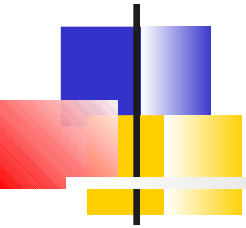
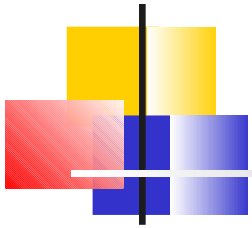


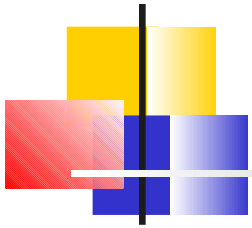
Program Design



Objectives

We will discuss, in general terms, how to

- Design and implement small, but nontrivial, programs in C.
- Understand and follow a systematic method for design and implementation of C programs.



Program Design

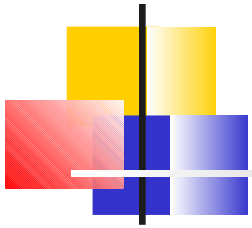
How to write the program for a big, complex job:

Divide it into small simple parts.

In C this means *functions*.

No function should be very long or very complex.

> *If you cannot look at a function on your computer screen and understand what it is doing, it is too long, too complex, or both.*



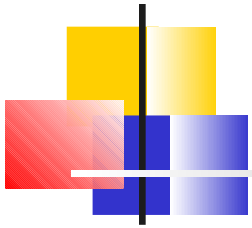
How to design a program.

Understand the requirements.

- Have a clear statement of *what* the program is supposed to accomplish.
  - Without regard to ***how*** it will be done.

Develop a test plan.

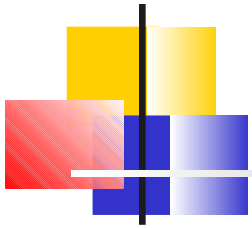
- Inputs and expected outputs.
- How will you know when you are finished?
- Should show that the program meets its requirements.



How to design a program.

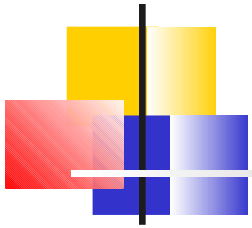
Determine *how* to produce the required results.

- You can't write a program to do something that you don't know how to do yourself.
- Ask if you already have a program that does this or something similar.
  - Never invent when you can copy! (except for exams and projects!)
- Look for the simplest solution that will meet the requirements.



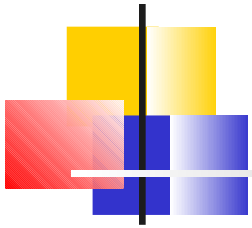
How to design a program.

- Develop a plan for the program.
  - Understand the problem and how the program will produce the required results.
  - Sketch out what the program will do at each stage of computation.
  - Sketch out how you will represent data in the program.



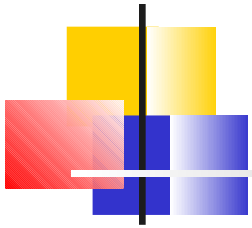
How to design a program.

- If the program gets input data, write the code to do the input.
  - If multiple inputs are required, write a loop to get the input and test for completion.
- Write the code to do the output.
- Write a ***stub*** for the function to do the work required of the program.
  - Don’t add complexity to the work function by requiring it to do input and output, or anything else that can be removed.



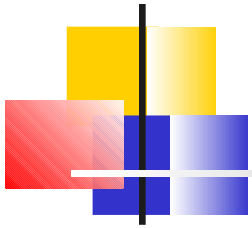
Writing the “work” function

- If it is clear how to perform the required function, write the “work” function.
  - If it requires more than one screenfull of code, divide it into parts.
    - Provide a function with a meaningful name for each part.
    - Call the “part” functions from the work function.
  - Repeat as necessary.



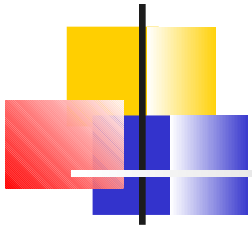
Writing the “work” function

- If it is not clear how to perform the required function, study the problem.
  - Look for published solutions.
    - Google knows *everything*.
    - Books on algorithms.
    - Books on subject of the program.
    - Articles.
  - Look for ways to break the problem into smaller, simpler pieces.
    - No silver bullet.
    - You have to understand the problem.
    - “Divide and Conquer”



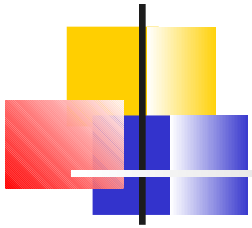
Divide and Conquer

- Look for ways to chip off a piece of the problem.
  - If I can’t solve this problem, can I solve a part of it?
  - Is there a related but simpler problem that I can solve?
  - Even if you can’t see the end, try to move forward.



How to implement your design.

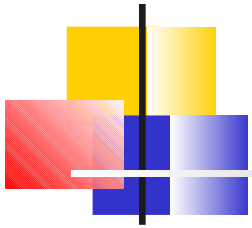
- Always have a working program.
- Start with just greeting and termination messages.
  - Add functionality in tiny steps.
  - Output intermediate results during development.
  - Compile and test after each step.
- Use stub functions as placeholders
  - Example: Just output a message saying the the function was called and showing agrument values.
  - Return a constant of the right type.



How to implement your design.

- Put in debugging messages to tell you what is happening.
  - Trace messages:
    - Function xxx starting
    - Function xxx complete
  - Show values of variables
  - Always follow trace message output with

    `fflush(stdout);`
- Put in assert statements to verify assumptions about values of variables.



Simple Black Box Testing

- Context: a program that only uses terminal I/O
  - That is, all input is via scanf and all output via printf

- Steps:
  - manually create a file **`input`** containing the input values expected by a single execution of the program
  - manually create a file **`correct`** that contains the correct output
  - Assuming your program was compiled to executable file **`main`**, enter the following command:

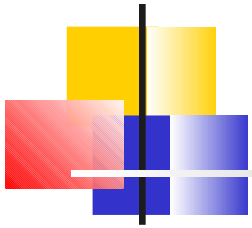
```
./main < input 1> output 2>&1
// Both regular output and error messages
   // will be sent to file named output
```

  - Then, either bring up the files **`correct`** and **`output`** in an editor and visually compare them for differences; or enter the command

`fc /L/N output correct` (Windows)

`diff output correct` (Linux/OS X)

- This will display the differences between the two files.



How to implement your design.

- Continue adding functionality in tiny steps until program is complete.
- **Test thoroughly**.
  - Re-examine test plan.
  - Need more test cases?
  - Are there critical cases that were not obvious in advance?
  - Are all code paths tested?
- Remove debugging messages.