Software Development 2: Incremental Devel.

1. Introduction
2. **Incremental Software Development**
   - **incremental development**
   - **top-down vs. bottom-up development**
   - **stubs and drivers**
3. Incremental Development Example
4. Software Bugs
5. Testing
6. Debugging

Incremental Software Development

Perhaps the most important piece of advice for being successful developing complex programs is:
develop the program incrementally, i.e., in stages!

Program bugs can interact in complex ways, so the difficulty of debugging code goes up *rapidly* as the number of bugs increases.

Debugging a program with twice as many bugs as another will typically be vastly more than twice as hard.

Working *incrementally* means adding code for a small amount of new functionality, then testing and debugging your program, before moving on to add code for additional functionality.

Incremental Software Development (contd.)

Not only does this approach limit the *number* of bugs in your developing program, it helps limit *where* they might be (most likely in the new code).

Many students mistakenly believe that working incrementally will be too time consuming.

Instead, they type in the entire program, and only then begin trying to get it to compile and work properly.

Unless you are an experienced programmer working with techniques you are familiar with, this approach is highly likely to lead to failure and frustration!

Please don't work that way—particularly when working with an unfamiliar language and/or new programming techniques.

Top-Down vs. Bottom-Up

There are two standard approaches for incremental development:

- **top-down development**
- **bottom-up development**

In *top-down development*, we start with an *abstract* version of our overall logic, then successively refine it until all of the logic has been implemented.

This generally involves the successive addition of *functions/methods* at *"lower levels"* of abstraction.

Lower abstraction levels means functions that come closer to doing what ultimately needs to be done to complete functionality, making less use of use of other functions we have written.

At the lowest level, we will have functions that make use of only system functionality (language operators and library calls).

Top-Down vs. Bottom-Up (contd.)

In *bottom-up development*, we start by implementing the lowest (abstraction) level functions first, then add successively higher-level functions that make use of these functions.

Only at the very last step will we have code that carries out (even abstractly) the entire functionality the system is supposed to have.

In general, either a top-down or bottom-up approach can be used with any project, though sometimes one may make more sense.

The choice of programming language can also be a factor in your choice.

Top-Down vs. Bottom-Up (contd.)

Languages like Lisp and Python have **interactive environments**, where you can run individual lines of code.

Interactive environments help support *bottom-up development*.

By contrast, compiled-only languages like C/C++/Java can require significantly more effort to develop bottom-up.

The reason for this is that with bottom-up development you will have many functions that must be tested separately (since you lack an overall abstract program).

Interactive environments make this relatively simple: just compile each function when completed and then run it to test it.

Without an interactive environment, **driver** programs will be required for bottom-up development.

Stubs and Drivers

Stubs and **drivers** are concepts that arise in the context of top-down or bottom-up incremental program development.

A **stub** is code that substitutes for the code necessary to provide some functionality, prior to that code being written.

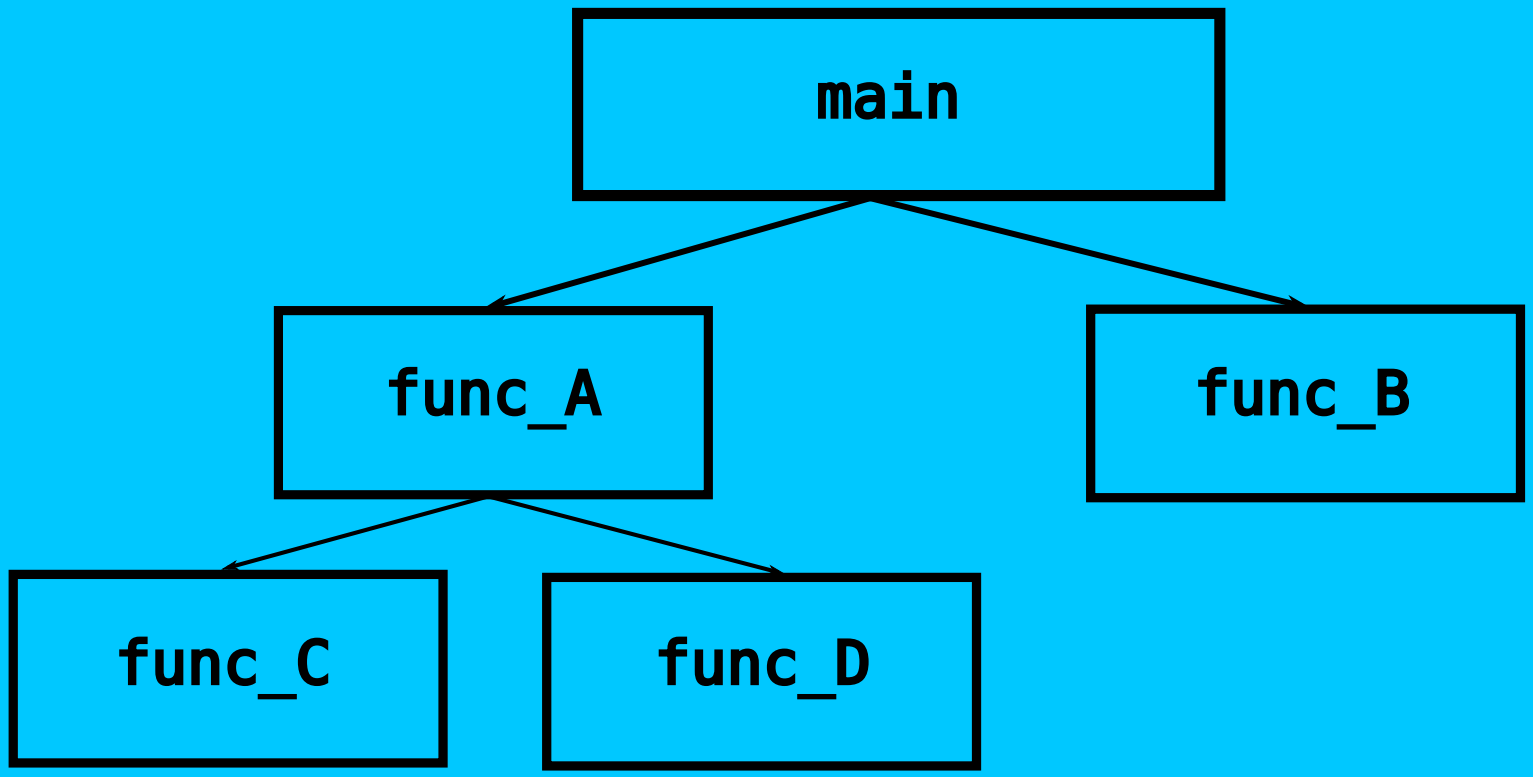
In *top-down development*, having an *abstract* version of the program basically means that the program calls *stub* functions prior to the real functions being implemented.

A **driver** is code that sets up data necessary and then calls an implemented function/method.

In *bottom-up development*, *drivers* allow components to be tested prior to the program as a whole being completed to the point where it can call the components.

Stubs and Drivers Illustrated

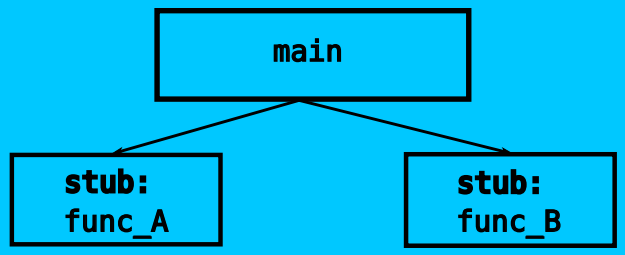
Suppose the function call design for a program is as follows:



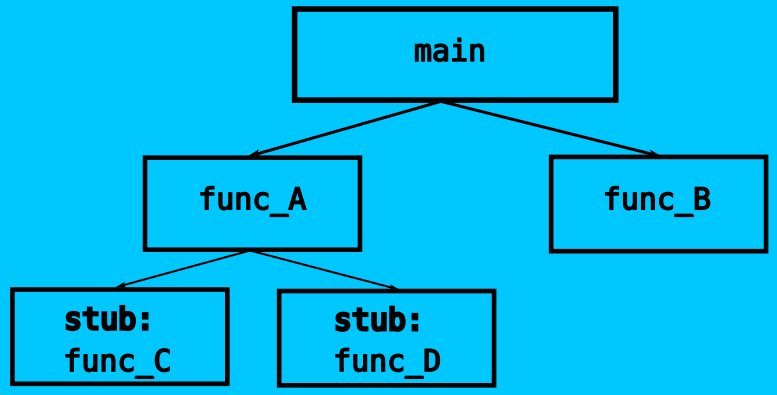
(Vectors indicate function calls.)

Stubs and Drivers Illustrated (contd.)

Top-down development would start as so:



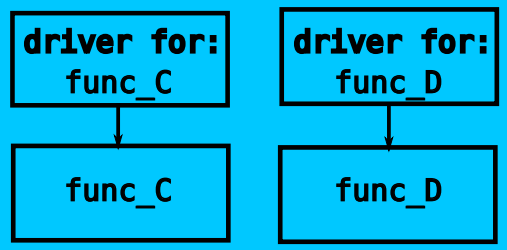
Proceed as follows:



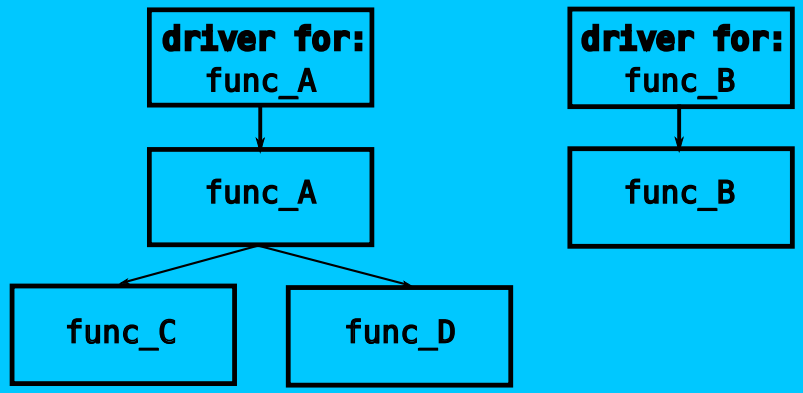
And finally end up with the complete design.

Stubs and Drivers Illustrated (contd.)

Bottom-up development would start as so:



Proceed as follows:



And finally end up with the complete design.