**" statement.
    - The "**with"** statement lists all modules from which entities are imported
    - The ***unit specification*** is enclosed within a **is...end** pair, and lists the entities which are exported from the unit.
    - Example:

```
With X;
Package T is
        C: integer;
        Procedure D(…);
End T;
Package body T is
        …
end T;
with T;
procedure U(..) is
```

```
        …
        T.D   -- access procedure D in package T
        T.C   -- access variable C in package T
        …
end U;
```

Another way to access imported entities:

```
with T;
use T;
procedure U(..) is
        …
        D     -- No need for T: access procedure D in
package T
        C     -- No need for T: access variable C in
package T
        …
end U;
```

- Program organization:
  - An Ada program is a linear collection of modules:
    - sub-programs
    - packages
  - Module declarations may be nested
    - Can make child packages
    - Can represent the program structure as a tree
  - The **separate** keyword allows for separate compilation (p. 264)
  - Example:

```
Procedure X (..) is
        W: integer;
```

```
        Package Y is
            A: integer;
            Function B (C:integer) return
    Integer;
        End Y;
        Package body Y is separate;
    Begin

    End;
```

- Another File:

```
Separate(X)              -- Sub-unit body
Package body Y is
    Procedure Z (..) is separate;
    Function B (C: integer) return Integer is
    Begin
        …
    End B;
End Y;
```

- Another file:

```
Separate(X,Y)
Procedure Z (…) is
Begin
    …
end Z;
```

- Separate compilation
    - A unit can be compiled only if all units mentioned in its "with" statement have been compiled previously.
    - A sub-unit can be compiled only if the enclosing unit has been compiled previously.

- Ada Private Type:
    - Similar to **private** members/methods in C++
    - Private things are secrets of the package and therefore are invisible to clients (p. 268)
    - Allows the creation of multiple instances of an object.

- Grouping in Ada
    - If there are many modules we need to:
        - reduce naming conflicts
        - Reduce the amount of recompilation when a unit changes
    - ***Child library units*** solve these problems:
        - Hierarchical naming scheme
        - Nesting at the package level