Ideas in the Structured Programming Paradigm and Some Implementation Aspects related to it

A CS 152 Lecture

PROF. R. K. JOSHI

Dept of Computer Science and Engineering

Indian Institute of Technology Bombay

What's Structured Programming?

- Modular Structural Decomposition in a top-down fashion
- Different concerns of the program are implemented as different subsections/modules which are structured in a top down fashion.
- Fixed style, clarity, productivity, ease of testing & maintenance and ease of redesign
- At the heart of structured programming is the idea of using only *Single entry and single exit* blocks

Top Down Decomposition

- Design a program as a top-down hierarchy of modules.
- This hierarchy is developed according to various design rules and guidelines
- The modules are evaluated as per the quality acceptance criteria to ensure the best modular design for the program.
- The modules are implemented using structured programming principles.

Modular Design → Modular Packaging

- Modular design
  - Group of executable instructions with a single point of entry and a single point of exit.

- Packaging
  - Assembly of data, processes, interfaces and machines

Dijkstra’s contributions to structured Programming

Single entry single exit blocks

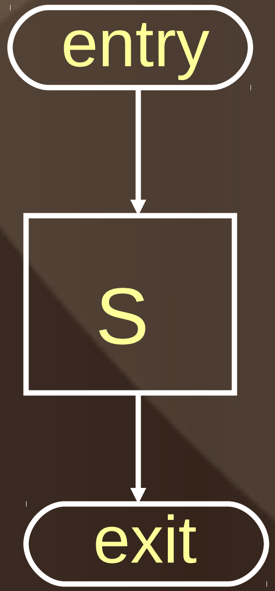
Goodbye to GOTO?

Single Entry Single Exit Structures

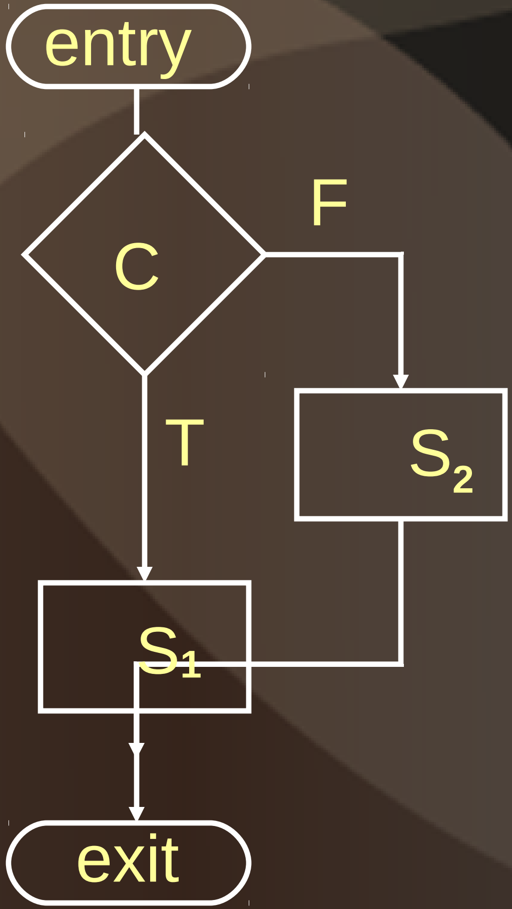
- A primitive statement
  - **S**
- Sequence of statements
  - **S1;S2;S3;….**
- If-then-else (Conditional)
  - **if C then S1 else S2**
- Conditional Repetition
  - **while C do S**
- Case statement
- Top down Function calls

Flow Graph Representations

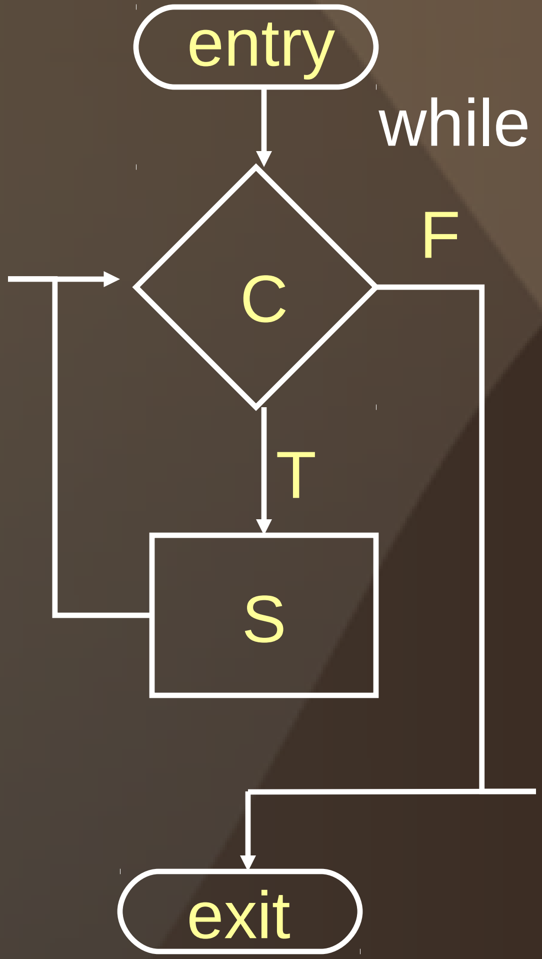
A Primitive Statement : S



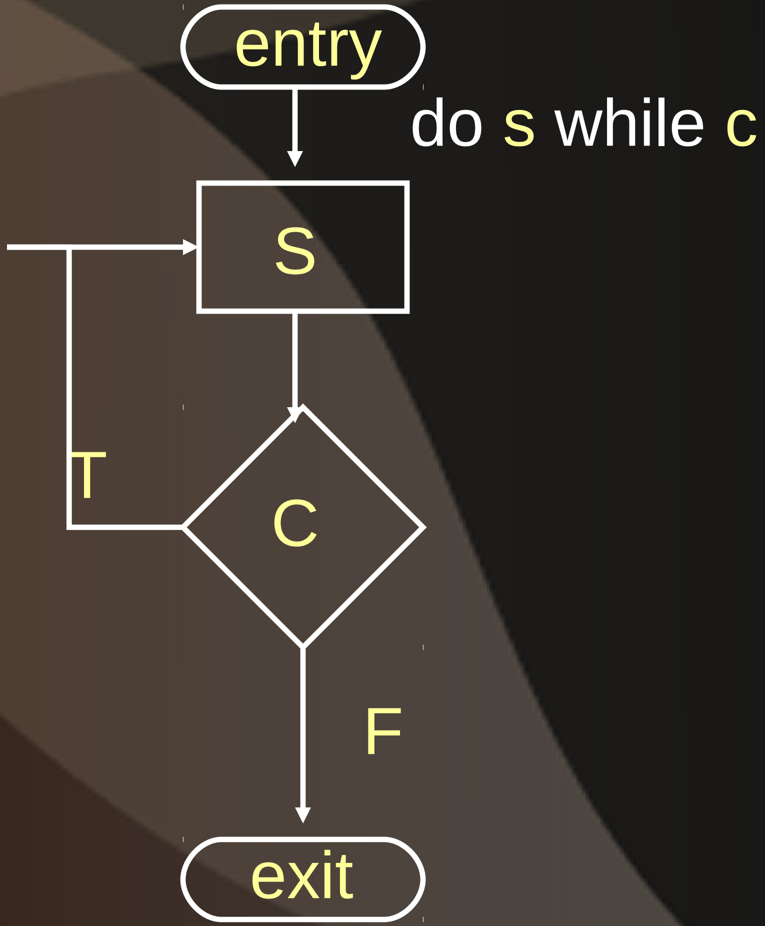
A Conditional Statement : if C then S1 else S2



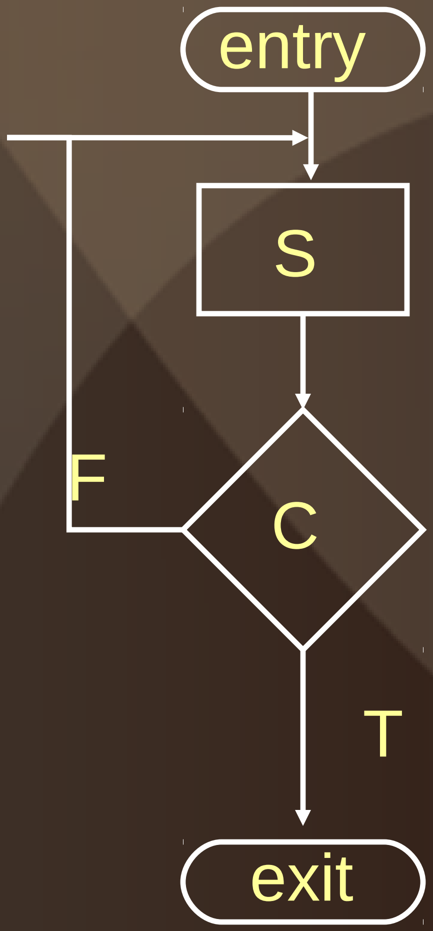
A Conditional while-do



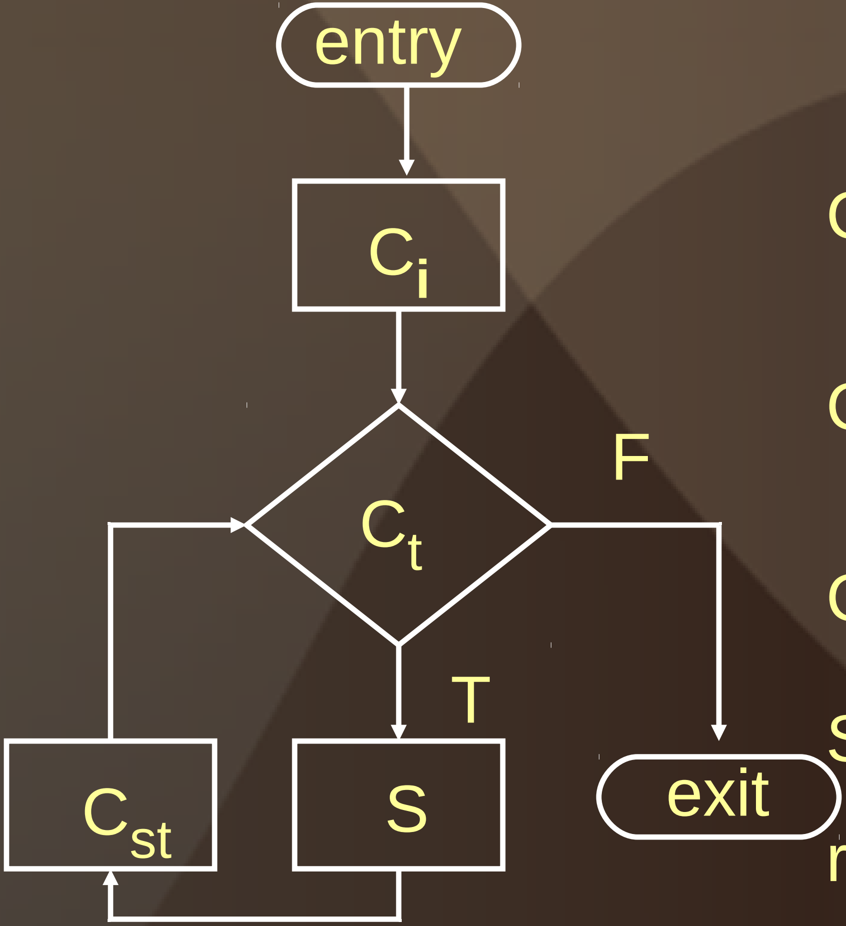
Repetition do while



Repeat S until C



Flow graph of `For` Statement of C



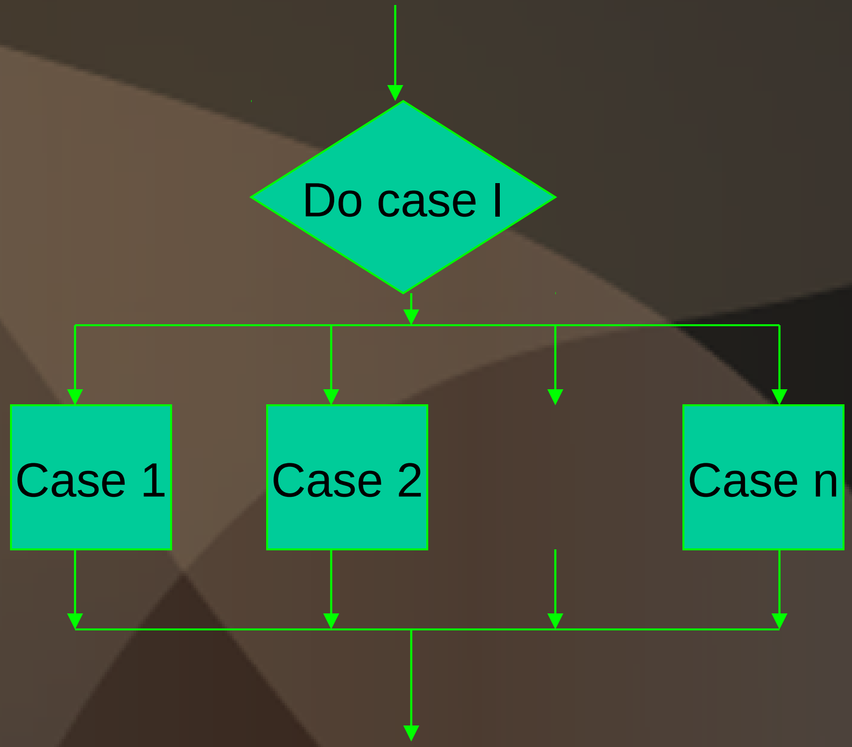
For (C_i; C_t; C_{st}) S;

C_i: initialization

C_t: Termination condition

C_{st}: Step

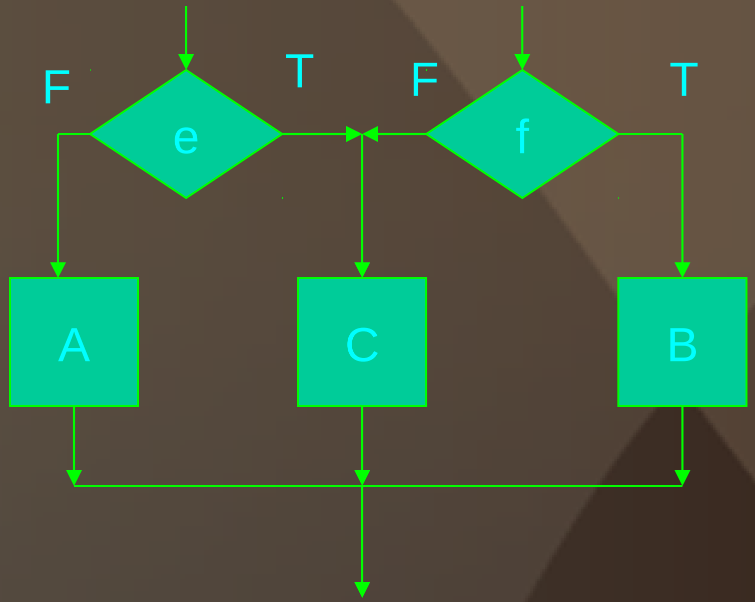
S: Statement to be repeated



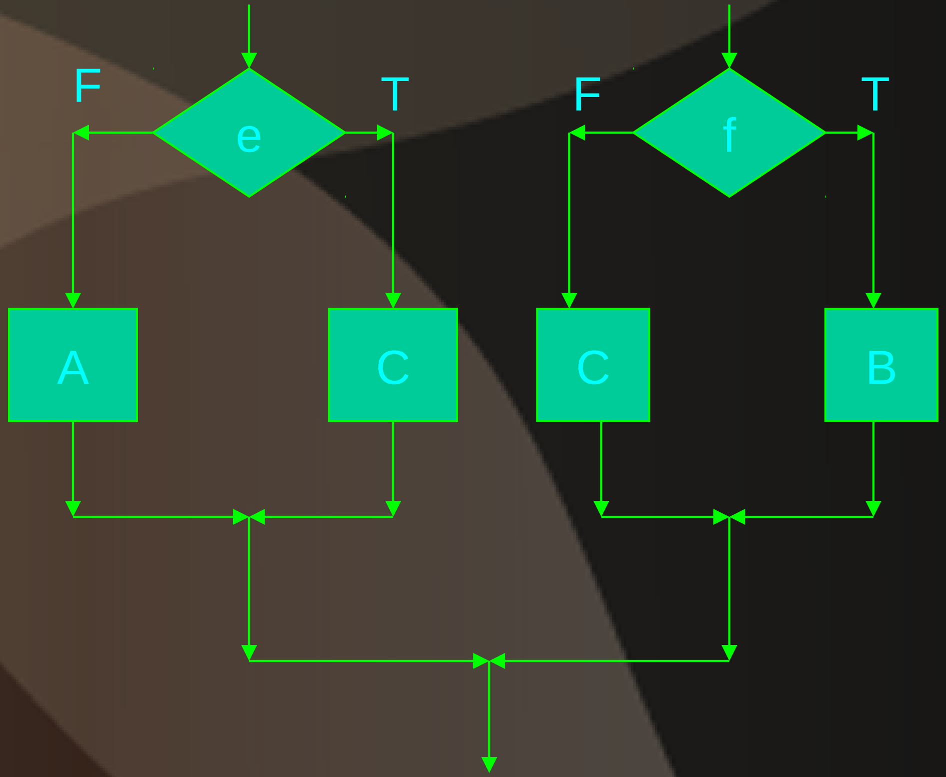
<u>Do case I</u>

```
    case 1
    case 2
    --------
    case n
```

Non Structured



Structured Equivalent



“Goto considered harmful”

- Goto produces unstructured programs
- Multiple entries into code & multiple exits are possible
- Programs become difficult to understand & debug.
- Very low level programs use goto (machine language), but goto is undesirable for high level programming.

An example with GOTO

```
main ( ) {
        int marks=20;
        if (marks > 80) goto label1;
        if (marks > 60) goto label2;
        if (marks > 40) goto label3;
        printtf(“F”); goto Last;
        label1: printf(“A”); goto Last;
        label2: printf(“B”); goto Last;
        label3: printf(“C”); goto Last;
        Last: printf (“\n”);
}
```

Removing the GOTO statements

```
main ( ) {
        int marks=20;
        if (marks > 80) printf (“A”);
                else if (marks > 60) printf (“B”);
                        else if (marks > 40) printf (“C”);
                                else printf (“F”);
        printf (“\n”);
}
```

Multiple Exit Forms in Modern Programming

- Often found to be convenient.
- Used where single exit forms may become cumbersome.
- Exceptions
- Return from a function.
- Labeled breaks & continue.

Exceptions

```
T C::function f (any s ...) {
        if  (..undefined..)
                •throw MyException();
        ...... member function logic....;
        return a value of type T;
}
```

- The above member function uses exceptions at entry level to detect violation of a precondition.
- The caller needs to handle an exception that is thrown by the call.

Multiple Returns

```
headnode (List l) {
        if (l== null)
                return (null);
        else
                return (l → head)
}
```

The above function uses the return statement twice. Thus you have 2 exit statements to return from the function.

it’s single exit code form is given below.

```
headnode (list l) {
          item * h;
          if (l== null)
                    h=null;
          else  h= l → head;
          return (h)
}
```

But the multiple return form avoids temporary variables, and it is also perceived to be convenient and readable.

Some other modern Multiple Exit forms

- Unconditional Break
  - Break into outer loop
- Unconditional Continue
  - Continue with next iteration of the current loop
- Labeled Break
  - Break the outer enclosing loop
- Labeled Continue
  - Continue the next iteration of the outer loop

Break and Labeled Break

- Break - exits from a block
  - e.g. Exit from switch, for, while, do blocks
  - example: for (...) { ...; ... break; ....}
  - Unlabled break terminates the innermost block statement
- To break out of an outer statement, use labeled break

```
alabel : ...
for (i=....) {
        for (j = ... ) { break alabel;}  }
```

- break is not the same as GOTO statement!

Continue and Labeled Continue

Continue : Skips to the end of current loop's body (while/do/for)

loop termination is evaluated
loop may continue with next iteration

```
for (...) { if (..) continue; ...}
```

Labeled Continue: Skips the current iteration of outer loop

```
alabel : for (i=....)
for (j=....) {.... ; continue alabel; ....}
```

Unlabeled Breaks

```
public class unlabeled_break {
        public static void main(String[] args) {
                int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076, 2000, 8, 622, 127 };
                int searchfor = 12;
                int i ;
                boolean foundIt = false;

                for ( i = 0 ; i < arrayOfInts.length; i++) {
                        if (arrayOfInts[i] == searchfor) {
                                foundIt = true; break;
                        }
                }

                if (foundIt)
                        System.out.println("Found " + searchfor + " at index " + i);
                 else System.out.println(searchfor + "not in the array");

        }
}
```

Labeled Continue

```
public class labeled_continue {
        public static void main(String[] args) {
                String s = "search substring with or within this";
                boolean found = false;
                String sub = "within";
                int len1=s.length();
                int len2=sub.length();
                int max= len1 - len2;
                int i=0,j=0,k=0,pos=0;

                outer:
                for (i=0; i<=max; i++) {
                        j=i;
                        for (k=0;k<len2;k++,j++) {
                                if (s.charAt(j)!=sub.charAt(k)) continue outer;
                        }
                        found = true;
                        pos=j-len2;
                        break outer;
                }
```

```
            if (found) {
                    System.out.println("original string:" + s);
                    System.out.print("substring found at position:" + pos + ":");
                    for (i=pos;i<j; i++) System.out.print(s.charAt(i));
                    System.out.println("");
            }
        }
    }
```

Some Modularity Units in paradigms: discover them by looking into visibility rules!

- Procedural/Imperative
  - Structures, procedures, files, functions
- Functional
  - Functions, higher order functions, modules
- OO
  - Objects, classes, packages, namespaces, files
- Declarative
  - Facts, rules, modules, files

Principles of Modular Software Construction

A primary concept in modular programming is the interface of a component---the manner in which the component interacts with its users.

The most common kind of interface in software construction is the procedure call. Execution of a procedure call supplies a module with a set of input values and requests that the module use the values in constructing result values made available to the caller.

To attain the full benefits of modular

programming the support

provided by the computer system in

support of the component

interface should meet the following

requirements:

Information Hiding Principle

Information Hiding

Context Independence

The user of a module must not need to know anything about the internal mechanism of the module to make effective use of it.

Invariant Behavior Principle

The functional behavior of a module must be independent of the site or context from which it is invoked.

--> Reusability out of Context

Data Generality Principle

The interface to a module must be capable of passing any data object an application may require.

Data abstractions

Secure Arguments Principle

The interface to a module must not allow side-effects on arguments supplied to the interface.

The “Pure Function” like paradigm, but at module level--

Recursive Construction Principle

A program constructed from modules must be useable as a component in building larger programs or modules.

Use a compiler to build a language, and use the language to build another possibly better compiler !

System Resource Management Principle

Storage management for data objects must be performed by the computer system and not by individual program modules.

Separate Concerns! Don't “fiddle” with things for which 'you' are not responsible..

Module Coupling

- Content coupling
  - Dependent on internals of another
- Common coupling
  - Shring via globals
- Stamp coupling
  - Partial sharing of composite data
- Data coupling
  - Data shared through in/out parameters
- Message coupling
  - Communication via message passing
- Subclass upward coupling
- Superclass downward coupling
- Synchronization coupling

Module Strength (Cohesiveness)

- Coincidental strength
  - Coincidental togetherness
- Logical strength
  - Similar things at one place (e.g. all outputs)
- Classical strength
  - All operations are related in time sequence (e.g. initialization seq,)
- Procedural strength
  - Operations performed make up/contribute to a procedure
- Communicational strength
  - Member functions communicate through shared state variables
- Informational strength
  - Many functionally cohesive modules are together since they operate on common DB
- Functional strength
  - All functions in a module contribute to a single functionality

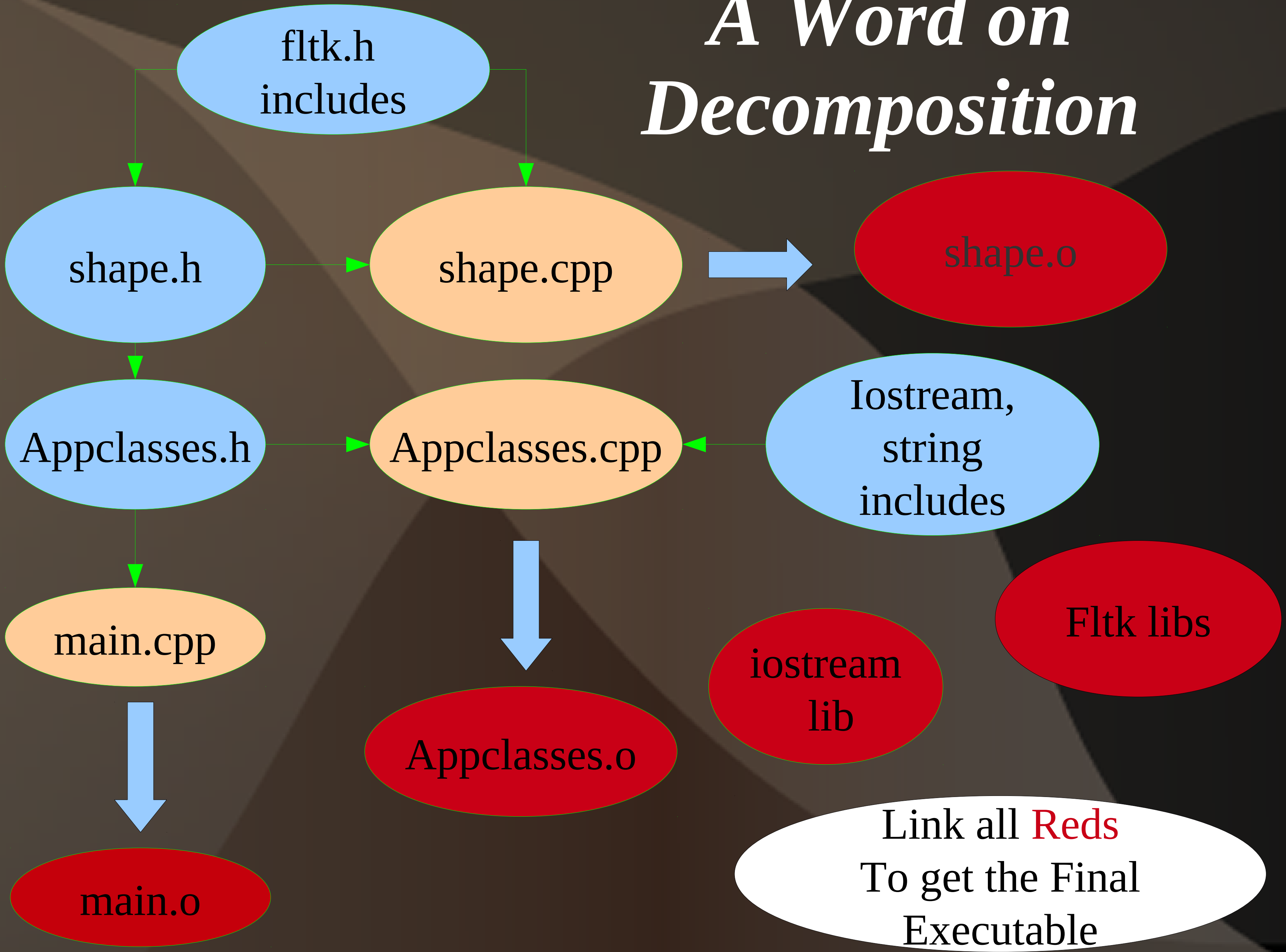
Advantages of Modular programming

- Easy to change
  - Open to change the internals, rewiring of components
- Easy to write and debug
  - Separate development of modules
- Easy to manage
  - Different people may handle different independent or loosely coupled modules after agreeing with the interfaces
- Top down
  - Architecture and Design first, i.e. “Model Driven Development”
  - There is also a paradigm called “Test Driven Development”
- Reliable
  - Tested components, reliable interfaces, keep using!

Disadvantages of Modular programming

- Time constrains
  - Forward method takes time since design has to be ready before implementation
- More care needed
  - Any mistake upstream is costly
- Reluctant programmers
  - Programmers want quick results (but they may spend more time in debugging!)
- More memory required
  - For good planning through separation of cocerns

A Word on Decomposition



A Makefile specifies the above diagram

```
target: dependent1 dependent2
        Command to generate or achieve the target
dependent1: dependent3 dependent 4
        Command to generate dependent1

... and so on for all dependents
```

- If a dependent is not specified as a target, it should be available directly in the folder as a file
- If any dependent has a timestamp later than a target, the target has to be made again
- The command-line program `make' finds it out and executes the commands specified to make the targets which should be redone

A Makefile

```
myapp: main.o appclasses.o shapelib.o
	g++ -o myapp main.o appclasses.o shapelib.o -I/usr/local/include
-I/usr/include/freetype2 -D_LARGEFILE_SOURCE
-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64 -D_THREAD_SAFE
-D_REENTRANT  /usr/local/lib/libfltk.a -lXext -lXft -lfontconfig -lXinerama
-lpthread -ldl -lm -lX11
main.o: main.cpp main.h
	g++ -c main.cpp
appclasses.o: appclasses.cpp appclasses.h
	g++ -c appclasses.cpp
shapelib.o: shapelib.cpp shapelib.h
	g++ -c shapelib.cpp
clean:
	rm main.o appclasses.o shapelib.o
```