The Man And The Algorithm

Robert Dondero, His Course, And the Grammer of Explanation

A Junior Paper by Allen Paltrow-Krulwich ’14,
Epistemology, Cognition, and Intelligent Systems.

0. Introduction

I took Computer Science 217 in the Fall of 2011. I have a great love for Computer Science, but unfortunately, it is unrequited. I enrolled with legitimate fears that I wouldn't be able to grasp the material, let alone compete with Computer Science majors, and might ultimately have to drop the course.

That didn't end up happening. Instead, I learned the material to a degree of mastery I have achieved in few other disciplines. I did so not through any Herculean feat of personal study. In fact, the opposite – COS 217 was one of the simplest and most straight-forward courses I have ever taken. Yes, the material was complicated, the assignments complex, the exams challenging – but I became certain that if I simply put in the time, I would inevitably end up mastering the material, and without particular exertion on my part. All I had to do, in a sense, was sit back and watch.

My experience, I believe, is not uncommon. Computer Science 217 is an enigma. It is, pound for pound not one of the most challenging Computer Science courses, and, relative to the backgrounds of the students who take it, the most difficult technical course offered at Princeton. When working afterwards in industry, I have shocked and impressed career programmers decades my senior by mentioning how some fact I learned in 217 explained a particular artifact of the language we were working with.

It is actually quite rare for a working programmer to understand the foundations of their programming systems, precisely because the material is so dense. The material of 217 is a tangled nest of convention, causality, simultaneous explanation at multiple levels of abstraction, programming skills, and best practices from industry. The degree of mastery its students achieve is remarkable, and requires an explanation.

The explanation, I believe, is one man. Robert Dondero has been the head preceptor of the course since 2001, and has designed large portions of the course materials. Dondero's precepts are widely considered by students to be the best of the course, and some of the best teaching at Princeton. A scroll through course reviews offered on a university website shows that he is more often mentioned than the professor of the course, by far. He is in fact mentioned in nearly every one, unheard of for a preceptor in any department.

A few anonymous excerpts:

> "217 is, as 126 was, a very well thought-out course. I really appreciated how lectures and especially precepts were directly pertinent to the assignments, and the assignments themselves were phenomenal.
> …
> That said, my experience could have been very different if I hadn't had the man, the myth, the legend: Bob Dondero. You can see the trend for yourself--Dondero is, as a previous reviewer said, the heart and soul of the course and he really knows what's going on. His handouts are absolutely necessary for doing each of the assignments, and he explains them in such a way that you can't possibly misunderstand.
> …

Do whatever you can to get Bob Dondero as your preceptor--it could make or break this course for you."
–Fall, 2008

"The key to understanding the course is having Dondero as your preceptor. I've never seen the other preceptor, so no clue how good/bad he is, but Dondero basically holds your hand through the material."
–Spring, 2008

"Try to get into Dondero's precept. He is the best, most thorough preceptor, and without him, the course is not really worth taking (if it wasn't required for COS majors)."
–Spring 2006

"There is only one thing to say about this class: Bob Dondero is a God. Lectures were substandard, but the material is fun and as long as you have Bob Dondero as your preceptor, you will understand everything. HE IS AMAZING! Make sure Bob Dondero is your preceptor!"
–Spring 2006

"Debugging, especially in the later assignments (assembly language, the Heap Manager assignment) is a huge pain. That being said, thank God for Bob Dondero. Besides the incredibly helpful precepts, which have been mentioned by the other reviewers, he will respond to any email within about 5 minutes at any time of day/night (within reason...I never tried after 3am or so) and he can spot just about any bug.

…

Also, if you're deciding between this and 226 but will take both, take 217 first so you learn how to program from Bob Dondero, the God of Programming.
–Spring 2005

"Note, however, that for a beginning programmer (a.k.a. me) 217 is hard. I spent much of the first half of the semester feeling out of my league and doing poorly on the assignments. Yet by the end of the course my self-confidence had grown enormously (now I'm taking 226 and it's like swinging with three bats then going to one).

The redeeming factor: precepts. Bob Dondero is the ultimate preceptor. If you don't believe me, ask anyone who has ever set foot in his classroom. His precepts are jam-packed from start to finish with everything you need to know to complete the assignments. His handouts are comprehensive, his stack traces thorough and clear, and his emails legendary. I cannot think of another preceptor, nay, another human being who performs his job with greater care and consideration. When I saw that the SCG was back online, the first thing I thought was, "I've got to write something about Bob Dondero!". What more can I say....

If you want to learn the C programming language (fast), plus some nifty things about low-level stuff like assembly language and memory management, take this class and make sure Robert Dondero Jr. is your preceptor."
–Fall 2005

"MAKE SURE THAT YOU GET ROBERT M. DONDERO AS YOUR PRECEPTOR.
I cannot reiterate this enough. Bob has won like 5 teaching awards for this course. He is inhumanly organized, amazingly receptive, and incredible at figuring out what is wrong with your code. He is also very focused on making sure that everyone in his precept understands everything he covers, and everything he covers is important for the assignments."
–Spring 2004

What gives? How can it be that such a wide array of students with such different backgrounds and learning styles are in such universal assent?

Instead of showing my appreciation by writing on online course review myself, I wrote this Junior Paper. Herein I hope to explain, in short, what Dondero is doing and what makes him exceptional – deserving of his eight "Excellence in Engineering Education" awards from

Princeton University, including one for Lifetime Achievement (he was incidentally awarded a second last year, though it was later decided you could only win one per lifetime).

What sort of an explanation do I want to give? When we say that a doctor has preformed an astounding diagnosis, we are impressed, but we are not shocked. There is nothing mystical about how the trick is done – the doctor went to medical school, learned about the functions and disfunctions of the body, then during their residency and early career see thousands of patients with those same disorders. When we see an astounding diagnosis, we are likely to chalk it up to pattern recognition of a subtle variety of symptoms previously seen in similar patients, and rightly so. *How* it is being done neurologically, we have no idea. *What* is being done, however, there can be little doubt.

I similarly want to show how every capacity and concept a student of 217 derives from some part of instruction, thereby explaining any instructional choice by grounding it in its overall purpose. I will therefore attempt to determine, functionally, what Dondero is doing at any given point of his instruction and why he is doing it. Why this order of examples? Why this order of precept topics? A successful account ought to be capable of predicting which changes in student behavior and capacity are incident which particular pieces of instruction, such that a student having missed *those* five minutes of precept can be expected to incorrectly answer *this* question on the final exam. If another preceptor spends 3 minutes on something and Dondero spends 10, I want an explanation of the difference in cognitive ability imparted the more coddled students.

I am not trying to argue that there is no human element or personal touch in a great teacher's instruction. But surely that can't be all there is – a great chef has an incredible range of expression, but his or her food is still nutritious, functionally capable of sustaining the diner, by virtue of its ratio of carbohydrates, proteins, lipids, vitamins, and minerals. Mustn't there similarly be prerequisite intermediary steps to learning a language like C? And shouldn't there then exist functional categories of instruction – within which there may be vast room for personal variation – yet without which a lesson could not possibly count as effective? These are the functional categories I hope to find in this investigation.

How do I hope to accomplish such lofty goals? I will take a moment to outline my proposed methodology.

I will first to compile a complete list of the desired "outcomes" for any given student of the course – every capacity, association, concept, belief, etc., that a graduate should have.

Then, I will take a wholistic look at every instructional experience a student will receive, including lectures, assignments, online assistance, and, most importantly, precepts.

I will then, assuming everything Dondero does is optimal, attempt to show every student capacity to be the result of a particular part of instruction, match every one to every other, in terms that are explanatory and falsifiable by observation and experiment.

Why assume optimality? Assuming optimality is different, of course, than asserting it. A ready analogy is that of evolutionary biology. When we assume that every trait of a living organism serves some purpose, nature is free to dissent, giving us an example that is unexplainable by the theory of evolution by natural selection. Similarly, if no self-evident and explanatory theory results from assuming Dondero's instruction is functionally optimal, that would be strong evidence for it not being so. My goal is descriptive, not proscriptive – I am not in a position to prove that Dondero is effective as a teacher, but assuming as much, I can do my best to describe which methods might make him so.

Why COS 217? The students who take this course are a unique cohort whose academic backgrounds are highly diverse, yet whose experience with computer science is extremely consistant, given the single introductory prerequisite course and relative rarity of high school computer science. Any invariants in the learning process, particular conceptual developments that are caused by particular instructional experiences, and which conceptual developments are prerequisite others, for example, are likely to be more noticeable here than in a group with uneven prior exposure to material.

I would also suggest that the curriculum of COS 217 is uniquely apt for this sort of investigation. Cognitive scientists after universals of learning may err in studying simple educational experiences like young children mastering basic arithmetic. These cases are so simple, it is hard to distinguish between the various cognitive tasks being learned and preformed, as success is so near universal. Only when students can be expected to misunderstand meaningful parts of the material are we in a position to see which capabilities, concepts, and understandings are singular units, gained or missed together, symptoms of the same order or disorder, candidates for a single functional term.

Finally, I will not be using prior research or any psychological terminology beyond that which is colloquially used in everyday descriptions of psychological processes, such as "concepts", "facts", "knowing", "understanding", "explaining", and so on. When I need to, I will specify what I mean by a given word or phrase, restricting certain common terms to refer only to a subset of cases which properly merit their use.

It will of course be helpful to have prior knowledge of Computer Science and the COS 217 course. Neither, however, will be necessary to appreciate the general arguments and explanations in this paper. I will attempt to always document my references to course materials, and to give nontechnical analogies to the examples I use from the course.

I will begin by compiling a cursory list of the desired outcomes expected for a student of the course, then analyze the components and possible functional purposes of Dondero's precept, first with focus on his method of "tracing", then on his selection of examples, and finally on the particulars of his extemporaneous explanation during the lesson.

What exactly is COS 217 actually a course on? It's unclear. Instructors describe the course's subject matter alternatively as "programming in the large", the writing and maintenance of large interconnected programs, "programming under the hood", why operating systems and higher level programming languages work the way they do, and "power programming", a term which seems less like specification than self-congratulation. In a moment of candor, Dondero suggested that the subject matter of the course would be best described as "what should a second-year student know about computer science that isn't covered in an introductory course and a traditional 'algorithms and data structures' course."

Yet what could be termed a "grab bag" of material could just as easily be considered the lynchpin connecting the theoretical, systematic, and practical aspects of a computer science education. In lecture students of the course will be introduced to the types systems that underlie modern infrastructure like Linux OS and the Internet. An understanding of C is vital for a fluent understanding of modern computer and network architecture because that architecture is largely written in C. Students begin the course having taken 126 or its equivalent, in which they use the Java programming language to learn about the basic elements of programming, so the first goal of COS 217 is to transition those students to programming in C.

C is a lower level language than Java, meaning certain functionality that can be taken for granted in Java must be manually assembled in C. Compared to Java, C is a much more powerful and therefore dangerous language – Java has certain safety features that prevent programmers from harming themselves with common mistakes, like asking for the 11th element of a 10-item list – to this, Java responds by complaining to the user that there isn't one and stopping the program, whereas C silently obeys literally – giving you whatever is in memory after #10, even if there's nothing there at all. The errors you encounter in C are subsequently much thornier. It is a shotgun to Java's military rifle, and COS 217 students will spend many hours repairing the damage caused by shooting themselves in the foot with its less tame features.

Higher level languages like Python and Java are far more supportive when reading code, allowing the programmer to write quick and vague commands like "go to the airport" instead of more literal ones like "put on your coat, walk out the door, get in the car, take this particular route unless there's traffic, in which case…" and so on. Yet those fancier languages are written in variants of C, and must therefore occasionally loose something in translation, making a wrong or less efficient guess in figuring out how best to obey the programmer. C makes no such guesses, and therefore makes no such mistakes. By learning C, students of 217 will come to understand why more abstract languages behave the way they do, and what the costs of sugar-coating and hand-holding really are. Though graduates of 217 may never program in C again, they will have a greater command of higher level languages and frameworks, knowing when to go with the grain and take advantages of helpful features and abstractions, and when these amenities are more trouble than they're worth and it's better to start from scratch.

The most important concepts of 217, in fact, have little to do with C specifically. As we will see, concepts like "modularity", "abstraction", "separation of design from interface", and "defensive programming" are the most essential and most versatile of the course, applying to any programming language or computing system. More than just aspects of well designed computer programs, these are styles of thought, ways of breaking big problems down into manageable chunks and crafting a solution that will be clear and maintainable to one or many future programmers. These are the most important and versatile concepts in the course, and will stay will graduates far after they've forgotten the vagaries of C.

Students of 217 will therefore encounter a startling variety of interrelated types of knowledge, facts about algorithmic efficiency that seem like mathematical formula, methods of programming clearly akin to essay writing, modular problem solving methods more that are more similar to engineering, and a general phalanx of propositional facts that constitute an understanding of why things are the way they and not otherwise, both in our computer systems and by extension our modern world. And the nexus around which this foundational understanding will be build is the precept Dondero designed.

I. Traces

After the students have shuffled into their seats in CS 220, Dr. Dondero begins the Monday lesson by greeting them, “Good afternoon everyone. I hope you had a good weekend. Please sign in.” As thirty or so students and a back row of attentive preceptors settle in, Dondero will distribute an enormous pile of handouts, including, among many personally crafted summaries of the day’s material for later reference, the several example programs around which the lesson will be centered.

Dondero focuses his precept around a fully functional example program. Despite having an incredibly diverse range of points to make, he rarely resorts to hypothetical or contrived examples, always doing his best to situate the whole day’s material in one narrative and realistic series of programs. What Dondero does with these programs for the greater part of the lesson is trace them. Tracing a program involves going through it line-by-line, interpreting what each one makes the program do, much as an actual compiler would. While the student reads from their copy of the program, Dondero narrates and draws a “stack diagram” – a visual representation of what the program is storing and recalling from memory.

In Computer Science, tracing is generally known as a method for finding bugs in malfunctioning programs. A program has a bug when it produces unexpected behavior. Unexpected behavior from a deterministic machine implies an unclear instruction given to that machine to carry out. A bug, therefore, suggests that the programmer doesn’t sufficiently understand what they have asked the program to do.

A trace is the antidote to such ignorance – start at the top, work your way through line by line, and by the end, you can’t *not* understand what’s gone wrong. Understand the parts, and you’ve understood the whole. Tracing therefore proves itself equally useful for a second purpose, guided instruction.

For those without a background in computer science, a computer program often takes a task like “Open the door”, and breaks it up into many smaller tasks, like “1. Walk up to the door, 2. Turn the handle, 3. Pull the door open. 4. Walk through …”. Tracing works because if a student understands how all the subcomponents work together, then they understand the whole, whereas any misunderstanding about the functioning of the whole must break down to some misunderstanding of the parts and their interrelation.

Traces have the interesting effect of capturing different confusions from different students. Much like the historically successful Toyota assembly line, any participant who sees a possible mistake or confusion is expected to stop the progress of the group until the uncertainty is resolved. A class of students, all confused about different parts of the program, will therefore all have their individual kinks ironed out by the end of a single one-size-fits-all piece of instruction. This aspect is one of the reasons that makes tracing such an efficient method.

Often in standard classroom instruction, a student will not know whether they really understand the concept being discussed – something may be unclear, but they can't really phrase a well-formed question about it, and don't want the teacher to become exasperated from having to guess the source of their confusion.

In Dondero's classroom, that problem doesn't exist. Students have an easy way to check if they've fully understood the material: given the particular line of example code being discussed, could they reproduce and fully explain the reasons for it, alone, from memory? If not, then something must be missing. And because the material is doled out line by line, it is both apparent to the students *that* they don't understand, and to Dondero, given the point in the program at which confusion occurs, *what* the student likely is misunderstanding, even if the student couldn't properly phrase it in a question. There are only so many things one can misunderstand about any one line of code, and Dondero doesn't advance to the next one until everyone is satisfied.

Dondero will often clear up misunderstandings that no student is even aware they've made. He will anticipate when students are likely to draw an unwarranted inference[1]. An analogous example might be that of a history teacher making sure that an aside about President Roosevelt was associated with the right one, a question that wouldn't be raised by a student who didn't realize there were two in the first place. The trace therefore provides a foundation for the progress of the lesson.

[1] During the first lesson on pointers, for example, after telling the class that the size of integer pointers is 4 bytes, the same as integers, he cautions them that this is just a coincidence of the system Princeton uses, and that they should not expect the equivalence to hold in general.

Though students are free to ask questions about any line of code, most questions are posed by Dondero himself. In fact, the entire format of the lesson is centered around answering questions motivated by particular lines of code and posed by Dondero to the class. Dondero knows that a line of C including a new instruction will provoke a question about it, just as an English sentence with an unknown word would. A line that flouts a rule the student has come to expect will prompt a question as well. In this way, Dondero is in control of the class's questions because he not only anticipates them, but knowingly produces and preempts them with his example programs.

Each of Dondero's examples motivates a series of questions that allow him to hit every point he wishes to make in the day's lesson. By asking "What does this line do?" he can explain what the various terms in the instruction mean and what they are for. By asking "How does it work?" to a line of code, he can show what behavior and changes to memory it causes on the stack diagram. By asking for the "Why?" of a particular line, he can explain the intent and reasoning a programmer might have in writing such an instruction. Dondero's "motivated questions" comprise by far the greater portion of the questions asked in his precept.

In fact, one of the first things I noticed observing Dondero's classroom was the marked lack of emphasis on student questions. Dondero often solicits students to express their understanding by a show of hands or nod of approval. Far less frequently, however, does he answer a student question straight-on, with new explanation. Dondero usually responds to questions one of several ways. When he does answer directly, it is almost always in as cursory and briefly a way as possible so as not to get side tracked, providing answers roughly the same length as the question. He will occasionally cut off a student who is asking the very same question he is about to raise in order to transition to the next example – with a polite but curt "exactly." [1] He more often ignores the question itself and either restates the explanation he has just given again[1], or tells the student that their confusion should be cleared up after the next example. After giving the example, he will check back in with the student, and in every case I witnessed, the student was satisfied. Invariably it seems Dondero has anticipated the student's confusion, and his lesson includes an example intended to head off that particular misunderstanding. Somehow, Dondero seems to be tuned in to the timing and phrasing of even badly put questions, and is able to gleam from them exactly what misunderstanding produced the question, and whether the edifying material has already been seen or is yet to come.

[1] For example, "Multi File C Programs", W 9/26, After example 2, a student asks if the separated function declarations need to be in the same file as the definitions, which is precisely why the lesson is called "Multi File C Programs", and the contents of example 3.

[1] For example, "Pointers", Mon 10/1. Dondero points out that while if two pointers to be equal, it implies that their dereferenced values are equal, the converse is not necessarily true. One student asks a question, and Dondero simply returns to the board, restating his previous explanation, that both pointers can be pointing to the same "6" in memory, and thus their dereferenced values "6" must necessarily be equal to each other, but that the memory addresses of the two pointers might be 1012 and 1008, which are not equal. The student exclaims "Oh, oh, oh ok."

Traces do more than just effectively manage confusion. Centering the lesson around a real life program situates the student. A program is like a story – you have a narrative arc, a rigid timeline of past and future events, and even when the student doesn't yet understand what is happening in a particular scene, they recognize the scenery.

There is a fixed cost to situating a student in the context of a program, just as there is a fixed cost to establishing familiarity with the plot and characters of a play or novel. When grad student TAs teach Dondero's material, they often take tangents that leave the context of the program, explaining loosely related concepts or new lines of code they've written contextless on the board. Using the time tested metric of panicked looks on student faces, I estimate that these diversions almost always end poorly, failing to instill the desired material as well as breaking the narrative context of the program, making it that much more difficult for the student to reenter the flow of the program – (imagine, for example, how confused you'd be if reading a novel in which, after three chapters, a second story is abruptly entered, in a different time and place, told about different characters by the same names, who after a few more chapters then are replaced, unannounced, with the original cast, setting, and plot.)

There are several ways in which a student can be "situated" in a lesson. We might call two of them "local situation" and "macro situation". A student is locally situated when they understand what is currently being discussed, while macro situation answers the questions "Why are we here? What does this have to do with the larger goals of the course?" One can be locally situated, understanding what the instructor is saying, without having any idea yet what relation it bears to larger goals – this occurs, for example, when a teacher makes a diversion beginning "let me tell you a story". The student trusts that though they have briefly lost their big picture footing, it will be regained shortly when the "moral of the story" is told and tied back into the day's lesson and theme of the course.

One the other hand, a student can be situated in the macro, knowing the significance what is being discussed but not understand the particular explanation being tendered. The student knows why "this sort of thing" – a new C function perhaps – might be significant, they just are failing to grasp the thing itself. An example of macro situation without local situation is the conclusion of every Sherlock Holmes story – the reader knows "why we're here", that the detective has assembled a group of suspects that includes the guilty party and will reveal him or her through a dramatic monolog and cross examination, yet suspense is built because the reader has no idea why in particular Sherlock is asking those questions, or who he is about to finger. Dondero's instruction maintains both of these types of situation using the tracing method.

Whatever the difference between Dondero and the other preceptors, and whatever the functional significance of tracing as a method of instruction, I suspect the answer will have something to do with the ways in which a student must be situated to fully understand and parse the information being presented them.

One final experience I had observing Dondero's precepts was phenomenon I'll call "precept goggles." When Dondero is narrating a particular C program, you are temporarily in his sway, able to easily follow complicated programs using new and unpracticed concepts. Sense and purpose jump out of the code as you follow along. For the duration of the lesson, you are able to quickly and easily follow the flow and behavior of the program, as well as the intentionality each line represents to the programmer.

After the party, however, comes something of a hangover. On reflection the next day, outside of the glowing halo of CS 220, it becomes much harder to remember exactly what the program did, and the purpose for each line. Dondero has done his job, and students can piece it together with steady focus and the occasional textbook reference, yet there is a markedly different feel to being in Dondero's precept room that is difficult to peg to any particular of his actions and is unmatched in either solitary work or, in my experience, the guidance of other preceptors. I have only ever had a similar sensation when I once was watching a foreign film with subtitles and forgot I was reading at all, looking away from the screen for a moment, and was shocked to discover that the sound continued but my stream of semantic information was suddenly choked off. Rereading one of Dondero's programs the day after precept produces a similar feeling.

Why is this particular phenomena important? It is easy to see that Dondero is saying more things at a faster rate than do other preceptors. By all accounts he seems to be covering more facts, though admittedly many of them are tangential asides that do not yet seem to serve any functional purpose. The "precept googles" effect is significant because even though he is doing more, students seem to find it takes less effort to follow his explanation. A writer who can be understood with less strain is in fact better and more skillful than the producer of denser prose. It seems to take less effort to parse the information Dondero is giving you, despite there being more of it. Just as grammatical speech is easier to parse than ungrammatical but semantically sensible sentences ("I dog with go park", e.g.), the precept goggles effect may be evidence that Dondero is following some sort of grammar, following some set of predictable rules as he goes through his explanation of the material.

Tracing is not just an excellent method of lesson planning, it is also a type of explanation. By comparing it to other forms of explanation that involve similar elements, I think it will become more clear what functional role the trace serves in Dondero's instruction.

One type of explanation, seen often in mathematics, involves the conversion of a single fact between different notations. Many systems like Algebra allow problems to be solved and facts to be represented in several different formats. We can express that two linear equations intersect by showing algebraically that there is a value x for which the functions produce the same y, or we can show that the graphed lines of the two equations intersect somewhere on the x,y axis. Both notations are a way of explaining the same fact, one in terms of equivalent expressions, another in terms of intersecting lines that represent the same.

Why is it powerful to have multiple notations that express the same fact, and multiple explanations of a fact in different notations? Besides generally fortifying the student's understanding, some operations are cheap in one notation but expensive in another. It is simple to divide a decimal number by 10, all you have to do is move the decimal one digit to the left. If you want to divide by 3 however, you'd much prefer to be using fractions. Similarly, If you want to know the exact time, a digital watch would be perfect, but you want to know roughly how much time has passed without doing mental arithmetic, analog might be easier.

Additionally, some facts may seem arbitrary in one notation, while being obvious or "necessary" in another. A fact is "necessary" if it couldn't be any other way. It seemed arbitrary to me for example, learning algebra, that a quadratic equation couldn't have more than two roots. However, if you understand that the shape of a quadratic equation is one big U, and also understand that the root of an equation is the point it crosses the x-axis, it suddenly becomes obvious why a U can't pass a horizontal line in more than two places. Changing notations, from algebraic to graphic representation, makes a fact that seemed arbitrary clearly necessary.

Tracing is similarly a method of explanation that converts a single fact or event between multiple notations. During the trace, every event is fully explained in every notation before moving on in the program. Certain operations are also much easier to understand in one than another – pointer behavior is predictable given C code, for example, but it is much easier to follow when looking at boxes and arrows representing arrays and pointers on the stack than it is to recall semantically what constantly-changing meaning each variable name currently represents in the program.

Similarly, facts about a program which seem arbitrary in one notation may seem simple and necessary in another. It isn't immediately clear, naïvely looking at C code, why you aren't allowed to dereference an integer pointer and set the value to a double. Yet convert that event to a stack diagram, and it is abundantly clear – how do you expect to fit eight bytes of data into a hole only big enough for four? C also allows certain bizarre methods of referring to elements of arrays,[1] which become (marginally) more sensible when considered in terms of the arithmetic parsing being done by the compiler.

Dondero repeatedly takes advantage of the trace to show how facts and events in one notation line up with those in another, and to point out the necessary connections between the two. A line of C declaring an integer variable, i.e. "int var = 5" is a fact in one "language" that "means" the same thing as the box on the stack diagram labeled " var [5]". Students must be shown the various correspondences between related facts in different notations in order to know what to expect when writing C. Explanation by translating between notations, therefore, is one important functional aspect of explanation by trace.

It is often the case in science and engineering that one tries to explain the relationship between what something does and how it does it. One of the most fundamental insights of computer science is that certain computational processes can be described without reference to the particular object or substrate implementing them. The word "calculator", for example, refers to any object capable of preforming sums, subtractions, divisions, and multiplications. Before computers, "calculators" were people, and their job was to mechanistically preform the desired bookkeeping. Modern electronic calculators are only called that by virtue of their ability to do the same calculations that human calculators once preformed, albeit using completely different methods and circuitry. To use the computer science jargon, they are machines "built to the same specification".

Designed objects, therefore, can be explained both at the "what" level as well as the "how" level. We can talk about the calculator as a series of resistors and capacitors, or we can take a step back and discuss it in the abstract as something capable of doing calculation. When I ask for an explanation of a particular calculator, it is ambiguous whether I want a description of the operations it is capable of preforming, or the particular configuration of its parts that are used to preform them.

This "abstraction" is one of the most important concepts in computer science. To fully understand every operation that a modern computer preforms would take a lifetime. Programmers manage this complexity by knowing more or less *what* everything does, but knowing *how* only their own programs work. When the code you are responsible for works fine, everyone can happily use it without knowing how it was written, and if it breaks, everyone knows where to find you. The modern economy too depends on similar abstractions. When I go to the store to buy a light bulb, I can't be expected to take a course in optical engineering. Consumers need abstractions, wattage numbers and "E26 size bulbs" for example, which allow them to know what they need, without knowing why exactly they need it.

Explanation at the level of C code often is a description of "what" something does, while explanation at the level of the stack diagram is "how" that functionality is produced. Through

[1] Array names are evaluated to the memory address of the array's first element, and the element number is added to it using pointer arithmetic. Rather than referring to array[2], therefore, one can take advantage of addition's commutation and refer to 2[array], though it would be "in terms of understandability, terrible."– Wed 10/3, "Arrays and Strings" precept.

tracing, a student of 217 will repeatedly learn that certain abstract entities of other programming languages are really fictions, new interfaces slapped on old widgets, with all the benefits and drawbacks of the original widget[1]. An analog example might be an independent contractor legally purporting to be a company, interacting with his clients alternatively as "CEO", "Customer Service Representative", and "Head of Sales", failing to ever mention the total number of employees.

Abstractions are very useful time savers, but it is essential to know how something works if you want to understand how to make it more efficient or less error prone. By using the trace method to take students under the hood, Dondero can explain why certain design decisions of C were made, and under what circumstances it makes sense to override them. They'll also come to understand the reasons behind conventions and best practices, like type checking, which might previously have only seemed to be a bother. One is also in a better position to guess what C will do in an undefined or ambiguous situation, such as double freeing memory, when one understands the implementation of the abstraction in question. This is essential when doing the actual programming assignments, for diagnosing bugs often requires not just knowledge of the rules of C, but also an intuition about what sorts of behavior would occur if certain rules were broken.

Students also come away with a better appreciation for why seemingly arbitrary upper level behavior is actually a necessary result of lower level constraints. Bureaucracies are often full of practices which seem nonsensical and self-defeating, yet which are understandable when one considers them the result of a group of separately motivated individuals, each with different restrains in power and capability, rather than a single rational unit. Similarly I have a friend who took Princeton's introductory computer science course the semester after I took 217, and was complaining about how Java wouldn't let him declare an array based on user input – it was fine with a list of length "5", but not when specified by a variable that equaled "5", which he reasonably saw as inconsistent behavior. "All in good time," I said, and convinced him to take 217 the next semester. This particular inconsistency is explained by considering the compiler and program at runtime as different agents with different states of knowledge, one level of abstraction lower than my friend was currently thinking.

One analogy often made is that programming is a compromise between the human and the machine – the programmer has knowledge of program's purpose, who will use it and for what, but lacks enough mechanical knowledge to write the machine code himself. He can therefore only roughly outline what he wants the machine code to do. The compiler, on the other hand, is a highly efficient but narrow-minded robot who needs to be talked to in entirely objective and literal terms. C code is a compromise between the two – the human must translate their

[1] When languages pretend, for example, that an array of characters with a few helper functions is a different sort of thing entirely called a "string"

subjective values and intentionality into literal instructions, and the machine figures out how best to follow them.

Just as there are many ways a particular piece of functionality could be implemented, there are a variety of functionalities that could solve a particular intent or problem equally well. If asked to explain why an iPod has a headphone jack, one might sarcastically answer, “so that you can put your headphones in”, but that risks missing the larger point that an iPod is a solution to the human desire to hear music. An iPod could just as easily (though perhaps less politely) contained an enormous speaker capable of blasting music from your pocket. It is important to see that these are two different functional approaches to the same human problem, and their comparative costs or benefits must be considered in terms of human intentionality, not mechanical functionality. It is similarly overly literal to compare bicycles and cars solely in terms of horsepower and fuel efficiency – they are different types of solution embodying different human values.

When asked why a designed object has a particular feature, therefore, we can answer the question one of two ways. We can give a literal explanation of what functional role the feature serves, “the headphone jack is for your headphones”, or we can explain why that particular *type* of solution makes sense given the human problem and implementation restraints.

Dondero uses the trace to give both explanations, showing what functional role a particular line of C serves in the context of a particular program, and what style of solution that feature represents to the human problem. Learning how to translate competing values and intentions into functional specifications is the greater part of programming, and the most important skill for Dondero to impart to his students. Additionally, some facts about the behavior of C cannot be explained by reference to functional specifications alone – the language contains certain assumptions about the people who will use it, and what they plan on using it for. Tracing allows Dondero to give both explanations in reference to the same example, making the student’s understanding more fluid and less compartmentalized in when handling tradeoffs in design and function.

A final type of explanation related to tracing is called “teleology”, named after Aristotle’s failed attempt to explain the physical behavior of objects, such as falling, in terms of their “purpose” or natural place. Teleology now refers to any explanation of an object’s behavior in terms of its purpose, intentions, or goals, even when the object clearly cannot really have any such mental states.

The legitimacy of teleological explanations in areas like biology is hotly debated. The human heart for example has no mind of its own, and its behavior is just the mechanistic product of a contracting net of muscle. On the other hand, we have an attraction to describing its behavior in intentional terms, saying for example that “the heart wants to pump oxygen to all the parts of the body”, though it cannot literally intend to do any such thing. If a biologist were asked why blood pressure rises in patients with high cholesterol, they might say that constriction in the patient’s blood vessels force the heart to "work harder" to push blood through, though it is literally just a pressure system finding homeostasis.

Are these ascriptions of purpose and intention clearly wrong? We no longer believe them to be literally true, but they are useful abstractions. The heart has many behaviors we don't yet understand, but we can predict that whatever they are, they will be ones best suited to reliably pump blood. Additionally, due to the theory of evolution, we can say that while the behavior of the heart is determined by the mindless contractions of a million muscle cells, the particular configuration of those cells was honed by selecting for pump-like behavior. Here we see again a two-fold answer when we ask why the heart has any particular behavior:

On the one hand, we can explain reductively why *that particular* upper-level behavior is happening, showing the macro phenomenon to be a necessary result of countless mechanical interactions between mindless parts. One the other, we could explain those lower level processes, the blood cells circling the body for example, in purposeful terms, that they're "trying to bring cells oxygen". This teleological explanation is not only an effective and explanatory short hand – it is also in a sense literally true. Though the blood cells themselves have no intentions, they could be said to have a derived intentionality from being the product of the semi-designed system of natural selection. Similarly, while a program cannot yet be said to have intentions of its own, it is possible to explain its behavior in terms of the derived intentionality of the programmer. Teleological descriptions fail, however, when the designed object is removed from the context in which it can preform its function – when the heart is on the operating table, or when the thermometer is heated to 10,000° F.[1] A designed object can be described successfully in teleological terms when it works, while description of any disfunction must be in terms of the implementation.

Tracing is a type of explanation that similarly involves teleological elements. During the trace, Dondero often attributes intentionality to the various subcomponents of the implementation. A for-loop traversing a linked list in search of a desired key, he might say, is "looking for" that particular value, even though the iterating component has no awareness of the conditions under which it will stop. Teleological descriptions of implementation behavior provide important information. It is quite possible for a student to fully understand what every part of an implementation is doing literally, and also understand the designer's intent, but fail to see how the two line up, which aspect of the intentionality are being satisfied by a particular stack operation, or vice versa. Dondero must fluidly weave together each of these aspects, providing one story that unites the program's purpose and implementation.

[1] I am indebted to Daniel Dennett for many of the examples in this explanation, c.f. "The Intentional Stance" and other works.

In sum, Dondero's traces of computer programs are a fascinating example of explanation because they involve all of the functional elements previously examined – explaining a fact in multiple notations, explaining an object's functionality in terms of its implementation and vice versa, explaining different functional approaches in terms of human values and subjective tradeoffs, and finally explaining the parts of an implementation teleologically, in terms of the derived intentionality of the designer.

Whatever instruction Dondero provides in his biweekly 50-minute sessions must somehow hit all those points, satisfy every one of those functional attributes, connecting every fact and concept in all of those directions, not to mention providing students of wildly different backgrounds and capacities with the facts and concepts themselves, and situate all of that information into one coherent narrative series of examples so that the student doesn't even know what they're learning until they've learned it and the lesson is done.

Which motives our overarching question: how on earth does he do it?

II. Lockstep

Though other preceptors have adopted Dondero's method of tracing, I witnessed marked qualitative differences in the degree and rigor with which their traces were carried out. Dondero puts heavy emphasis, both in practice and in my interviews, on preforming traces in "lockstep". His adherence to this element may explain certain benefits experienced by his students.

As we saw earlier, tracing involves going through every line of a program, one at a time, following what effect each has on the standard output and memory storage. Sometimes, when the programmer instructs the computer to do a particular action several times in sequence for example, it makes sense to analyze a block of code together as a single event. Yet taking into account these meaningful chunks, Dondero stands by his word – each event is analyzed completely in every notation, "lockstep", before moving on to the next event. No skipping back and forth, no diversions except those tightly anchored to the current step in the lesson's chronology. As we established earlier, the example program is like a story, situating the student in a narrative that must occur sequentially to be comprehensible.

What exactly are the points that must be hit each time before advancing to the next step? Let's take a line from the first demo program in the course, "circle1.c". This sample program takes as input the radius of a circle, and outputs the circle's diameter and circumference. At one point, the diameter is calculated with the following instruction: "`iDiam = 2 * iRadius;`" Dondero indicates it as the unit to be analyzed by first reading it literally, "the variable iDiam is set to the value of iRadium times two". Then, he explains what the purpose of that line is in the context of the larger program, saying "this line takes the radius stored in the variable 'iRadius', multiples it by two, and stores that in the integer variable 'iDiam', so that we have it later on to print for the user." Next, he'll update the stack diagram, showing how the value of iRadius is recalled from memory, multiplied by two, then stored in the box labeled iDiam. Then back to the program, showing why those stack operations successfully carry out the instruction.

What Dondero seems to be doing functionally at every line of code is explaining "what" the code does, the meaning of the instruction in terms of the C language, "why" it does it, an explanation relating the line's functionality to the programmer's intent and the context of the larger program, and finally "how" the line is carried out, by jumping down to the stack level and showing what operations are preformed to implement the desired functionality. These three points, the "what", "why" and "how", must each be explained before moving onto the next event.

How does Dondero hit those points? First, as we saw earlier, facts that seem confusing or arbitrary in one notation may seem simple and necessary in another. Certain contortions required at the C level are easily explained by visually graspable constraints imposed at the stack level, one's inability to fit 8 bytes of data into a 4 byte hole, for example. To explain why the program has a certain functionality requires an appeal to a higher power, the intent of the programmer. Similarly certain lines of C are easily explained by jumping to the stack diagram, the reason for certain stack operations is clearer when explained teleologically, attributing the intentionality of the programmer to the behavior of lower level parts ("storing the evaluated expression in iDiam" therefore becomes "saving the calculated value of the diameter somewhere we can find it later to print to output").

In this sense, I believe that intentional language is best considered to be a third notation for representing the program, and that to describe a particular line of code or part of the stack diagram with intentional vocabulary like "wants", "tries to", "looks for", "waits", and so on, as opposed to literal and mechanistic terms like "stores", "recalls", "computes", "jumps to", etc., is an act of translating between notations akin to moving between the other two, or as we saw earlier, between algebra and coordinate graph.

Why might it be important to maintain a lockstep delivery in this sort of program trace? Each event is a separate meaningful unit, and each has a "what", "how", and "why". In order to produce a single understanding in the student's mind, Dondero must hit each of those points in rapid sequence, producing a unified conception of the event with three fluently interchangeable aspects.

Some preceptors seem to act as though the interrelations between the three logically follow automatically from inference, and that explaining the three individually is sufficient. Yet there is no reason to believe this is true – the connections between facts are separate knowledge that must be explained separately if we want to be sure it is imparted.

Many know for example that Napoleon lost the continental wars he was engaged in, and that Jefferson got a great deal on the Louisiana Purchase, but relatively few high schoolers are aware that the worsening stature of Napoleon's army created the need for cash that coincidentally benefited Jefferson. Analogously, just because a student knows that C is standardly built with a two-pass compiler, and also knows that a function declaration or definition must precede a function call, it doesn't follow that they grasp why the one entails the other. In fact, the larger part of student mastery can be explained by more and denser interrelations between the same standard set of facts.

We can therefore, I think, find an invariant principle in Dondero's methodology: connections between facts must be spelled out. If a fact in one notation explains another, he'll tell you. If a behavior at one level of abstraction is more easily explained in another, he'll tell you. If a particular choice is one of several equivalent ones the programmer could have made just as well, he'll tell you. If a fact is due to an unbreakable law of computation and couldn't possibly be otherwise, he'll tell you. Assuming that Dondero is actively explaining every interrelation between facts goes a long way in explaining the differences between his style and that of other preceptors.

Lockstep delivery therefore is essential to honing the student's intuitions about causal facts – which lines of C cause which lower-level behavior. First, there are often lines of C that are reliably interpreted in certain ways that the student should be aware of. When declaring strings, it is wise to be careful if you plan on changing their value later on, as certain notations will cause the string to be saved in read-only data, as opposed to the stack. By seeing certain lines of code always produce certain behaviors in memory, the student comes to associate the two and trains their intuition about causal relations. Second, it is sometimes important to know which parts of an instruction are responsible for which parts of an event. Dereferencing an undefined pointer is risky, but you aren't in trouble until you actually try to put a value there. By maintaining lock step delivery, Dondero treats as separate events that other preceptors group together, producing intuitions that are far more finely tuned. A dangling pointer, for example, should produce in a student a mild unease equivalent to seeing heavy vase a bit too close to the edge of a table. Nothing bad's happened yet, and it's fine as long as you keep a constant eye on it, but any momentary lapse or sudden move could knock it off the edge (causing, in our analogy, data corruption and hopefully a segmentation fault).[1] If the creation of a dangling pointer and its dereference are explained together, the student will have a more blunt understanding of which particular lines of code caused which parts of the problem. These sorts of causal intuitions can only be produced by explaining the effects of each event individually.

Another corollary of this rule is that students should not just be shown the correct way of doing something, but shown every possible way of failing and the varieties of damage each one causes. Dondero's examples are filled with lines that seem innocuous but produce horrible data corruption and segmentation faults. Not only do these examples help hone the student's understanding, analyzing them in lockstep imparts how prior events can make an otherwise innocent instruction deadly, and what best practices could have avoided the error. In these cases, particular lines of C are not always black and white, definitely error-prone or always good-to-go.

Finally, lockstep tracing builds into the lesson a bit of redundancy. Though we can never assume that every student will make every inference, any given connection between facts will probably be guessed by a large number of students, thereby putting some slack in an otherwise rapidly paced lesson. As we saw earlier, the trace is a fantastic tool for creating a one-size-fits-all piece of instruction that will resolve many unique student confusions. Start at the beginning, work through one line at a time, explain every fact and possible inference the student is expected to make, and by the end they can't *not* grasp the whole thing.

Given what we've seen so far, it seems clear that there is a pattern in Dondero's instruction formed by maintaining rigid sequence, and traversing every possible connection between different facts and notations before advancing to the next step. Next we will see if we can match this pattern of instruction to a set of cognitive tasks and capabilities that might cause them.

[1] After producing a segmentation fault, Dondero will often caution "you should feel lucky! Imagine how much worse it'd be if it corrupted all your memory and you didn't even find out!" which is good advice, as the student will likely spend hours fixing segmentation faults during the programming assignments – "Pointers", Mon 10/1

III. Concepts

I was recently listening to Jay-Z song on the radio. Just as it was getting to the good part, there were sudden gaps in the audio. Not in the instrumentals, I realized on closer listening, only in the lyrics. Immediately an obvious explanation occurred to me: I'm listening to a censored version of the song because it's on the radio, whereas I'm used to listening to the uncensored version on the album. That explains why the gaps are only in the lyrics.

This deduction occurred over the course of a few brief seconds, and would have been promptly forgotten had it not provided a fairly good example of what I am about to call a "unified" or "furnished" concept. Someone can know the meaning of the world "censorship" and still be flummoxed by the gaps they hear in a radio broadcast. A lot of prior knowledge helped me to infer to that explanation. I've had the word censorship defined for me. I've been presented with cases of explicit censorship in history, and have had instructors indicate which aspects of the example evoke the term. I've seen fellow students riled by acts of administrative "censorship" that I didn't think really merited the term. I've seen actions that connote censorship despite not literally involving cut tape or blacked out text – situations where someone wanted to say something but knew an observant authority would in some way punish the transgression. It must be from these experiences that my capacity for inference derives.

There seems to be a difference, therefore, between roughly "demarked" concepts, ones produced through cursory description but no examples, and the more mature and practiced concepts, "furnished" with a diverse and interrelated set of examples. Demarked concepts are isolated, and must be referred to by name directly to call to mind. Furnished concepts are expansive, and come to mind unprompted whenever we see the phenomena that fits the pattern. After mastering the concept "diminishing marginal returns" in economics class for example, the term frequently comes to mind when I see a situation where more of a desired thing produces less of an intended result, even when goods and money aren't involved. Concepts are a kind of pattern recognition, and the better we understand them, the better we can recognize instances of the pattern unprompted.

When a student is debugging their programming assignments in COS 217, there is never a signpost indicating the source and type of error, with "dangling pointer" in large red print. Students are expected to observe unexpected behavior and to imagine what sorts of mistake might have caused it, preforming "inference to the best explanation" in forming a hypothesis, then conducting an investigation. Just as I diagnosed censorship to explain unexpected behavior, debugging requires a well-furnished concept of any potential bug, capable of being called for indirectly by observing abstract clues.

In order to diagnose a dangling pointer as the source of an error, a student must not only know diagrammatically what a dangling pointer looks like – a box in local stack memory that points off wildly into space at nothing in particular (hence, "dangling") – they must also have an understanding of the sequence of C instructions that produces a dangling pointer. No single line is the culprit, but rather a vague conspiracy among several – one that asks the operating system for a block of memory to use temporarily, one that creates a shorthand arrow to that memory for easy use, and one that tells the system that we're done with the block, and returns it for recycling. We are then left with an arrow pointing to the spot where our extra memory box used to be, but it's ours no longer. These three lines of code create the "dangling pointer", but not an error – just like the vase teetering dangerously near a table edge, there's no actual problem until the programmer asks to do something to the memory in question, an innocent request that springs the trap, corrupting memory, crashing the program, and throwing an error.

Thus, when the student goes to find the code that triggered the problem, it could be hundreds of lines away from the actual source, the components of which could themselves be scattered throughout the program. A student must therefore see past the literal presentation of an unexpected behavior such as data corruption or a segmentation fault, and see that behavior in the larger context, as an effect of an abstract cause, a memory trespass the sort caused by incautious production of dangling pointers.

That student needs to know more than just the meaning of the term "dangling pointer". Their concept must be expansive and practiced, subsuming a range of facts about C and stack behavior. In order for the student's concept of dangling pointers to encompass these equivalent facts across several notations, their concept must have been furnished by examples presented in lockstep across those notations. It is Dondero's inclusion of fantastic blunders in his example programs, along with his careful lockstep traces of what exact damage they do, in every notation, that produces the furnished concept of "dangling pointers", enabling students to diagnose them unprompted. We discussed how lockstep tracing might produce better and more finely grained causal intuitions, and now we see these would well suit a debugging programmer.

How else does Dondero produce these flexible "furnished" concepts, as opposed to merely "demarked" ones? It was previously noted that Dondero never assumes any inference is obvious, and spells out every connection between facts. When explaining a concept, he similarly shows examples of it in every notation, preparing students to recognize the entity in any possible presentation.

When Dondero teaches an important category with many types of members like "techniques of abstraction", he doesn't just show many examples, but also relates particular examples to others, showing them all to have a family resemblance, deserving of a single class.

Just as Dondero shows a diverse set of prototypical examples for each concept he teaches, so too does he show prototypical examples of facts (or "propositions"). If he wants you to understand that "cats meow," he'll show you, not just tell you. Seeing examples of a dangling pointer causing memory corruption furnishes the proposition "dangling pointers are dangerous" in the student's mind. Similarly, there are noticeable differences in the behavior of students who understand the proposition, having been led through examples, and those who merely know it, by being told it by a trusted authority. Propositions can be furnished or merely demarked in the same way concepts can be. Those who merely know “stack resident arrays must be declared before compile time” might nod, agree its true, then go declare arrays with integer variables from standard input, in direct contradiction of the fact. We see again that failing to fully furnish and interconnect facts and concepts can produce segmented knowledge, inconsistent beliefs, and irrational behavior. There is a difference between knowing a fact is true and acting like it, and it’s the difference between having seen examples that furnish the proposition.

We can make a generalization from all this that if Dondero wants a student to use a concept in a particular way, he gives them examples that make them practice it. If he wants you to be able to recognize and diagnose problems caused by dangling pointers unprompted, he’ll show you more and different examples than if he merely want’s you to grasp the meaning of the term “dangling pointer” in a sentence. We could call this the “practice makes perfect principle” – never assume that theoretical knowledge alone will put students in a position to make correct applications in the wild.

These commonsensical dictates probably sound quite uncontroversial, so let’s take a moment to imagine what sorts of misunderstandings and incapacities might be produced by failing to follow them.

Someone might, as we mentioned, know the meaning of a word like “dangling pointer”, but fail to recognize examples of such bugs unprompted for lack of experience with the sorts of symptoms they cause.

By failing to fully spell out connections and synonymous terms, students can form compartmentalized knowledge which produces inconsistent beliefs. For a simple example, you might be visiting a University and meet with the president, who happens to also be the chair of the Philosophy Department. You know it to be the case that you met the president of the university, but without the intermediary fact, would falsely deny having met the head philosopher. Similarly in 217, a student might know what intentional purpose pointers serve, correctly identifying their use as the correct approach to a particular problem, yet might not realize that pointers are just a special use of the integer variable, producing nonsensical beliefs like, “the memory address isn’t a number, it’s a place, and the pointer is an arrow”, when really, the arrow is an abstract representation of a number which represents an address. Often when students pose incoherent questions, it is because they have incoherent concepts, concepts with components which were not made to cohere by tightly binding interrelated facts.

On a related note, students can become overly reliant on “what” level abstractions, not connecting them with “how” level implementation. Most American teenagers not only cannot drive cars with a stick shift, they do not even fully understand what an “automatic” car is automating. So reliant have they become on the abstraction, that most cannot go without it. Similarly, in industry, students dealing with databases who don’t understand which commands take advantage of the cache from a previous query, and instead which go back to the database all over again, could end up with code thousands of times less efficient than a correct implementation. To use a similar example from the course, operations like matrix multiplication involve choosing whether to go across a grid by rows or by columns. In abstraction, the choice seems arbitrary, but when that abstraction doesn't line up with how memory is stored in page-size chunks, your program will cause enormous inefficiencies.

Unified concepts, on the other hand, allow "responsible abstraction", the ability to deal with an abstract object as a seamless unit, but also to drop down and conceive of its parts when need be. We can determine that two SQL statements are functionally equivalent, and only then determine their efficiency. A programmer can be taught to act as though they understand the lower level by blindly following rules – "in matrix multiplication, always go in column major order" – but at some point, it becomes more efficient to understand the reason than to the memorize rules. Understanding reasons is the ultimate lossless compression – not only can you produce all the particular the rules for free, but you'll have better intuitions – I've long since forgotten the particulars of how function calls are implemented in assembly from when I took 217, but I still remember that C function calls aren't free, as I once thought.

There are three factors, it therefore seems, that separate a fully furnished concept from one which is only provisional and demarked – fluency, breadth, and unity.

First, facts about the concept are understood in every equivalent phrasing and notation, and can be grasped, abstractly, without reference to any particular one – pointers are both integer variables and "boxes with arrows", but can be handled in abstraction from either of these particular notations. This ability to translate easily between different aspects we can call fluency with a concept, which Dondero achieves through lockstep tracing and the spelling out of connections between facts in different notations. When a concept lacks fluency, we should expect to see knowledge in unbridgeable clumps, producing inconsistent beliefs and failures of inference.

Growing up in Manhattan I would on occasion visiting Central Park, usually going by subway and entering the park at the southernmost point. Despite knowing my way around uptown Manhattan on either side of the park, and how to get around within it, I never have any idea how the two worlds line up, and when I exit the park East of the Turtle Pond, I could just as easily in my mind end up on 60th St. or 120th. This misunderstanding is an example of how local clumping of interrelated knowledge results in global inconsistencies and inabilities of inference.

The second differentiating factor between furnished and demarked concepts is exposure to a diverse set of prototypical examples. We might call this the breadth of the concept, which Dondero produces by showing representative examples of every possible type, and by having students practice recognizing unprompted the different "faces" of the concept, the various ways particular instances might present themselves in the wild. Beyond just knowing that the different types of a category count as such – tigers and lions as "cats", for example – a student should also understand *why* each counts as a member of the same category. We might call such a concept "unified" if it was clear why all of its different instances should be meaningfully united as the same. Dondero achieves this by drawing analogies between members of an abstract category and pointing to the salient details of each one that make it count as a member. If we fail to give the concept of a student breadth and unification by showing broadly representative examples, and uniting them under a single meaningful category, we can expect that student to fail in recognizing instances of the concept unprompted, and in grasping analogies to that concept in contexts which are not literally identical, but share functional characteristics, such as how examples of pointers are used in a language like Java, despite not literally being a part of the language's specification.

In general, abstract concepts like "tool" can only be furnished if a child already knows about particular tools, like hammers and screwdrivers. Similarly, students of 217 are given many examples of programs that use for and while loops before they are expected to think about "iteration" in the abstract beyond any of these particular constructs. If you skip this step, trying to explain to students an abstract category or concept before they have examples to moor it to, they will never surpass a cursory demarked understanding of the term, and will never be able to use it flexibly.

For short hand, we might say that concepts which are fully furnished in all these ways– fluent across notations, broad in their application to a diverse set of examples, and unified by establishing a family resemblance among all members of an abstract category – are "understood", whereas concepts which are demarked by name without examples are merely known. This distinction, I think, is continuous with ordinary usage of those terms. I know that Guam is a place but know nothing about it nor would I recognize my location if dropped there. My concept is lonely, demarked by just the name "Guam", and containing only the attribute "a place". Similarly, you can "know" what pointers are by memorizing that they are "an integer variable whose value is a memory address". Yet instructors who stop there have produced students only capable of cocktail conversation at a Google luncheon, not programmers capable of passing the technical interview.

It seems clear that unification and breadth are important in concept formation, but even a mediocre instructor can give a few examples. What makes Dondero's instruction on this point exceptional? We will see that Dondero achieves these ends by creating "mental models" that hasten the integration of student knowledge.

As was mentioned earlier, understanding reasons is more efficient than memorizing rules. Yet sometimes reasons are very complicated, and while it pays to learn them fully once, we'd like an intermediary understanding as a heuristic for day-to-day use. Additionally, students sometimes have been shown to suffer from the "fan effect", where the more information they learn about a topic, the harder it is to remember any particular fact about it (similar to how the more names there are in the phone book, the longer it takes to find any particular one.) These students need a better way of mentally organizing new information for quick use than pure association and memory.

It is for these reasons that Dondero puts so much effort into producing “intermediary mental models” – scaffolding concepts that structure new information, and which can be more easily referenced.

The first such mental model you probably learned in grade school is the number line. Why do we teach kindergardeners to place numbers on a horizontal axis, instead of just imagining them individually? Kids can just memorize that three is lesser than four, so surely we’re wasting resources on teaching then to draw pictures, right? Well in fact, while the group of toddlers memorizing inequalities would have a head start over their more deliberately-paced peers, eventually the tortoise would win the race. The number line provides us with scaffolding to associate new information around old landmarks like 500 or 1000. The number line lets us cheat, tagging 697, “just under 700”, which is good enough to declare it greater than 34. Additionally, every numerical fact we learn is constantly being integrated with our previous knowledge of every other, aggregating that information to give us useful intuitions or “gut feelings” about which sorts of numbers are appropriate for a the price of a Cappuccino or the number of cufflinks anyone really needs.

Students who fail to learn mental models like the number line will seem to progress until they abruptly hit a point where memorization simply fails and their knowledge base collapses under its own weight. Teachers who do not invest in mental models are guilty of a sort of negligence, and it may not be until much later in the student’s career that their shoddy foundation work is discovered. Regular methods of testing, similarly, cannot be expected to pick up on these differences in mastery in the short term, as students can become highly efficient at cramming. The moment of truth, rather, will come six months after the final, when the less methodical have forgotten most of what they learned, and those who mastered the mental models can still recite it.

A similar mental model is the organization of one’s geographic knowledge on a bird’s-eye view map. Rather than memorizing how to get from every location in Central Park to every other, which would become computationally cumbersome, we slowly construct a map by relating landmarks and main pathways, then use that map to organize new information like shortcuts. In fact, the park is was designed to make this sort of navigation simple.

Yet the benefit of mental models comes with certain risks on which we must keep an eye. As I mentioned earlier, my mental model of Central Park is detailed in some ways, and lacking in others. Because my mental map of upper Manhattan is segmented from my map of the park, it occasionally gives me incomplete or wrong geographic intuitions. Mental models should be fully integrated with prior knowledge, left to right, as in my case of the park, and top to bottom, uniting upper-level abstractions with lower-level behavior. Mental models also must be constantly maintained to stay effective, always integrated with new knowledge at the particular points on the model it will be called for. “I found an alleyway” is useless information unless stored under “shortcuts between my house and the grocery store”. Such storage can also fail to be reciprocal – if a particular path is the shortest distance from point A to point B, or an explanation is bidirectional between two facts (e.g. equal adjacent angles in a triangle mutually requires equal adjacent sides), this is a connection that must be "spelled out" in both directions, associated with the path from A to B and B to A. Mental models are finally like any abstraction or shorthand – useful in some cases, and more harm than good in others. Students should learn to use certain mental heuristics with an awareness of the risks.

Dondero uses mental models at the large and small scale. It was mentioned earlier that Dondero wants a student who imagines a dangling pointer to visualize a box with an arrow coming out of it and pointing somewhere meaningless off in the distance – this is a "shorthand" explanation for what a dangling pointer is and why it's dangerous. He then associates several types of C code capable of producing dangling pointers to this same image – freeing memory without setting the pointer to null, creating undefined pointers, and setting the pointer to the value of a casted integer, for example. All of these actions are dangerous, and all for the same reason. Rather than learning the same rule three times, he connects the three examples to the same mental image, saving time and reenforcing the shared concept. Then, the various symptoms that identify a dangling pointer are associated with the image of the dangling arrow. Thus, a student can mentally traverse from any particular error message, to the dangling arrow picture, to the sorts of code that might cause the dangling arrow picture, and quickly check if their program contains any code on that shortlist. These interconnections centered on the mental model will save the student hours in the debugging of homework programs.

In the macro, Dondero's stack diagrams are the most important mental model of the course. Not only are they a simplified abstraction of what the actual stack looks like in memory (students will learn the difference when they go spelunking in memory during the "Buffer Overrun assignment), but Dondero will later use the stack diagram model as a scaffolding to integrate new knowledge of assembly, an intermediary language between finished C and machine code. Students already are fluent in the relationship between various lines of C and the changes they make to the stack diagram from Dondero's lockstep traces. Instead of just having students memorize which lines of assembly are equivalent to which lines of C, Dondero will fully integrate assembly into the stack diagram, just as he did for C originally, showing which stack operations are caused by which assembly instructions. This makes the graphical language of the stack an intermediary tool of translation between C and Assembly, forcing students to actually think programmatically in assembly itself instead of just writing lines of C and translating, the equivalent of actually thinking in a foreign language instead of just translating back and forth from what you want to say in English. Here, all the student's work in building their intuitions about stack diagrams pays off when it comes to rapidly integrating assembly into their prior knowledge. We can begin to see how Dondero uses these intermediary mental models to unite related disparate knowledge around "hubs" of explanation, making understanding fluent and recall rapid.

There's one final mental model you probably already know and love – the story. Humans are hardwired for narrative, using stories to frame our interactions with each other, ourselves, and our world. We care so much more about human stories than machine instructions, in fact, it should come as no surprise that there are simple ways of repurposing our narrative faculty to strengthen our technical concepts.

It was mentioned earlier how attractive we find teleological explanations, ones that attribute human beliefs and desires where they might not really belong (e.g. "the heart is trying to get all the body parts oxygen"). Yet, on the other hand, teleological explanations can be highly predictive and explanatory when discussing a designed system like a computer. Though the running processes don't literally have beliefs and desires – don't "look" or "wait" for anything – they sure act like they do. This is because their behavior derives from the intentions of the people that designed them. By telling ourselves the same stories as the designer, we can much more easily predict what a program will do under optimal circumstances. However, like any mental model, stories need to be taken with a grain of salt – designed objects will only behave predictably under the exact circumstances that they were designed for, and though they share aspects of their designer's intent, they all together lack their maker's judgement. Put a Roomba vacuum cleaner on a highway, and it will just keep vacuuming.

Stories are a method of compression, taking a long series of literal events and emphasizing only the ones that are functionally important to us. Dondero often explains low-level program behavior with purposeful terms – the program puts a variable on the stack in order to "save it for later", attributing aspects of the programmer's intent to the various subcomponents of the implementation. As such, stories can be a useful tool for remembering and handling complex system behavior, so long as you're careful – the stack doesn't actually care about saving your data anymore than your red blood cells care about bringing you oxygen. Even when an explanation doesn't literally treat a program as a little person with beliefs and desires, a technical explanation can still contain "story elements", brief analogies to human characteristics that aren't meant to be taken literally – a program acts the way it does because "it couldn't know" something, meaning literally that it's programmer couldn't have possibly written it to access a relevant piece of information. These little "white lies" are illustrative and harmless sugarcoats that the student should understand to be shorthand.

Just as there are patterns among particular combinations of letters and particular meanings in English, there are patterns among particular generic chunks of code and particular intents the programmer probably had in writing them. Taking advantage of this, skilled programmers can navigate code more easily by reading it at two levels – literally, as the machine does, and as a story, mapping chunks of intentional meaning to chunks of code. This is why programmers often dislike being interrupted in the middle of a coding session – there is a fixed cost to situating oneself in a program and learning to navigate it as chunks of intention. This semantic information overlays the code with meaning in the programmer's perception just like knowing how to read overlays the sight of letters with sounds. Dondero builds up this ability in his students the same way a kindergarden teacher does, showing them how to read each line intentionally, then letting them practice.

Sometimes Dondero takes advantage of even more primitive associations in describing code. When discussing defensive programming, all the ways in which a developer must defend their program against irregular and unexpected inputs, Dondero takes the particular practices that defensive programming entails (input validation, checking that function arguments are non-null, etc.) and makes a strong analogy with actual self-defense and poison. He will use terms describing particular lines of code as defending the program against attack from someone who has criminally violated your program's contract, or protecting oneself from dangerous input. Here he is prompting the student to map their intuitions about danger and sanitation directly to particular lines of code. It might seem sufficient just to memorize all the best practices involved in catching and handling undefined input, but uniting all these practices with the student's prior instincts about self-defense and sanitary food drives the point home in a way much more visceral, and therefore, more likely to come to mind as a reminder even when it's late and the assignment's almost due – primitive instincts can override pragmatic concerns and laziness in a way that "best practices" can't. Another example we saw earlier of this was Dondero's associations of dangling pointers with the student's intuitions about danger in the case of the vase near the edge of the table, another highly effective mapping.

These semantic associations, however, can only come after one has a thorough literal grasp of what the program does. One has to recognize that a Lamborghini is a type of car before one can learn that it is an expensive car, and you'll need an understanding of how a car works before you can hope to understand why a Lamborghini might deserve that association. Similarly, associating the assert statement with self-defense and the careful cooking of one's food can't come before the student has a thorough grasp of the assert statement itself. We might summarize this point by saying facts which are "positive" (not as in "certain", but as in "a literal, objective, value-neutral statement") precede associations which are "normative" (subjective, value-laden, ascription of "good" and "bad"), or, "positives before normatives".

Once the positive facts are garnered, the programmer needs contextual experience mapping types of code to types of semantic and normative concepts. To determine if code is "sloppy", you need to know who wrote it and what purpose they wrote it for – code which is conservative for a personal project might be completely slipshod in industry. Code that you write for others to maintain is similarly more likely to count as "irresponsible" if it lacks proper comments and documentation.

A fully grasp of what "sloppy" code looks like, for example, requires not just associating lines of code to semantic categories – you need to add an understanding of context. In the real world a similar example is tone – formality might be a sign of respect with a boss, but a sign of anger with a spouse. In order to recognize anger or sloppy code, we therefore need a prior understanding of what would be appropriate for the context in order to determine if a particular case is aberrant. I similarly couldn't explain to you specifically what sloppy code looks like, the best I could do was give you some examples, explain their context, and point to the points of departure from what we would expect to be an appropriate level of care given the circumstances.

Stories and semantic associations, therefore, and the most flexible and interesting of the mental models Dondero invests in. They not only provide an outline that can be used to structure new information, but enable all sorts of important cognitive tasks like programming intuition and augmented reading of code. The sorts of concepts that can produce this type of mastery are broad and abstract. They will mark the difference between competence and mastery.

Over the course of their computer science career, a student builds their understanding from the bottom up. First they learn to satisfy certain programming prompts. They will recognize patterns – when asked to write a program that does something several times, use a loop. When asked to store data across functions, use a global variable. Students must learn why the naïve answer, global variables for instance, is correct before learning what new problems it introduces. The student will, with practice, see that global variables end up producing very hard to fix bugs, and learn to use local variables instead. This is an example of a one-off fix to a problem arising from the naïve solution to a programming prompt. The student will learn other such one-off fixes, writing test code, and using more descriptive variable names. Only once the student is aware of many such fixes will they start to see patterns. A particular bug might be literally caused by forgetting to consistently change a constant value throughout the program, but taking a step back it becomes clear the bug was caused by using hard-coded constants in the first place. Patterns begin to emerge as students realize entire classes of bugs can be fixed by best practices like descriptive variable names and short modular functions. Finally, the student will begin to see patterns in these best practices themselves, unique ways of thinking that unite particular practices like avoiding global variables and using descriptive function names as derivatives of the same abstract perspective on programming.

These highest level patterns form the “pervasive” concepts. Modularity, abstraction, design-by-contract – they are the white whales of the course – the most vital to mastery, the longest remembered, the most cross-applicable to other fields, and the hardest to pin down to any single moment of instruction. These broadest concepts will be hardest to unify, and their many facets will be a challenge to deal fluently with. We should pay close attention to their formation and use, for any truths to be gleamed about robust concept formation must apply to these, our most extreme cases.

In one sense these concepts are highly intuitive, and a programmer must often rely on nonverbal feelings that something in their code isn’t right. Yet in another sense they are as objective as any true or false pronouncement in the real world – we rely on our intuition to determine if someone is angry, but there is still and underlying fact of the matter. Modular programs are a similar case, where experts may each describe the term differently, and make different design choices when programming, yet still maintain a strong nonverbal consensus about which particular examples count as modular and by virtue of which particular parts of the program.

Pervasive concepts cannot be explained for the same reason recognizing sloppy code and angry words can't be reduced to a set of rules – their presence or absence in a situation is determined by abstract patterns in highly context dependant features, depending on the program's purpose, users, and prospects for long-term use. Expert programmers often "feel" that a piece of code isn't modular, and it is very difficult to teach a feeling to a classroom of students. How can you learn what cannot be directly explained?

The way we learn pervasive concepts may be similar to how we learn the meaning of words as children. It is very difficult to come up with a single definition of the word "game".[1] They're not all played for fun, as some sports can be played professionally and games of chance, out of addiction. Not all games have multiple players, for example solitaire, nor points and winners, like catch, and so on. Yet despite it being impossible to come up with a single set of rules for membership in the category "game", we use the word with complete fluency and consensus, even when describing novel examples or making analogies to other domains, like "games" played in conversation. We also have an instinctual reaction when we see a word misused – even if we can't explain what rule it breaks, it just seems wrong.

Just as one builds up a highly broad and abstract concept of "game" by first seeing adults use the word correctly in a variety of contexts, so too must must one first see many examples of "modularity" and "abstraction" in the context of particular programs which are fully understood literally. We form robust abstract concepts by seeing many literal examples and abstracting the similarities.

Dondero needs to invest in many such rich examples before the student begins to understand a pervasive concept like modularity. First the student must fully understand why a program solves the problem, why it literally works, then Dondero can point to features of the program that are still problematic and suggest improvements which would make it more modular. In this way, Dondero is always explaining modularity in the context of a particular program, pointing to aspects that he has just literally explained to the student and reexplaining them in the context of modularity. Case by case, this is how he builds up pervasive concepts, and accordingly the intuitions of his students when programming.

When does Dondero teach these pervasive concepts? Fundamental themes can't be taught on any particular day, but should pervade throughout the whole course, taking advantage of examples that come up chronologically by making tying them into the larger themes. It makes sense, for example, for a history teacher to discuss the importance of federal-state power or wartime suspension of civil rights following a unit on the American Civil War. Dondero similarly takes every opportunity to tack a plug for abstraction or modularity onto the end of any relevant unit or answered question.[1]

[1] Example from Wittgenstein, "Philosophical Investigations".

[1] During my observation, an online group discussion formed regarding the relationship between ASCII character standards and C. A student had confused themselves thinking that because characters are stored as integers

Though these themes are the hardest to instill in a student's understanding, they are the most useful in real life. No one will quiz you on the when the treaty of Versailles was passed, or try to catch you off guard about the names of Henry VIII's wives. More likely, you will see someone brazenly state that a particular politician's actions are "unprecedented", or refer to a practice with little historical basis as being a part of national heritage. Moments like these are why we learn history; moments like these are separate memorization from mastery. Pervasive concepts are the conduit that connect our learned facts and prior knowledge with situations that arise in daily life. It is by their virtue we can say we've learned, as opposed to merely being educated.

In fact, once the more abstract concepts have been learned, the student doesn't need a perfect grasp of the literal concepts that got them there. As I mentioned, I have forgotten most of the particulars of assembly programming. I will probably never have to program in C again. Yet the concept of abstraction I gleamed from learning about the relationship between C and assembly stays with me, and is all the more robust and versatile for having been formed from real and complicated examples of abstraction. This is the paradox of mastery, that often we must learn particulars to furnish the abstract, after which, that particular knowledge is of secondary importance. They are a ladder you must climb up then can push away. Yet particulars are much easier to test than abstract mastery of fundamental themes. Easier to throw some questions on bit-fiddling in the C number system than to test a student's ability to deal fluently with abstraction.

Modularity, abstraction, and interface design aren't about writing better C programs. They regard the design of systems and rules that must with humility accommodate the unforeseen. We learn them because we must make piecemeal changes to complex and evolving pieces of our world, satisfy actors who don't know each other and don't want to, allow groups with different values, divided goals, and no shared language to cooperate. We must learn to withhold judgment on the behavior of systems until we grasp their full range and underlying restraints, appreciating the reasoning of who came before us, and making sure our choices protect those who come after. These are the concepts of 217. These are the concepts students must learn if they want any hope of understanding the modern world. It is Dondero's appreciation of their importance, if nothing else, that makes him a truly great teacher worthy of imitation and study.

in C, they needed to keep track of the particular character codes on their system. They didn't, Dondero explained, because the whole point of C's character function was to "abstract" details like that away – you tell it to store an "A", and it will figure out what number to use.

VI. Bindings

Dondero's instruction succeeds or fails to the degree it binds – new knowledge to old, cause to effect, symptoms to bugs, programs to intuition, fact to explanation. We've seen the types of connections he needs to make – connecting single events between different notations, connecting chunks of code to intentional stories, integrating new information with mental models, and tying particular examples to the pervasive concepts throughout course.

What are the methods he uses to make each of these connections? Are there many? Do they depend on the type of connection being made? What are the meaningful categories of connection?

We noted earlier how every event and fact gets translated across each notation, from the programmer's intent, to the C program, to the stack diagram. These were glossily referred to as the "what", the "how", and the "why" of code. If we want to see exactly how facts are connected in producing student mastery, we will need to unpack these categories and make the distinctions more rigorous.

As was noted, C code is a compromise between the programmer and the compiler, an abstract description of "what" the program should do, leaving the details of "how" to the machine. In this sense, the code is merely a description of the "function" we wish a program to serve. Facts about the C code are functional facts, relative to the facts of implementation which regard how that functionality is carried out.[1] When we ask "How?" to a functional fact, we are asking for an explanation in terms of facts of implementation, showing how that functionality is implemented by stack operations and jump instructions.

Similarly, when we ask why a particular line of C is there at all, we are asking for an account in terms of the programmer's intent, a demonstration that the functional fact represented by the line of C serves some part of the programmer's intent, a relation of the functional fact to an intentional fact. Similarly when we ask what the stack is doing, we probably don't want a literal description in terms of implemental facts, but rather an explanation in terms of the larger intentionality being served by that the stack-level behavior – "saving the user's name to greet them with later", rather than "making a data copy of a character array".

Every fact or event in precept, therefore, is situated in a larger triangle. On one point you have a fact about the intention of the programmer and the design goals of the program. One the second, you have a functional fact, an outline of program behavior that satisfies the program's intent. On the triangle's final point, you have the "how", the set of mechanical facts about how the machine plans to carry out and implement the agreed upon functionality. For a real life

[1] The relationship between function and implementation is of course relative. It is a generalization to call all lines of C functional, because one line might be implementing the function spec defined by another. Similarly, the stack behavior is implemented by virtual memory and other lower level behavior all the way down. Yet all C is functional relative to stack behavior, and any behavior of the stack diagram is implementing some line of C. We will discuss how to model these finer distinctions later on. For now, suffice to say that when an upper-level and lower-level fact are related, we call the former a functional fact and the latter an implemental one.

example, perhaps my intention is to get to work, my functional decision is to go by car, and the implementation involves all the prerequisite choices like model or the route I will go before I can begin driving.

For illustration, let us take an example I've based on an example from the precept on pointer arithmetic[1]. In our example, a program has been written to check student attendance. It does so by taking a list of students, starting at the first one, and going down the list one by one, checking if each one is present by running the "checkAttendance" function on their student id. In program memory, there is a list of numbers, each one corresponding to the ID of one student. We have a "pointer" arrow marking where the first student's ID is, and a one marking the last student. We have a third arrow that we will use to keep track of where we are in the attendance.

Our program has three parts, three stories being simultaneously carried out involving the very same events. First there is the intentional story: "Starting with the first student, go through the list one by one and check if each one is present." This story starts when we begin the attendance, and the important events in that story are the calling of names and the advance to the next student.

Our second story is functional. It goes like this: "take a list of integers and set a pointer to the first and last. Take a third, set it to the first number in the list. Take the current number, run the "checkAttendance" function on it, then add 1 (using the ++ symbol) to the current number, moving to the next one. Stop when you get to the last number." In this story, the characters are numbers and pointers, and the important events are the calling of functions and the advancing of the pointer to the next number.

Our final story is implemental, and it goes like this: "put an integer array in on the stack. Set a pointer to the memory address labeled "FirstStudent". Repeatedly add 4 to the value of that pointer, calling the "checkAttendance" function on the dereferenced value of that pointer. Continue until the memory address labeled "CurrentStudent" is equivalent to the one labeled "LastStudent", then stop.

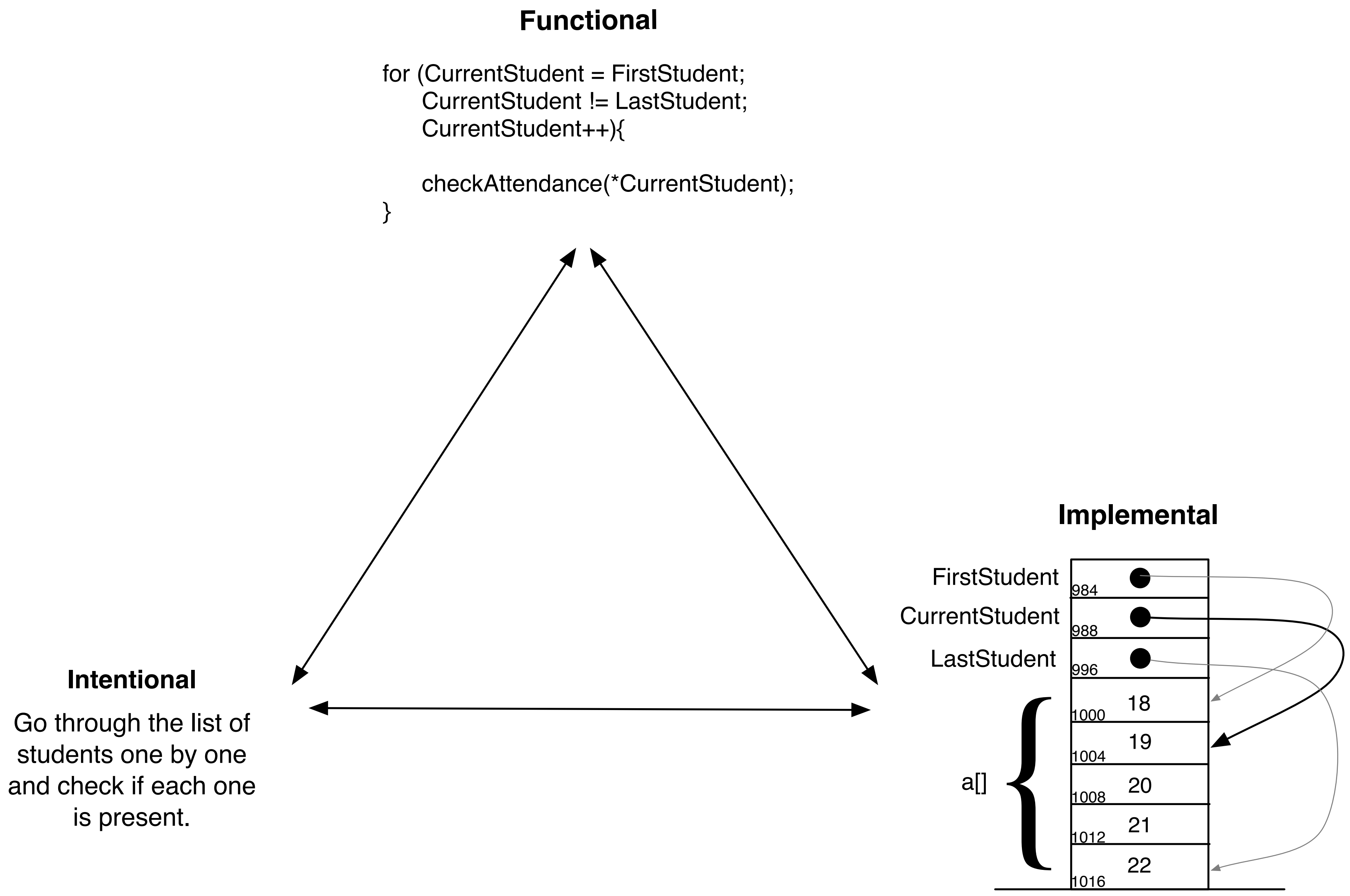
These three stories involve the same sequence of events, told alternatively in intentional, functional, and implemental vocabulary. In order to fully explain the program, we will have to relate each of these stories to each other by relating every event across stories, binding them together into one coherent whole.

In Diagram 1, we see how these three stories might be visualized as points on a triangle. In order to fully understand the program, we have to bind equivalent facts in the different notations, connecting the triangle's points.

We also want to explain facts that seem confusing or vague in one notation with a better explanation in another. If we ask a functional question like, "why is CurrentStudent pointing to the value "19"?", we get an intentional answer like "Because we're about to call for Sally, and

[1] Many details that are important in C but not four our purposes have been removed. This is not because they are unimportant, any more than in the case of assuming masses are spherical in introductory physics. The rules we determine in simple cases will be later shown to hold for more complicated examples.

Diagram 1



```
for (CurrentStudent = FirstStudent;
     CurrentStudent != LastStudent;
     CurrentStudent++){

     checkAttendance(*CurrentStudent);
}
```

Go through the list of students one by one and check if each one is present.

that's her student ID". If we ask a question like "Why is does adding 1 (with the ++ symbol) to CurrentStudent cause its value to be incremented by 4?" we can drop down to the stack diagram and answer "CurrentStudent is location of an integer in memory, in a list of integers. Integers are each 4 bytes, and so when going down that list, we jump by 4 to find the next one (just like the postman goes down the block in units of houses instead of square feet.)" By giving the variables in C descriptive intentional names, it becomes more obvious why we stop, for example, when our "CurrentStudent" arrow is pointing at the box labeled "LastStudent", instead of the more opaque "1016".

In fact, we can generalize using the triangle diagram to say that we should be connecting every fact with every other fact in both directions. For each event in the program, we need to:

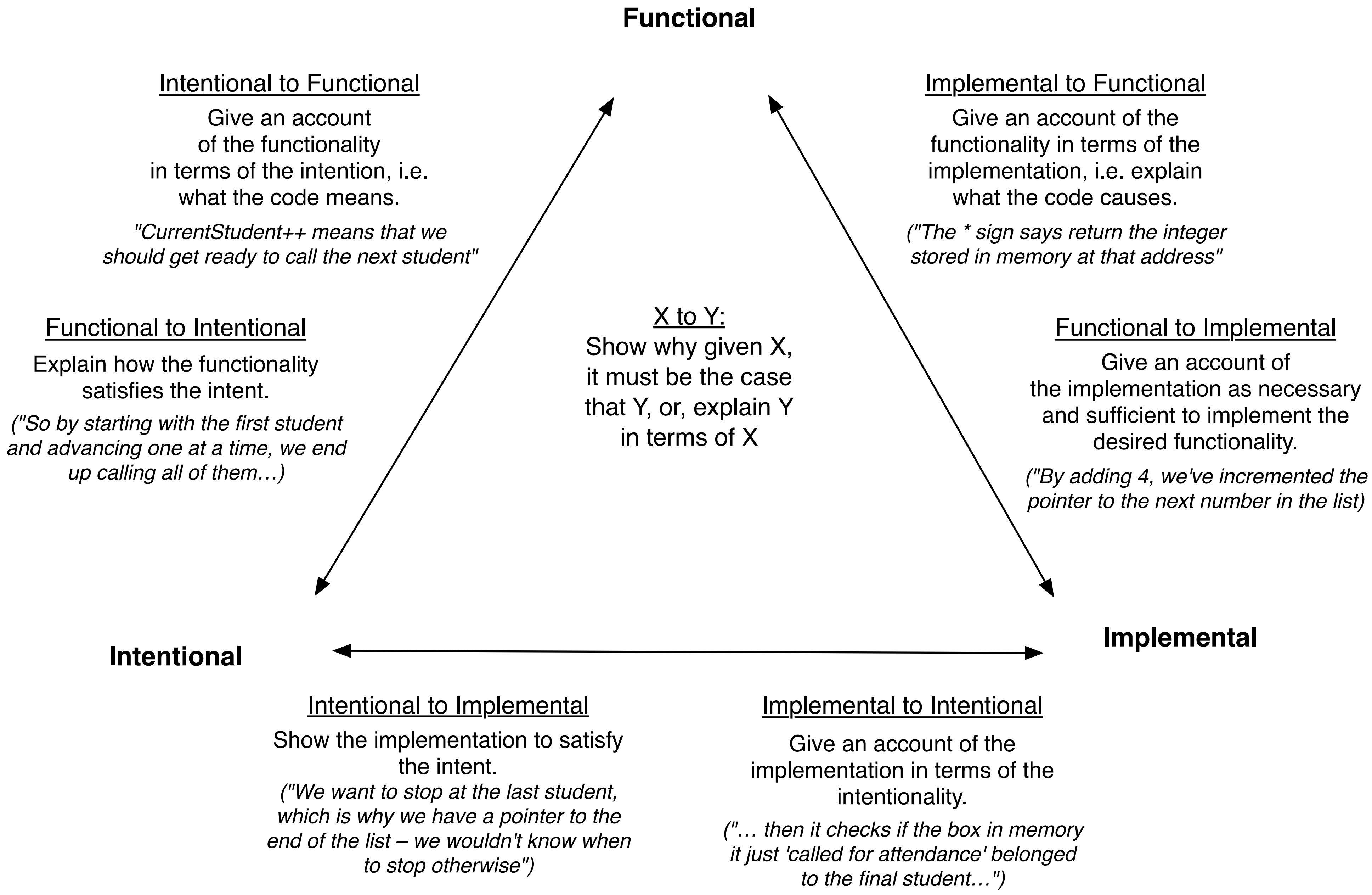
- Connect the intention to the function, explaining what the code means in terms of our intentional story.
- Connect function to intention, showing the code to satisfy our original intent.
- Connect intention to implementation, showing the implementation to satisfy our intent.
- Connect implementation back to intention, describing the parts of the implementation in terms of the purpose they play in satisfying the programmer's intent.
- Connect implementation to function, explaining what effects the code causes at the lower level.
- Connect function to implementation, demonstrating the lower level implements the functionality.

Diagram 2 lays out how this would look, with examples of something Dondero might say while explaining each of these points in our example program.

When we point at a particular line of C code and ask "why is that there?" we could be asking for an intentional response, explaining the relationship between the functionality the line describes and the part of the programmer's intent served by it, or we could be asking for an implemental account of what lower level behavior that line of C causes.

A real world analogy might be that when legislators are writing the text of a law, perhaps banning the use of particular types of firearms by civilians. Though their intent is "prevent criminals from getting dangerous guns", they cannot write this into the law, and instead must make a list or set of criteria that ban particular products from the market. Further, they must specify how they intend to enforce the law – are there penalties for violating it? Increased funding for law enforcement? In trying to understand the text of a law, we try to understand why someone with the intent of the lawmakers would write that line of the bill, why the line satisfies the intent, how the lawmakers expect the line to be carried out, can it even be carried out? Are the activities that result from the law a faithful implementation, is every resulting action, police arrests of violators for example, in line with the original intent of the law makers? These are the same sorts of questions we must answer in order to bind the intentional, functional, and implemental aspects of a program together into a single story.

Diagram 2



Functional

Intentional

Implemental

X to Y:
Show why given X, it must be the case that Y, or, explain Y in terms of X

Intentional to Functional
Give an account of the functionality in terms of the intention, i.e. what the code means.
"CurrentStudent++ means that we should get ready to call the next student"

Functional to Intentional
Explain how the functionality satisfies the intent.
("So by starting with the first student and advancing one at a time, we end up calling all of them…)

Implemental to Functional
Give an account of the functionality in terms of the implementation, i.e. explain what the code causes.
*("The * sign says return the integer stored in memory at that address"*

Functional to Implemental
Give an account of the implementation as necessary and sufficient to implement the desired functionality.
("By adding 4, we've incremented the pointer to the next number in the list)

Intentional to Implemental
Show the implementation to satisfy the intent.
("We want to stop at the last student, which is why we have a pointer to the end of the list – we wouldn't know when to stop otherwise")

Implemental to Intentional
Give an account of the implementation in terms of the intentionality.
("… then it checks if the box in memory it just 'called for attendance' belonged to the final student…")

Provisionally, if every fact about a program falls into the categories of intentional, functional, implemental, and every explanation is a relation between two facts, we can use this model to build a functional vocabulary for describing types of explanations and types of connections they produce. The types and directions of the edges of our triangle can be used to meaningfully refer to individual "connections" a student can succeed or fail in learning from a particular piece of instruction. The edge between our intentional fact "stop checking attendance when the last student has been called for" and the line "CurrentStudent != LastStudent++;" refers to the connection between those facts – an explanation of why in C and English these are saying equivalent things, an explanation like: "we are asking the for loop to continue incrementing the pointer CurrentStudent, but to stop when it is pointing to the same student as LastStudent, meaning we've reached the end of the list – this is why we created the label "LastStudent" in the first place, the cycle wouldn't know when to stop otherwise".

When two facts are related by an edge, the edge is a way of describing a particular piece of knowledge or type of explanation. If we ask "What is being explained?' we could answer, the edge between Fact A and Fact B. The edge between "Napoleon was losing the continental wars in 1803" and "Jefferson bought Louisiana for $15 million" is a category of possible explanations that contains "Because Napoleon was losing the wars, he needed money and sold America Louisiana for a bargain". This is an example of an explanation that would serve the functional role of connecting Fact A to Fact B. Any explanation that attempts to explain A: the price of Louisiana in terms of B: France's loss of the Napoleonic wars, is in the category edge(A,B).

An edge is also a way to describe the state of a student's knowledge – they can know both A and B but fail to grasp edge(A,B). The explanation that functionally counts as explaining edge(A,B) is the one that furnishes in a student's mind the understanding of edge(A,B).

On one last note, we can imagine each directed edge as being the answer to a question motivated by the point at the edge's source. The line pointing from a functional fact to an implemental fact represents the answer to the question, "How is this functionality implemented?", while the edge pointing back answers "what functionality is this implementing?" The edge pointing from functionality to intentionality answers, "Why is this functionality here?" while the edge pointing the other way answers "What part of the functionality satisfies this part of our intent?" The edge between implementation and intentionality answers "Why is the lower-level behavior doing that?" whereas the edge going the other way answers "What part of the implementation satisfies this part of the intentionality?"

Having established this framework for classifying units of student understanding, can we use it to find the meaningful chunks of instruction that instill them? I suspect that if we use use the framework edges of typed directional edges, from intentional facts to implemental facts, for example, we will find a great deal of consistency in the type of explanation Dondero uses to instill that connection. The type of the edge will be explanatory in predicting the type of explanation Dondero uses.

For the edge between intention and function, Dondero must explain the various parts of the function using intentional vocabulary. I believe he does this through intentional narration – by taking a line of code, and reading it aloud, but instead of reading it literally, translating it into the terms of the human intention behind them. When narrating the three clauses of our example's for loop, he might speak them aloud as such:

"CurrentStudent = FirstStudent;"	"Starting at the first student,"
"CurrentStudent != LastStudent;"	"and until you hit the last student"
"CurrentStudent++){"	"run the following block of code, then increment to the next student, and repeat"

In this way, the student can read both the literal instruction, and the intentional meaning behind it at the same time. This style of narration binds the program's functional and intentional stories in the same way that narration in a documentary explains the intentionality behind the practices being filmed.

We may now be in a position to shed more light on the "precept goggles" effect – if Dondero is using intentional narration to connect particular lines of code to parts of the programmer's intentionality, students might not notice that he isn't reading the code literally, integrating additional semantic information about the intentional connotations of each line which the student integrates into their reading without noticing it. This could be how Dondero gives students more information, but is easier to follow: like a mother bird, he's already pre-chewed the C, preparing the intentional semantics for consumption, so that when he feeds it to the students by voice in addition to the literal line of C being read, he isn't giving them more information to process – he's already done the processing for them. Some may, like me, be shocked to find that without his helping hand, the training wheels of his narration, it is much more difficult to stay on the bicycle. Any time Dondero is trying to connect a particular functional fact to an intentional fact, therefore, he uses the explanatory strategy of narrating intentional semantics over particular lines of code.

When Dondero is trying to instill the connection between a functional fact and its implementation, the method of explanation he uses could be called a "functional decomposition". First describe the functionality, what the behavior the line of C is specifying should be done in functional terms. Then he drops down to the relevant part of the implementation in the stack diagram. He now tells the the implementation level of the same story, the series of actions taken by the stack on behalf of the relevant line of C. Then, he turns around and shows how those lower-level actions would in combination produce that behavior specified by the C code, constituting a successful implementation. Finally, he explain why this particular implementation was chosen as opposed to any other, and what constraints to the functionality were imposed by facts about this particular implementation. In sum, he explain how the functionality specifies, determines, and shapes the implementing code, then turns around and shows why that implementation is successful. This "functional decomposition" strategy is performed whenever trying to relate a functional fact to an implemental fact.

The connection between intention and implementation, finally, I believe is furnished through teleological explanation. To return momentarily to our circulatory example, we could describe the heart as "a mesh of contracting muscle" or we could say "it's a pump for getting oxygen to the body's cells". The former is literally correct, but the latter relates the heart's behavior to its overall purpose. In 217, Dondero uses similar teleological explanations of lower-level behavior to connect facts about the implementation to facts about intentionality. By describing, for example, the "CurrentStudent" pointer as "going through the list", Dondero is ascribing to that pointer intentionality it doesn't really have – the pointer doesn't know what "next" means, and it doesn't know what it's moving towards. We know what it's doing, however, and by reminding students of it through a teleological story, Dondero doesn't have to repeat himself saying "then the pointer gets four added to it again" and can instead stick to the important parts like "the pointer keeps going one block at a time towards the LastStudent."

We saw earlier how a student can grasp a concept to varying degrees, depending on whether it is merely demarked, or furnished with prototypical examples in various contexts. We mentioned too that this distinction holds for facts about those concepts as well. We saw also how the different aspects of a concept could be segmented, or fluently bound together. The picture that should be developing is that concepts are formed from seeing patterns abstracted from related examples, affixing a word to that pattern, then creating loose assemblies of interconnected facts about the concept using that word. We've just unpacked what is meant by "interconnected", that for any two facts which are equivalent, causally related, or otherwise interdependent, there is an edge connecting them that student either has or doesn't have, which must be spelled out during instruction. I am now in a position to state generally one of the theses of this paper: that in any moment of instruction, Dondero is doing one of these things, providing instruction in one of these functional categories. He might be giving an example that he will then use to furnish several individual concepts and propositions. He might be explaining the relation between a functional and intentional fact. However if any piece of his instruction is important, it can be shown to satisfy one or more of these discrete functional purposes.

Given this framework, we may be in a position to answer another question, how is Dondero able to clear up student confusion even when the questions being asked by the students aren't clear? If there are a finite number of edges between functional, implemental, and intentional facts, there must be a finite number of holes you can have in your understanding, and in turn, a finite number of subsequent misunderstandings. Dondero seems to be able to isolate, given the phrasing of the question alone, what particular explanatory edges the student is missing, and knows immediately whether the material in his lesson intended to furnish that edge is still to come, or should be repeated now for the student.

How might the phrasing of a question betray the student's particular misunderstanding? If we recall from my example of Central Park, one of the manifestations of my ignorance of the relationship between the part and upper Manhattan is my ability to conceive of impossible occurrences – to think and speak as though walking west from a pond on 71st might lead me to 128th. Student questions, even if they ask about something completely different, can often betray particular conceptual misunderstandings. For example, a student's phrasing may betray that the student has only a single concept for two separate things – if they ask about "the" President Roosevelt, no matter their question, what may need clearing up is that there are two. Similarly, a student who speaks as though pointers and variables are separate things might not have grasped that pointers are a particular functionality, implemented by integer variables. What they need, therefore, is a rehash of the relationship between the functionality of pointers and their implementation.

If this is how Dondero so skillfully handles student questions, perhaps any student misunderstanding could be identified with some series of missing fact nodes and explanatory edges, which could in turn be furnished with some piece of instructional content that functionally traverses and fills those gaps in the student's comprehension. Perhaps we could also say with a fair degree of certainty that a particular lesson that purports to be on "pointers" couldn't possibly fully furnish a student's understanding, because it misses a particular set of important facts and edges.

This raises a final point – if this framework is correct, and edges between particular facts of different types are the meaningful categories of understanding and explanation, mightn't we be better structuring instruction and testing around it? If test questions could be modular so as to test for a particular edge between facts, perhaps we could much more precisely diagnose student confusion, and better organize explanations intended to clear up those confusions.

V. Examples

As we see in greater detail the many points Dondero strives to hit in each lesson, many of which would be passed unnoticed by a less exacting instructor – we must begin to wonder how he is able to structure them into a single piece of instruction. How does Dondero plan such multifarious lessons, ensuring that he hits every point he wants to make without jumping around and disengaging the student?

The focal point of every lesson is the example program. For each class Dondero produces a series of realistic programs that, when traced, allow him to hit every point he wants to make organically. What sorts of points are these? The day's lesson may be a concept like "Pointers". Dondero will want to provide prototypical examples of how to use pointers in C code in the context of realistic programs. This enables the student not just to use pointers in the abstract, but to recognize the situations that call for them. Given the breadth of the topic, Dondero must limit his scope to those aspects of pointers important for a first year C programmer to know. Some of those facts will be about pointers in themselves, such as, "a pointer is a variable whose value is a memory address". Some of those facts will relate pointers to prior concepts, like "array arguments are passed by pointers in C". He must finally decide which explanatory edges he wishes to furnish between those facts, explaining for example the fact that "dereferencing a pointer set to 0 is guaranteed to give a segmentation fault" with the fact that "lower numbered memory is owned by the operating system".

Why use a series of example programs at all, instead of just going through all these points one by one?

First, even if Dondero could somehow get the student to understand in working memory all the concepts and facts he is trying to teach them that day, all would be quickly forgotten unless integrated with prior knowledge. One of the key goals of Dondero's example sequences is to communicate to the student where in long-term memory the day's information should go.

Additionally, as we saw earlier, there are various ways that a student must be situated in order to fully integrate the new information being presented them. We mentioned two types of situations previously – the local sense in which a student understands and is able to absorb what is being discussed, and a macro sense, in which they understand why they're learning this material in the first place. We also mentioned that the example program itself provides a type of linear "story" that provides another type of situation. Even when a particular line of C doesn't make sense, the student understands what the program is for, and has understood all the lines in the program up to this point. Thus, there is a third kind of "sequential" situation provided by using linear example programs to structure the lesson. Chronology situates the student in a larger narrative flow throughout the lesson. In order to fully appreciate the many competing goals satisfied by Dondero's example sequences, we will need to unpack these three types of situation a bit more.

If Dondero got up at the beginning of class and said, “Pointers are what we’re learning today. Here’s an example of a dereference statement in C”, students would have no idea what he was talking about. He’d be using words that denote concepts they’ve never seen or heard of before. These students would lack all three types of situation and would be hopelessly lost for the duration of the lesson. Yet few instructors are this terrible, and this imprudence is fairly obvious.

By comparison though, what if Dondero got up and said “Pointers are a type of primitive in the C programming language. Pointers are a type of variable whose value is a memory address. Here’s how to declare them in C…” A student might be able to follow him for a bit, though never fully understanding the larger purpose for this instruction. Yet each explanation would rely directly on the last, and as soon as a student had missed a single point, they’d it impossible to reenter the lesson. That lost student would be unable to even articulate what exactly was confusing them, for lack of the vocabulary and concepts they’re missing in the first place. Additionally, if the entire lesson is delivered in only one meaningful chunk, students will either understand or miss the entire thing. And even if they somehow get to the end without missing anything, fully understanding pointers in only working memory, that knowledge would remain unintegrated with prior knowledge, and would be quickly forgotten. With only local situation, explanation following explanation, students can’t keep up with the material or integrate it with prior knowledge.

Somehow Dondero’s example programs are able to overcome these problems, giving students something concrete to hold on to and ask questions about, breaking the lesson into smaller chunks, and successfully integrating it with prior knowledge.

People like to have consistent states of knowledge, even when their worldview is woefully incomplete. Before the pointers lesson, the student’s world is consistent. Sure, they don’t know what pointers are. But they also don’t know that they don’t know what pointers are, and they don’t know of any situations that would call for pointers. They live in an oversimple but completely internally consistent world, a world in which they can write every program they can conceive of. They don’t want to hear about pointers, don’t want to hear about anything that doesn’t fit into their rosy little world – if you tell them about pointers, they will smile, paraphrase what you say, then forget it all as quickly as they possibly can.

In order for them to accept new information, that information must solve a problem for them. It must explain something that is confusing them, must provide some concrete benefit that they can appreciate. It must make their world more consistent, not less so. In order to integrate new knowledge, therefore, their world must be made temporarily inconsistent, so that what the very thing they need to restore balance is the next explanation.

How can we induce this state of temporary inconsistency that prepares the student to accept new information? In the easiest case, the student can be exposed to evidence that a belief they hold is wrong – a talking dog, a purple tree. These experiences would be highly unexpected, but would fit into the student’s current set of concepts – “Wow a talking dog, I guess dogs can talk afterall", and so on. This student needs no guidance beyond the experience, because they already can describe and integrate it into propositional knowledge themselves. This only works, however, because the new knowledge is "lateral" to current knowledge, not requiring any new concepts or categories to integrate.

A second way of producing inconsistency in a student's knowledge is to show them an event or occurrence they cannot explain. This is the opposite extreme as the last example, an experience which not only doesn't fit into a student's current vocabulary but doesn't even prompt a sensible question. Imagine we drop a man from the middle ages into Times Square. That person would not only lack an explanation for what they saw – an explanation requiring concepts like 'video screens', 'Coca-Cola', and 'confused tourists' – they wouldn't be able to form a sensible question to ask about it. They would probably just reject the experience full-stop, chalking it up to madness or intoxication, and disengadge from any further discussion. Students become similarly disengaged when a teacher tries to shove the day's lesson at them all at once, producing too startling and unassimilable a chunk of information for them to swallow, all the while expecting them to swallow it whole.

There is, however, a middle way, somewhere between dropping the children off the deep end of the pool and feeding them a restricted diet of palatable and boring foods they've seen before. In this case, the student is shown an example that is inconsistent with prior knowledge, but in one place only. Imagine if we now show our friend from the middle ages a printed book. This artifact is completely novel, unassimilable with prior concepts alone, yet continuous with prior knowledge – prompting the question: "How do the scribes get the letters to be so clear?" instead of just "Huh?" Rather than just producing puzzled looks, showing an example that is unexpected or inconsistent in one way only will prompt a well-formed question.

In this fashion, we could bring our medieval friend into the modern day step by step, doling out new information and experiences individually and chronologically, each motivating the next. When making a student's state of knowledge inconsistent, it is important to similarly do it bit by bit, lest they simply give up and become disengaged from the lesson. Just like a big meal, the entire day's lesson can't be swallowed at once, and must be broken up into bite-sized pieces. Each bite must be fully swallowed – each inconsistency fully resolved – before going onto the next one. This gives the student hope, lets them see they're making progress, puts them in the market for more information.

Yet how can one possible produce intermediary states of understanding a single piece of knowledge? Pointers are what they are – surely you either understand them or not. The basics of pointers at least must be an inseparable unit – what pointers are, what they're for, how to declare them, and so on. How can we possible break up such singular concept into individual bites, each of which leaves the student (for the time being) completely sated?

Here is where sequential situation comes in. Humans already have an explanatory technology for parceling out pieces of information, each of which must be temporarily understood only in part, none of which can be understood fully without the whole. This is the story, a chronological sequence of scenes and explanatory tangents, each event locally understandable but lacking full context until the end.

Everyone loves a mystery story, where the most important event of the tale isn't fully understood until the very last chapter. No one feels out of context in a story just because they don't yet know how it ends, so long as they remain situated: chronologically, understanding where they are in the sequence of the story, locally, understanding in each scene what is going on and what is being discussed, and finally, in the macro, understanding what the bigger picture is – not yet knowing, perhaps, who the murderer is, but knowing that its a murder mystery. If, in the middle of that mystery, the author included a long tangent about the protagonist's childhood fixation with bugs, this would break the reader's macro situation – even though they would understand who and what is being discussed (local situation) and understand that it was a flashback relative the rest of the story (sequential situation), they would be totally lost as to the significance of this new information in the context of the larger story. The reader's original macro understanding of the story might turn out to be wrong – maybe, in a twist ending, the person faked their death and was never murdered in the first place. The reader must still be given that framework for understanding the story until such time as the twist can be made. This is similar to when students can't understand the practical purpose of a topic until after they've learned it. In these cases, we must create a fictional reason, a provisional macro situation, like telling a child that math is all about arithmetic puzzles until they're ready to understand the good stuff.

When we are presented with unsituated information, in a novel or in a classroom, we often ask "situating questions". If we cannot follow a particular part of explanation or dialog, losing our local situation, we might ask "What are we talking about? What's going on?" If an instructor jumps around an example, breaking chronological lockstep, or a chapter of our novel begins with a startling flashback, we become disoriented and ask "Where are we?" When we lose sight of how what we're learning fits into the larger picture, losing our macro situation, we ask "What's the point of all this? Where's this going? What's this all for?"

Dondero avoids provoking these questions by preempting them, couching his lessons in a single narrative. Just as we are content not knowing how the story will end until later, his students are completely content in not knowing everything about pointers right away, so long as they know in the macro what the lesson is on, understand what each line in the sequence does before moving onto the next one. When an explanatory tangent is made his students understand in the local why the current line of the program has motivated this line of discussion, and know that's where the discussion will return before moving onto the next one.

Using an example program thus lets Dondero keep his hand on the valve of confusion, slowly releasing unassimilable and disjointing experiences bit by bit, building up the student's immunity instead of trouncing them with the full disease.

Each significant line of the example program represents an event or fact that is unassimilable into the student's current knowledge base. It may contain an instruction they don't yet know, or break a rule they thought couldn't be broken. The line of C motivates a string of questions, each one prompting another until the string is completed. Once every string of questions motivated by that line is answered, the student is restored to a consistent state of knowledge – they have explanations for everything they've experienced, every line of C code they've ever seen, up to that point. Only then does Dondero advance to the next line, opening a new inconstancy, filling it with new explanations, explanations which themselves motivate questions, more explanations, and more questions, until either every question is answered, or a particularly thorny one is explicitly tabled for later. Each line should motivate one well-formed question, the answer to which may motivate one more, and so on, until every one is dealt with.

Why is it important to raise single, well-formed questions? If a student can fully phrase what they're seeing, it assimilable in terms of prior concepts – "Wow, there's a talking dog", for example. On the other hand, when an experience is completely unassimilable into prior concepts, like our medieval friend in Times Square, an instructor will just get confused looks. In between, when an experience cannot be fully phrased into prior vocabulary, sometimes it can still be phrased into a well-formed question, by leaving out the word or concept the student lacks. If I tell someone that "Abraham Lincoln freed the American slaves" and they are able to respond with the well-formed question, "Who is Abraham Lincoln?", they've at least put their finger on what information they need in order to make the statement sensible. They understand the rest of the sentence, know what they're missing, and can use the part they don't understand to articulate a question asking for that information. If on the other hand my interlocutor didn't know what slaves or Lincoln were, my statement would provoke only a confused "what?" This is not a well-formed question for a particular piece of information but rather a request for something the asker can't fully describe. Raising questions like these is pointless, as in doing so, you will just have to go back and explain the simpler concepts to being with, like slavery, which you could have just explained in the first place.

On the several occasions I asked Dondero how he plans his lessons, he told me, "I try to figure out what they know, what I want them to know, and build a bridge." Dondero often uses a metaphor of a tree to explain integrating new information with prior knowledge. Only after closer inspection of Dondero's example sequences did I began to understand what he meant.

Centering the lesson around an example programs is an effective way to situate and break up large chunks of information. This explains the functional purpose of using an example program, but why the sequence of example programs Dondero invariably uses? How does Dondero integrate new knowledge with prior knowledge? What does "integrating with prior knowledge" even mean?

We use our prior knowledge to navigate the world around us. We get into our car, and access our knowledge of driving. We see a bear running for us, and access our knowledge of escaping.

To be useful, new knowledge must be integrated with old so that it comes to mind when prompted – no point in remembering how you ride a bicycle when you get in a car, or having your driving knowledge stored in two separate places to be accessed separately when needed at once. I might hear on the evening news that my morning commute to work is blocked by construction, yet tomorrow morning I may still wake up, brush my teeth, get dressed, and drive straight into the traffic. It's not that I didn't understand what the news said, its that I didn't integrate it with my prior knowledge, the subroutine of habit that I initiate when prompted with the cues of a weekday morning.

When a war reporter sends home dispatches about particular battles and political developments, the folks back home are wondering "Does this mean we're winning or losing?" When teaching middle schoolers germ theory, the unit should end with a reminder, "this is why it is important to wash your hands!" If new information is actionable, it must be integrated with knowledge of the situations that call for the action. If new information relates to prior beliefs we hold or questions we have, it must be integrated directly with those questions and beliefs if we want it triggered on their mention. What is the point of learning something new if no sequence of events or cues can possibly bring it to mind? To borrow from 217 parlance, that information would be "garbage", knowledge in memory inaccessible by any reference.

Some part of the purpose for Dondero's example sequences therefore is to indicate to the student which of their former beliefs or behaviors should be altered by the coming information. That prior knowledge must be brought to working memory, altered by a series of examples that produce the new concept, save the altered version to memory, and solidify it with practice to rewrite over old habits.

Let's take, for instance, the sequence of examples from the precept on "Dynamic Memory Management". Literally, it is a lesson on the family of "Malloc" functions. Instead of giving an example sequence, we could imagine Dondero just giving one program of repeated calls to the malloc function, showing how literally the function is used. Students could probably follow such an artificial program, as they know what sort of thing a function is, and might even understand the sorts of situations that call for malloc. Yet this student, when presented with a need to declare an array based on variable user input, would simply try to declare an array with runtime input, leading to a undescriptive error and general frustration at the language designer. This sequence of events must be be headed off, preempted by forcing the student to see the situations they might need malloc *first*, and only then giving them that knowledge. This serves two in the same purpose: it makes their prior knowledge inconsistent before offering the information that will restore balance, by showing them the insufficiency of stack resident storage before showing them the alternative, and second, it overwrites their reaction to programming prompts involving arrays allocated based on the size of user input, leading such situations to bring to mind the new knowledge. By calling prior knowledge to working memory first with an example, new knowledge is directly integrated into that prior knowledge as soon as it is understood, and the two are saved together as one unit.

The first example in the dynamic memory precept doesn't involve dynamic memory at all, rather showing students an example of a program that takes a list of 10 numbers from the user and sorts them by size. In fully understanding the program, the student recalls their "decision tree" regarding memory allocation: Is there one value? Use a variable. Are there many? Use an array. They find this knowledge perfectly sufficient for the goals of the first example, and understand it completely. Diagram 3 illustrates how Example 1 opens a hole in the student's knowledge that can be filled by dynamic memory management.

By choosing a program that sorts user defined input, the conclusion of the first example allows Dondero to naturally transition to the next, by pointing out how it would really be much nicer if the program could take any sized input, a nice example of abstraction. The next program is the sort of program the student given their current knowledge would write and expect to work. The program, instead, fails. The student is shocked. Their world has become inconsistent in a meaningful way – they can grasp why taking unspecified input is important, and they really thought they knew how to do it. By giving the second example, Dondero has anticipated a common misunderstanding and taken preventative measures, directly inoculating students against making it. A short window has opened where they both want an answer to restore cognitive equilibrium, and would be in a position to integrate that answer into their prior knowledge.

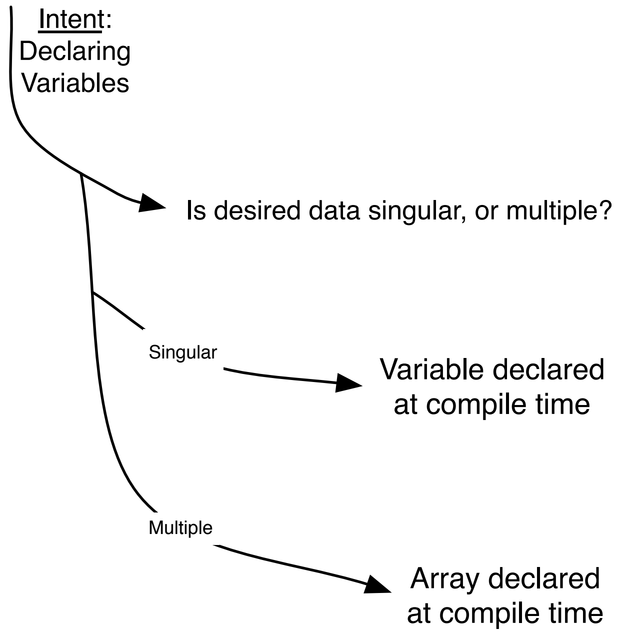
Dondero takes the opportunity with example 3, a very simple program involving the same program specification, but this time using the malloc function. For every fact Dondero wants the student to know about malloc, there is a line in the program that furnishes it, and allows Dondero to explain using an explanatory tangent moored to that particular line. The student can understand Example 3, is satisfied, and by the end of the program is restored to a consistent state of knowledge.

While the series of examples is still in working memory, Dondero has another window of opportunity. He has spent so long developing this example sequence that the student is well versed in the various parts and design goals of the program. He can now take the opportunity to incorporate pervasive or normative concepts, showing what would make the program more modular, or more abstract, which in this case involves removing the redundant user specification of the input size. "See how redundant specification of a constant opens you up for inconsistency? See how avoidable it is? See how we make the program more modular by writing one reusable function that adapts to the input as it comes?", he might say, tying what is modular in this case to the larger principles of modularity in general.

Thus Dondero's sequence of examples allows him to hit every point he wants to make organically, all while integrating with prior knowledge, overwriting prior beliefs and habits with new ones, and giving him a chance to tie the whole thing into the pervasive concepts of modularity and abstraction.

Integrating New Knowledge

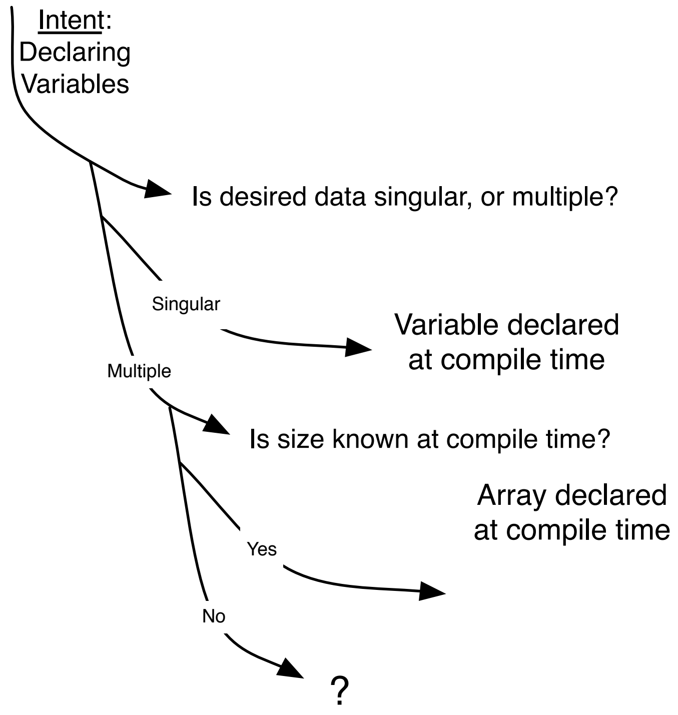
Diagram 3



Intent:
Declaring Variables
Is desired data singular, or multiple?
Singular
Variable declared at compile time
Multiple
Array declared at compile time

Dynamic Mem Management Ex1

Brings relevant prior knowledge to working memory



Intent:
Declaring Variables
Is desired data singular, or multiple?
Singular
Variable declared at compile time
Multiple
Is size known at compile time?
Yes
Array declared at compile time
No
?

Dynamic Mem Management Ex2

Unexpected result motivates question: "How does one allocate an array if size is unknown at compile time?"

Dondero 10/8 Dynamic Memory Precept:

"The size of a stack resident array must be specified at compile time. If you're going to have an array on the stack, you're going to have to tell the compiler how big it's going to be. Which implies that if you need an array that is unknown at compile time, it can't be on the stack *its gotta be someplace else*"

In our discussions, Dondero called these sorts of example sequences “123” examples, because you could start with prior knowledge (1), tell a story with a series of examples, forming the bridge (2), then arrive at the new concept (3) organically.

We might generalize the creation of example sequences for “123” cases as follows. First, given the concept you want to instill, what is a question fully phrasable in the student’s current vocabulary for which the answer is that concept? In the dynamic memory management example, it was, “How do I declare an array whose size depends on the size of user input specified at run-time? For Multi-File C programs it was, “How can I define my functions in whatever order I want?” These questions provide a fictional macro situation. Just like the mystery novel with a twist ending must still establish itself as such to temporarily situate the reader, students learning a new concept cannot understand the true goal of the lesson in their current vocabulary, and thus need a fictional purpose to frame the example sequence until they gain the concept and see its wider purpose.

Once you have the question in mind, produce a first example that motivates the question but does not involve the concept – for malloc, one that involves solving the stack resident array problem by arbitrarily capping maximum input size. The first example should have a glaring design flaw that motivates the concept, like an arbitrary cap on input size – if the concept is really so useful, it shouldn’t be difficult to come up with a true limitation that a technology like dynamic memory was intended to fix in the first place. Perhaps at this point the student is sufficiently convinced they don’t know how to improve the program given their current state of knowledge, and perhaps they need to be shown how little they know, by showing that the solution they would produce wouldn’t work, as Dondero does in dynamic memory management example 2. Now, they are ready for the next example, which solves the problem with a naïve implementation using the concept. Finally, we tie the example into overarching themes like abstraction, modularity, and defensive programming by improving it along those lines with a fourth “industry-strength” example. New concepts are thus situated in the context of prior knowledge, and left integrated and ready for application.

What happens, however, when there is no way to integrate new knowledge with old, no motivating question that can be phrased in terms of prior concepts? When I was considering the pointer precept, one analogy that kept coming to my mind was how I would explain to a child that a rabbit was made of atoms. Rabbits are conceptually simpler than atoms, we learn them first, as singlular rabbit units. Similarly, we are presented with a unique problem when trying to explain concepts that are lower-level than current knowledge, as opposed to lateral it.

We established that for Dondero’s “123” method to work, we need to be able to phrase a well-formed question whose answer is the new concept. Yet this wont work in cases of “functional decomposition”, where the new concept is needed to understand prior knowledge, because students who understand a concept only in high level abstraction don’t themselves realize their knowledge is incomplete. In the case of rabbits, we cannot segue by asking “What are rabbits made of?" because technically the child doesn't yet understand what we mean by "made of" – rabbits to them are one indivisible unit. Second, we'd have to get them all the way through organic chemistry before they were ready to understand *how* the rabbit is made of atoms, to appreciate the rabbit at two levels of abstraction – the organism level, and the physical level. This chasm is too broad for the 123 method, where the bridge, "2" must be built before prior knowledge, "1", leaves working memory.

This, I think, is the problem that Dondero faces when trying to teach a concept like pointers or assembly. He cannot use the "123" trick we previously discussed, prompting the student with an example fully understandable in terms of prior concepts, yet which is suddenly made insufficient by a new example – people got along just fine before atom theory, they just lived in a world where there were more types of stuff. It is exceptionally difficult, therefore, to make a student understand what information they're missing when that information is a lower level explanation of prior knowledge.

These we might call "132" examples, where the new concept must be fully explained, then the old concept recalled, only finally integrating the two. The child must learn about rabbits, then about atoms, molecules, and so on. Only finally are they in a position for a physical explanation of the rabbit. Similarly, Dondero is forced to explain pointers on their own out of context before introducing examples that make them useful. To the student there are no obvious advantages to thinking about arrays in pointer notation, just like there are no obvious advantages to the child in understanding what rabbits are made of – all their beliefs about rabbits, "rabbits hop" for example, will still be true once they learn organic chemistry, they'll just appreciate behaviors like "hopping" in a more fundamentally mechanical sense. Try explaining the benefit of that to a four year old. It is much harder, therefore, to invent a fictional macro situation for a "132" case, a temporary provisional problem we can pretend to be solving while actually furnishing the larger concept. We must instead self consciously proffer the concept whole and contextless before giving the student even a provisional reason for wanting it. Diagrams 4 and 5 illustrate the difference between "123" and "132" cases.

We mentioned earlier that this style of explanation, "functional decomposition", i.e., explaining the functional behavior of a thing in terms of its implementation, requires the student to fully understand the lower-level story before integrating it with the upper level behavior. This is why they must understand what assembly instructions are in isolation before going back and showing how they implement various functionalities of C. Yet this presents Dondero with a problem. How does he justify to the student an unsituated diversion into seemingly unimportant C functionality like pointers? – Quickly, and with many apologies.

These sorts of intermediary briefings break the student's macro situation, meaning the student will have to keep the entire pointers unit in working memory until it can be transitioned into the long-term memory with examples. Really that wont occur until the string assignment, which forced them to integrate pointer solutions with their understanding of arrays by literally solving the same problems twice, once in each style.

Dondero's Tree ("1,2,3" Case)

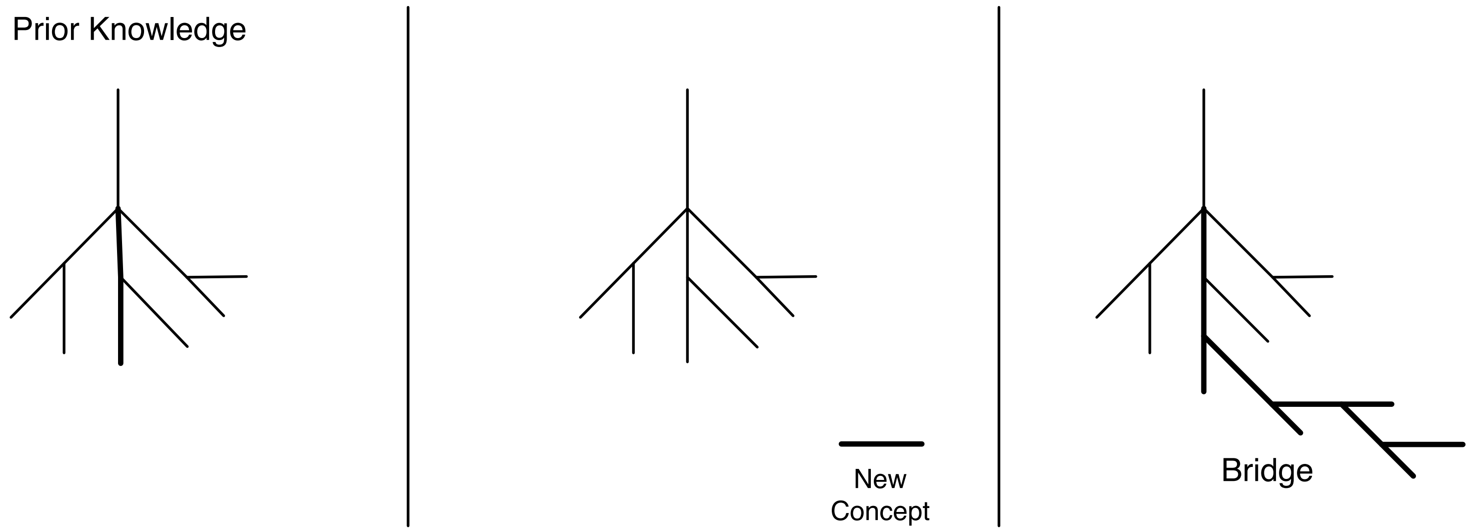
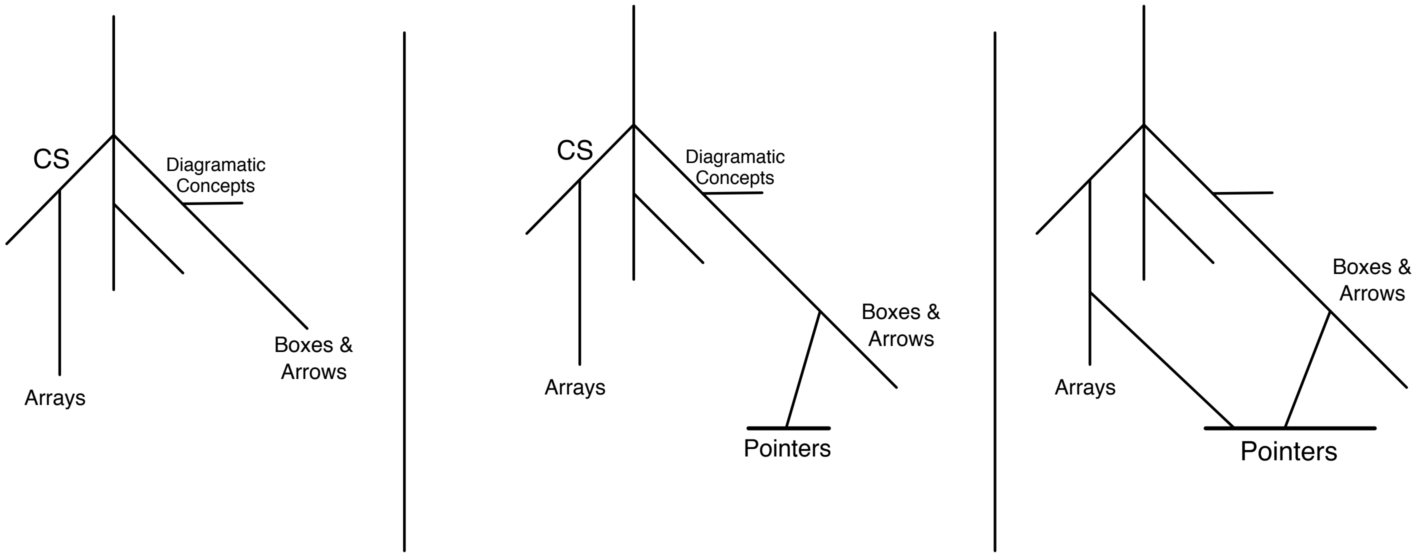


Diagram 5

Dondero's Tree ("1,3,2" Case)



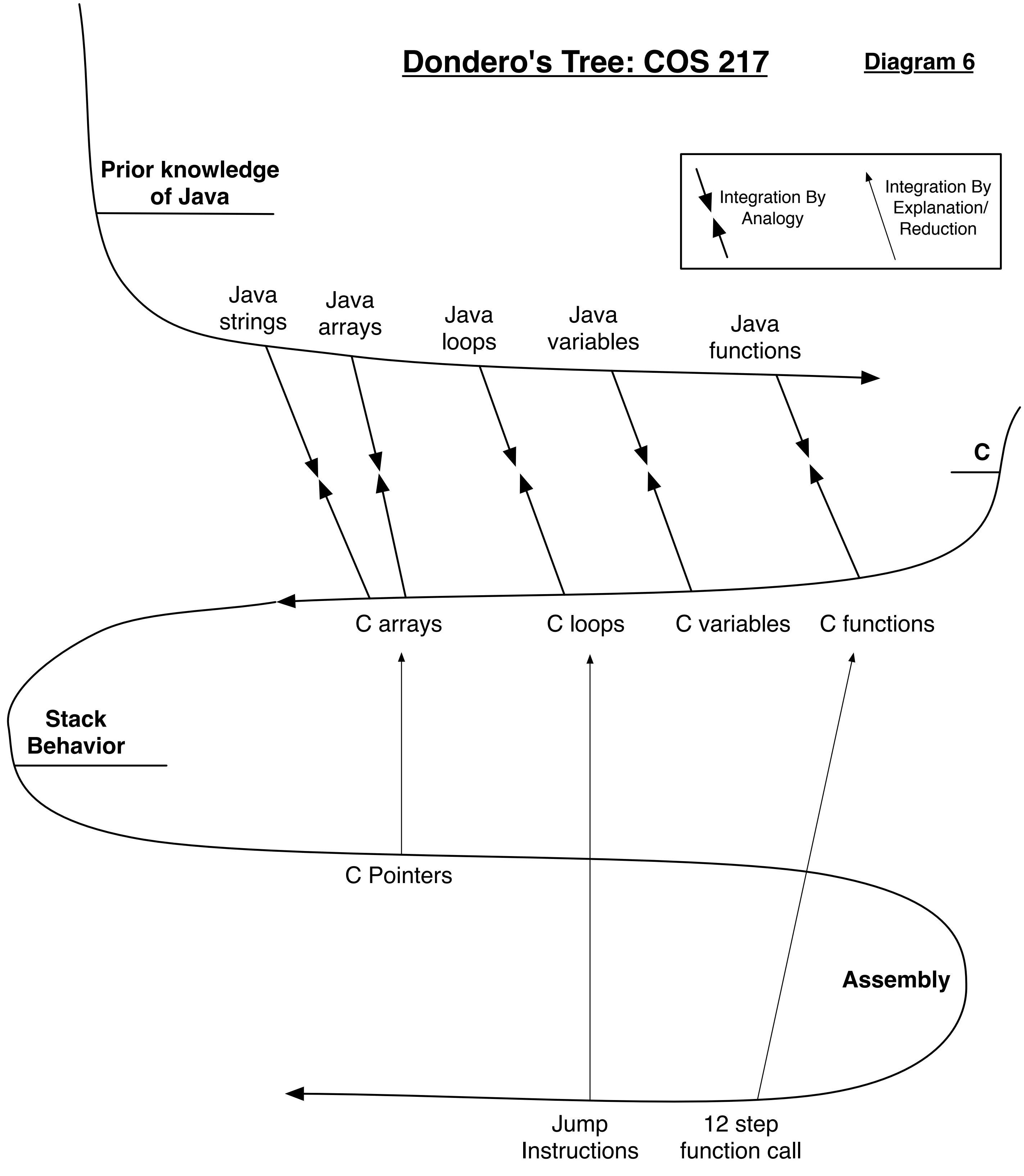
Another point to note is that because new information about pointers cannot immediately be situated into an example, the ptrs.c example program must literally hit every fact Dondero wants to make exhaustively, one-per-line, instead of organically producing examples he can use to explain the facts with tangential asides – a method he uses in dynamic memory management, for example, to use all of example 2 as material to explain and furnish the proposition “stack resident arrays must be declared at compile time”.

This analysis also puts us in a position to better understand the structure of the course as a whole. Diagram 6 illustrates how we can express the entire material of 217 as being integrated first with prior knowledge from COS 126, then integrating the more advanced topics later the course with earlier material.

Dondero’s method for constructing example sequences is the foundation of his instruction’s success. His sequences must simultaneously serve several goals. They must bring to mind all prior knowledge that should be altered by the day’s lesson. They should provide the student with all three types of situation when encountering new information; they should be situated in the macro, provided with a provisional explanation for why they’re learning the day’s concept, explainable in terms of prior vocabulary; they should be situated chronologically, with all explanation moored to a particular point in the current program, and with no movements out of lockstep in the chronology; they must finally be situated in the local, understanding why a particular string of questions has been motivated by a particular line of the program, and having that line of questions fully resolved before moving on to the next line of the program. The example sequence must also tie the current program to the larger goals of the course, showing how a minimally sufficient program might be made more abstract or modular.

Dondero's Tree: COS 217

Diagram 6



VI. Delineation

We finish the previous chapter with a greater appreciation for why Dondero centers his lessons around example sequences. They situate the student in a story that will organically give Dondero the chance to explain the day's topic bit by bit, instead of dumping it on the student all at once. And all the while, the new information is being integrated with prior knowledge, overwriting previous beliefs and habits. When Dondero writes an example sequence, he is throwing a football to the endzone. He's setting himself up for the touchdown. It is when he actually begins delivering the lesson that he catches the ball and makes it to the end zone. We can see that Dondero's example sequences set him up to hit every point he wants to make – what exactly do we mean by "hitting" each point, and how does he use the example sequence to do it?

We discussed earlier how the knowledge Dondero wishes to impart is a densely connected web of concepts, facts, and interconnections between those facts. We saw how every type of strand on that web, every functional category of knowledge had a matching functional category of explanation used to produce knowledge of that type. We saw for example, that certain types of intentional narration might be the type of explanation that can furnish the edge between between an intentional fact and a functional fact, or how a concept like dangling pointers might be furnished by seeing several examples of them.

We previously described the facts of the course as triangle, uniting functional, intentional, and implemental facts. Now we are in a position to see that this was a simplification, as every line of C or behavior of the stack is both functional relative some implementation, and implementation relative some function. We can now see therefore that the knowledge Dondero wishes to impart is no triangle, but a densely connected web of concepts furnished by example, facts relating those concepts, and a dense graph of intentional, functional, and implemental edges relating facts at different levels of abstraction. Diagram 7 shows how the function implementation distinction can recurse arbitrarily downward. That graph often doubles back on itself, for example, with function calls explained as a feature of C one day, then a complex composite of stack and register behaviors later.

Yet despite the dense interconnections of this web, explanation must be linear, for a student can only listen one word at a time. We have a rough understanding of how we would instill any individual piece of the web. The natural question to ask next, therefore, is how we pick the order in which we explain those pieces, how do we take that web of desired understanding and delineate it – literally make it linear – into a series of individual units of explanation – so that at every intermediary step the student feels situated in the flow of the lesson?

We're already on our way to answering that question. We can see that rather than just asking and answering abstract questions with abstract answers and disjointed examples, it makes sense to introduce sequential situation, an intermediary frame of reference that is inspired by the macro goals of the lesson and can serve as a launching point for local explanatory excursions.

Diagram 7

Level N

Intentional Story

I want it to tell the user: "String"

Functional Story

```
//prints "String" to
//standard output
void print_string();
```

Implementation Story

```
char s[7] = "String";
char * pointer = s;
while (*pointer != null){
    print(*pointer);
    pointer++;
}
```

Level N-1

Intentional Story

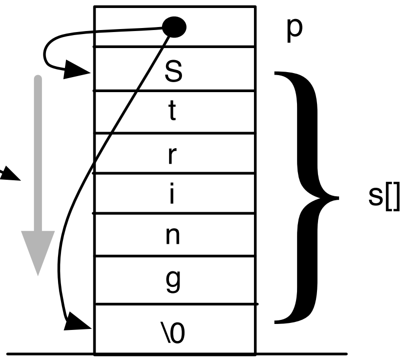
I want to print a message, so I'll store it in memory, then go through the message one letter at a time and print each one to the user's screen until I hit the end of the string, and then I'll stop.

Functional Story

```
char s[7] = "String";
char * pointer = s;
while (*pointer != null){
    print(*pointer);
    pointer++;
}
```

"Put a seven char string on the stack. Declare a pointer and set it to the first character. Until that pointer dereferences to null, dereference it and print the result, then increment.

Implementation Story



Functional Story

Implementation Story

…Assembly...

But how exactly do these explanatory excursions work? When Dondero puts a line of C in an example program, how does it set him up to motivate a string of questions during the lesson? Which sorts of explanatory tangents are prompted by which sorts of examples? How do we know when to finish the current discussion and move on to the next line in the program?

We established that the sequential situation of the example program allows Dondero to release unassimilable information bit by bit. One way to tell whether it was the right sized bit was to see if the student could phrase a motivated question about it in their prior vocabulary indicating the specific part of the example that confuses them – for example, when our medieval friend was able to ask why our scrolls were so well printed. If the bit of new information is too big, the student will only be able to ask a poorly-formed question like "What?" or "Can you explain that?", or a situating question, like "Where are we?" – questions that do not progress the lesson towards any of its desired goals.

By breaking up new knowledge into different lines of a program, Dondero can explain different parts of a concept bit by bit. But how does he know whether a single line will break off more than the student can chew in a explanation?

Dondero manages this complexity with "strings of questions", when a single line of code produces one well-formed question, the answer to which is satisfactory, but which motivates a second well-formed question, and so on until the string is concluded with a terminally satisfactory answer, or an explicit putting-aside of that ultimate question until later. A particular line might raise the question, "what's malloc?" to which the answer will prompt the question, "well, why would we want that?". Just like when playing the "Why" game with a toddler, at some point you run out of explanations and must terminate the recursive call.

There are different types of "strings of questions", each serving a different purpose. In the case we just mentioned, sometimes a particular line of C must be explained with prior concepts. Strings of questions are how Dondero relates the particular line of C to a general fact, and relate that general fact to prior facts. He also uses strings of questions to motivate a discussion of how a particular program might be made more abstract or modular. Perhaps he sees a line of the program validating input, and this sets him up for a tangent on defensive programming. Perhaps a function takes an array argument, and he can finally explain what "call by reference" really means in C. Strings of questions let him organically move the discussion from a particular line of the program into more abstract and thematic territory, before returning to the next line.

Let's look at one such example. In the precept on pointer arithmetic, the student is told that pointer arithmetic is similar to integer arithmetic. This sets them up for a predictable mistaken inference, that pointer arithmetic is in units of "1" as well. Rather than just telling the student the real facts of the matter, Dondero skillfully embeds that abstract fact into a line of C that raises a string of questions, each motivating the next, organically leading him to the general explanation.

First, the student sees that the line "Pi0 + 1 == Pi1" makes it true that 1000 + 1 == 1004, which motivates the question, "How can it be that adding one is equal to adding four?" The answer is that the 1 is being multiplied by 4 automatically. This prompts the question, “Why is it being multiplied by 4?”, to which the answer is, “because that’s how big an integer is in memory, and these are integer pointers.” This motivates the question, “Why does it matter that they’re integer pointers?” to which the answer is that there is an invisible “sizeof(type)” operator that the “1” gets multiplied by, so the type of the pointer determines the size of the multiplier. This functional fact motivates the intentional question, “Why does it do any multiplication in the first place?” Asking “Why?” to a functional fact requires an intentional fact, in this case combined with an implemental fact. Here answer is, “because we want to use pointers to refer to individual elements of arrays” (intentional fact), and “arrays are only sensibly referred to in units of the primitive size, as you can see from these box-and-arrow diagrams ” (implemental fact). Here an absolute intent combines with an implemental restriction to produce a relative or instrumental intent, an intent we only have because it helps us get the larger one.

Thus, instead of dropping all of that knowledge on the student in one explanation, Dondero’s series of motivated questions slowly unfolds, each answer bringing the student a bit closer to a consistent state of knowledge, and each answer giving the student a clear next question they can phrase using prior vocabulary.

Using this method, Dondero never has to ask a student to take a particular fact as brute, true without reason, but instead is constantly weaving connections of causality and necessity, tying low level facts back to design intents and systemic restraints. Dondero walks through the program, and uses each line to propel him into realms of more abstract facts and concepts, explaining the current line with a prior fact, or typing a particular programming technique back to the principles of modularity. This behavior can be generalized by stating that when a string of questions unearths a new fact, the answer to the next motivated question is meant to “buttress” that fact, for example by showing it to follow from prior beliefs, or explaining it with a still higher fact or motivation. Diagram 8 shows a model of how this recursion might work. Diagram 9 shows how the flow of an example like Dynamic Memory management might be modeled.

How then can we explain Dondero’s asides? More than any other instructor, Dondero’s lessons are filled with tangential asides that seem unrelated to the rest of instruction. I believe they serve a functional purpose, and using the recursive question model we are finally in a position to explain them. Dondero’s asides make functional sense if you assume that every statement of fact should be followed with an explanation of why that fact is the case and not otherwise. Sometimes, the best explanation is a more fundamental fact. Other times, the best explanation is a cautionary tale from industry illustrating why we avoid memory leaks. This further explanation serves to buttress the new information, just as my belief that function calls aren’t free was buttressed by, and out survived, my knowledge of the explanation in assembly.

When do asides occur? At some point in a recursive string of questions, Dondero is finally able to present a single general fact, pointer arithmetic behavior as a coherent whole, for example. When a string of questions brings Dondero to a new fact that cannot be further

Diagram 8A

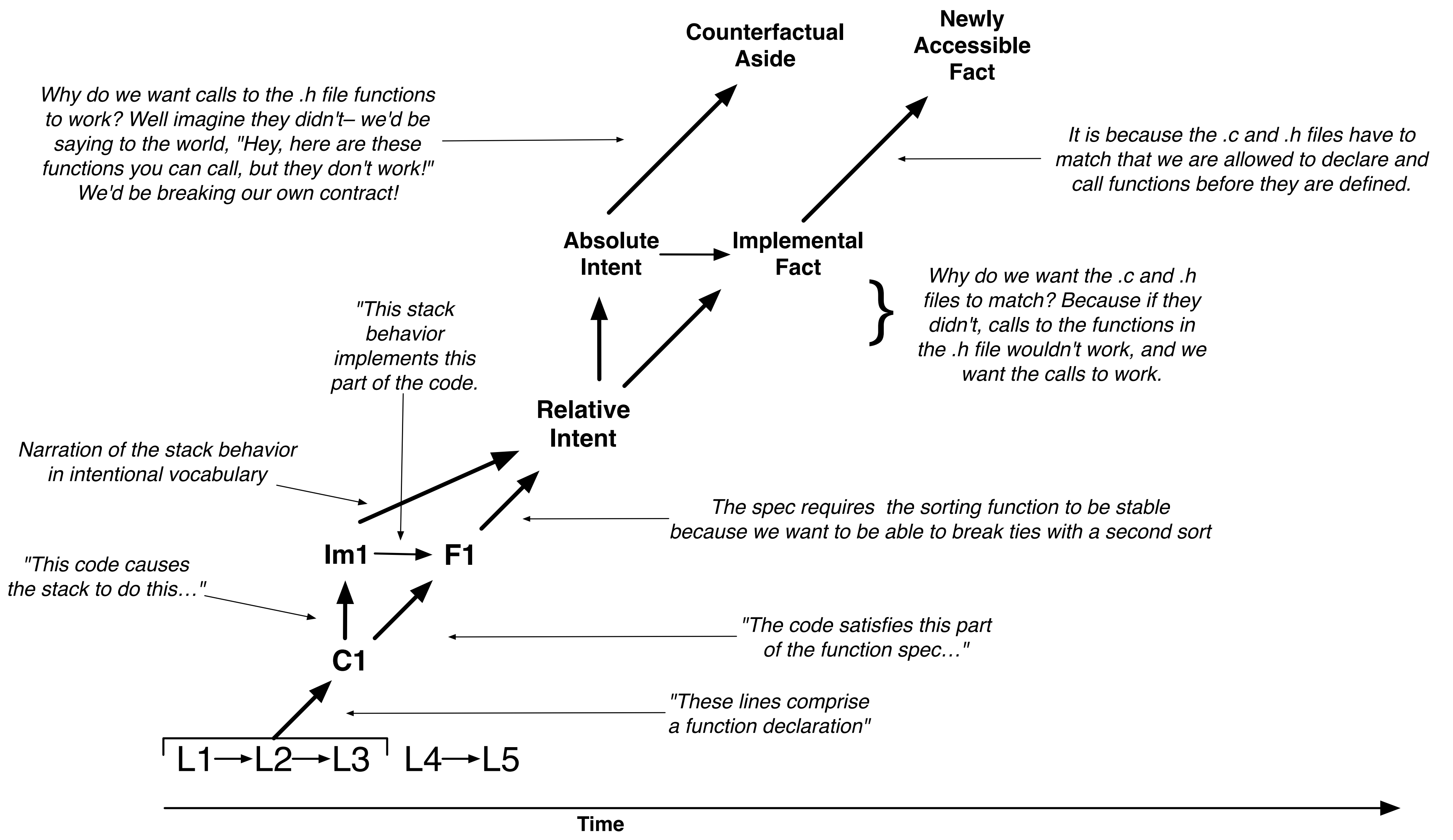

Counterfactual Aside
Newly Accessible Fact
Why do we want calls to the .h file functions to work? Well imagine they didn't– we'd be saying to the world, "Hey, here are these functions you can call, but they don't work!" We'd be breaking our own contract!
It is because the .c and .h files have to match that we are allowed to declare and call functions before they are defined.
Absolute Intent
Implemental Fact
Why do we want the .c and .h files to match? Because if they didn't, calls to the functions in the .h file wouldn't work, and we want the calls to work.
"This stack behavior implements this part of the code.
Relative Intent
Narration of the stack behavior in intentional vocabulary
The spec requires the sorting function to be stable because we want to be able to break ties with a second sort
Im1
F1
"This code causes the stack to do this…"
"The code satisfies this part of the function spec…"
C1
"These lines comprise a function declaration"
L1 L2 L3 L4 L5
Time

Diagram 8B

- L1, L2, L3 – "What are those lines?" (Implemental Concept + Demonstration of Membership)
  - C1 – "What does that mean?" – (Functional Fact + Demonstration of Implementation) + (Intentional fact
    - F1 – "Why does it do that?" – (Relative/Instrumental Intent + Demonstration of Satisfaction)
      - RI – "Why do we want that? – (Absolute Intent + Implemental Fact + Demonstration of Inference)
        - AI – "Why do we want that?"
          - Counterfactual Aside as though ¬AI
        - I1 – "What significance does that have?"
          - Newly Accessible Fact from I1
        - Demonstration that AI + I1 = RI
      - Demonstration that F1 satisfies RI
    - Demonstration C1 implements F1 (Functional Decomposition)
  - Demonstration L1, L2, L3 form a C1
- Stack Empty, advance to L4
- Stack Empty, advance to L5

Diagram 9A

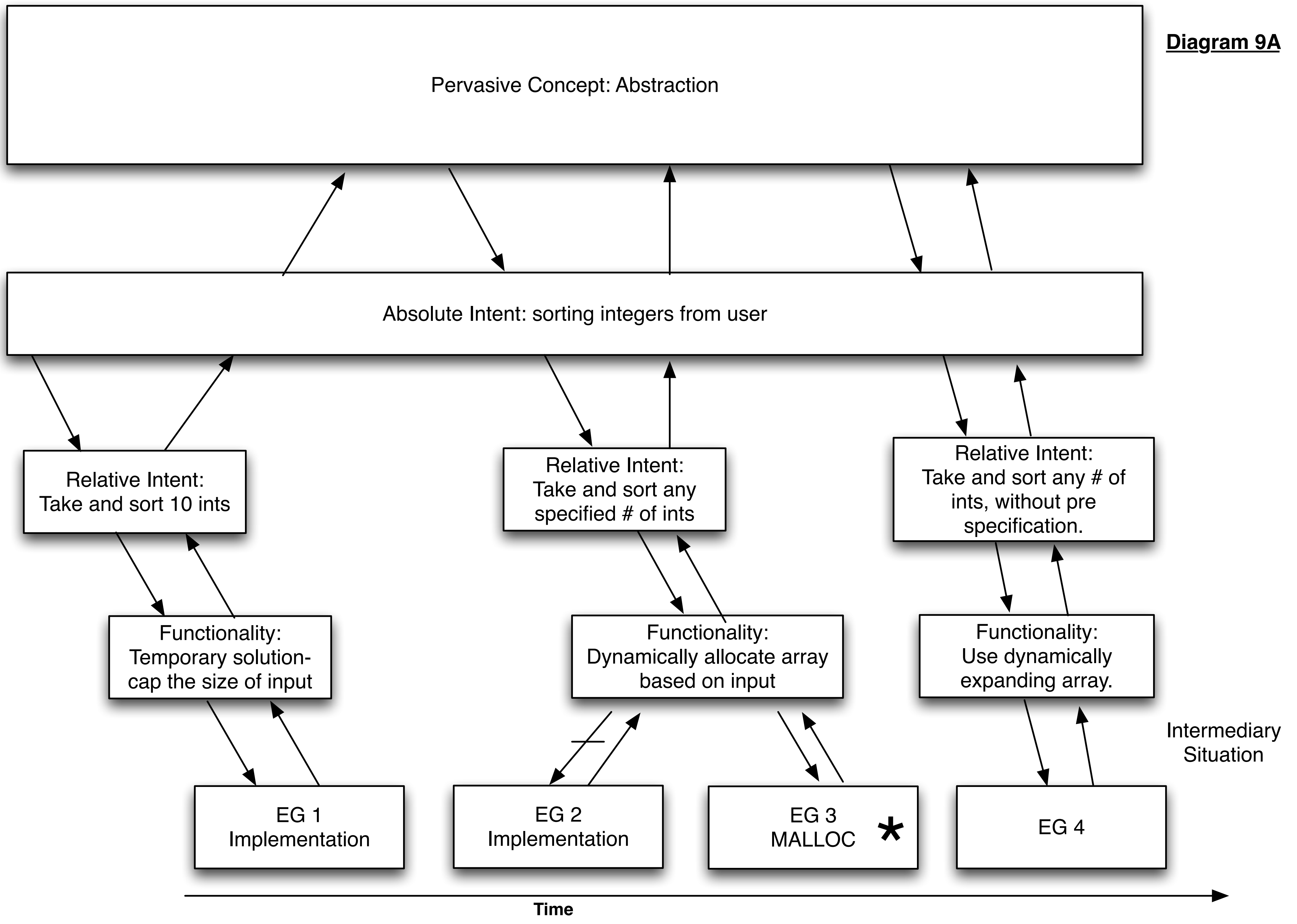
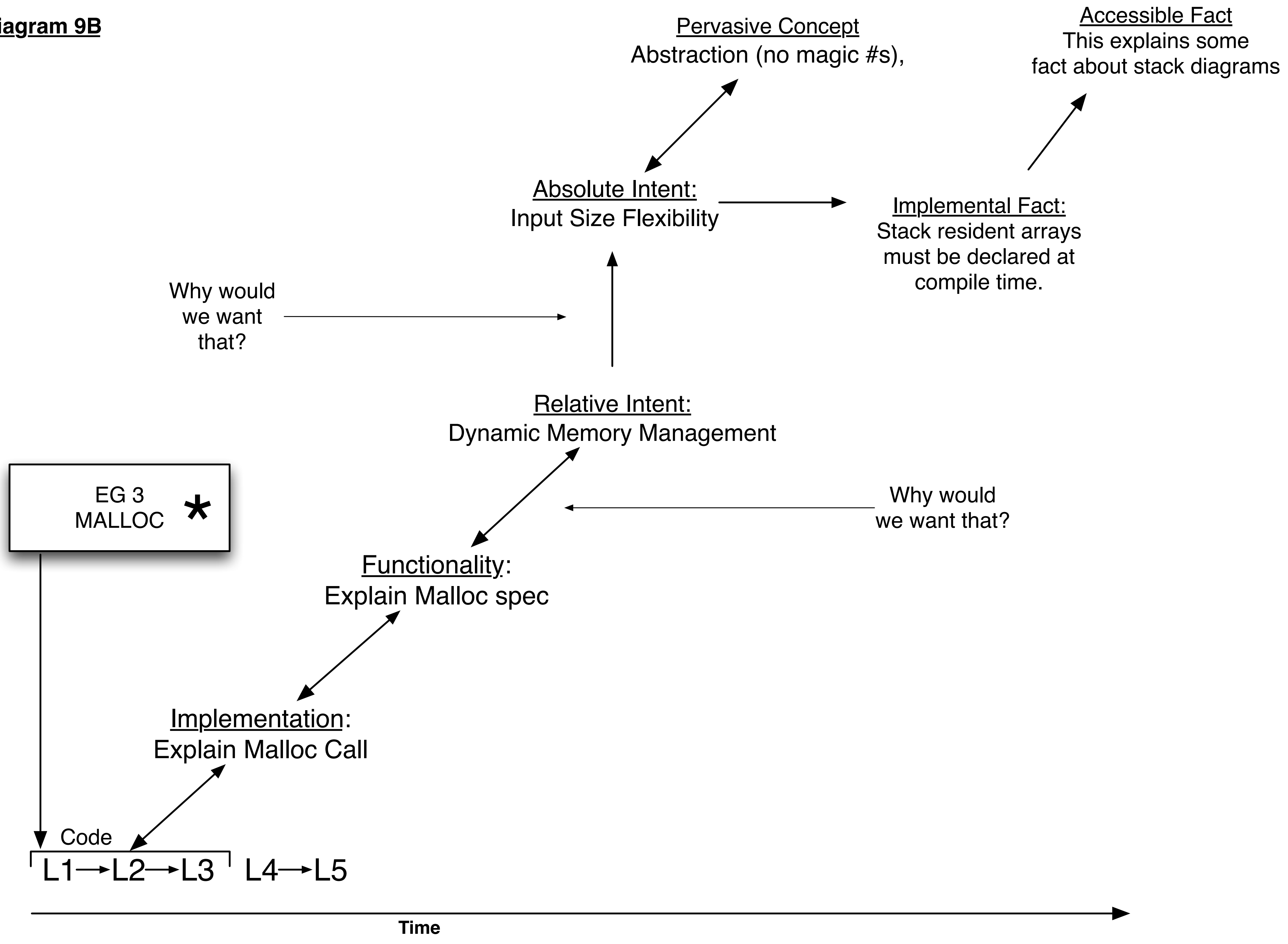

Pervasive Concept: Abstraction
Absolute Intent: sorting integers from user
Relative Intent: Take and sort 10 ints
Relative Intent: Take and sort any specified # of ints
Relative Intent: Take and sort any # of ints, without pre specification.
Functionality: Temporary solution- cap the size of input
Functionality: Dynamically allocate array based on input
Functionality: Use dynamically expanding array.
Intermediary Situation
EG 1 Implementation
EG 2 Implementation
EG 3 MALLOC *
EG 4
Time

Diagram 9B



Pervasive Concept
Abstraction (no magic #s),
Accessible Fact
This explains some fact about stack diagrams
Absolute Intent:
Input Size Flexibility
Implemental Fact:
Stack resident arrays must be declared at compile time.
Why would we want that?
Relative Intent:
Dynamic Memory Management
EG 3 MALLOC *
Why would we want that?
Functionality:
Explain Malloc spec
Implementation:
Explain Malloc Call
Code
L1→L2→L3 L4→L5
Time

justified, he likes to give a few corollaries of that fact, phenomena which were previously unexplainable to the student, but now which makes sense given their new knowledge. These "newly-accessible-fact asides" buttress the new information – students are much less likely to forget a fact if it is the reason for another. Newly accessible fact asides are tangents that are now traversable because of the fact that the student has just learned. Sometimes, as in the assembly unit, entire lessons are based on "newly accessible fact" asides, reexplanations of C behavior now explainable in terms of assembly. Other times, Dondero's newly-accessible-fact aside is just a casual remark – after learning about pointers, for example, he mentioned that Java also uses pointers but abstracts them away so that the programmer doesn't have direct access to memory. This fact, surprisingly, stuck so well I remember expecting to hear it in his lesson a year later.

Other times, Dondero will use an aside to terminate a recursive string of questions. Just as when playing the "why" game with a toddler, a string of explanations must at some point end. Some facts are ultimately required by physical or mathematical restraints, and no further explanation can be given. Some facts are required relative to other ones – the fact that function calls must be proceeded by a declaration or definition, for example, is explained by the restrictions imposed by using a 2-pass compiler. The student must be satisfied with this explanation, because there is nothing further to grasp.

Some facts are a matter of convention, and Dondero justifies them by explaining the reasons for adopting the convention – explaining conventions of C function names, for example, by reminding students that there are no classes in C, and asking them to imagine how they would find the definition for a function in a 500 file C program? This hypothetical is an aside that buttresses the new facts about naming conventions, and which terminates the string of questions arising from a particular line of the program.

Some facts are true because their opposites are ridiculous, and Dondero will go one step further than other instructors, not just telling the student the fact, but asking them to imagine if it were otherwise. Some instructors might just tell students that the function declarations in their .c and .h files must match, or they'll get an error. Dondero explains one step further, asking the student to imagine what it would mean in terms of "contracts" for the .h file not to match the .c file – "hey world, here are these functions you can call according to this contract which I'm not going to follow! That'd be ridiculous!" he might say. This "counter-factual aside" is how he answers the "why" question motivated by a fact that cannot be explained by any further fact, and it therefore serves to terminate a string of questions, and provides a transition back to the next line of the C program (just like a real termination of a recursive call does.) Diagram 5 shows how we might actually model the recursive flow of this string of questions.

The picture that should be developing is one of a graph of interconnected facts and concepts, where the day's lesson furnishes several new nodes that must then be connected back to all prior knowledge through asides. Dondero's students will have a more densely connected understanding due to his many asides, mastery that is hard to test for, but which makes them more flexible thinkers. As we saw, knowing reasons is better than knowing rules – not only can you derive the rules from the reasons, but reasons put you in a position to see when certain general rules don't apply.

Can we say more about this type of recursion? I believe that there are further patterns to be found by classifying the type of facts and the types of strings of motivated questions they raise. I believe with greater time and data, it would be possible to so classify the types of questions raised by particular types of facts, thus putting us in a position to deterministically predict, given a program, what questions and asides will be motivated by which points.

The larger point I hope this drives home is that highly interactive and multifaceted lesson flows can be the result of simple deterministic rules, in the same way highly complex fractals can be generated with simple rules. Dondero's explanations involve skillful parries and leaps back and forth that must be the result of incredible effort and practice. Yet by viewing his highly dynamic instruction through the lens of functional categories, we see reoccurring patterns – patterns that can explain the capacities that develop in his students. A better understanding of these patterns would put us in a better position to evaluate and train new instructors – even great doctors had to memorize checklists once, and by telling new instructors what exactly they're missing will put them in a better position to improve, until these asides become second nature and they develope Dondero's constant intuition about where he is is the lesson and what points he needs to hit.

Conclusion

We set out to observe a great instructor and see if we could explain his methods. By developing a functional vocabulary for describing the different types of concepts and capabilities we wish to instill in students, and the corresponding explanatory methods that will instill them, we are getting closer to a full explanation, not just of Dondero's methods, but of the methods of felicitous explanation in general.

What would a full explanation look like? For a great instructor like Dondero, we should want to know exactly why he is doing what he does and not anything else. We set out to explain everything, the program-centered lessons, the frequent asides and diversions, the use and order of example program sequences, and so on. Though we are still far from that full explanation, I hope it is becoming clearer what it would look like.

I hope to leave the reader with a great deal of suspicion that great instruction is not just a matter of personal idiosyncrasy. That there are functional prerequisites to learning, grammar to explanation, and great instructors count as great because they've internalized and rigorously following these rules. There is room for innovation, just as there is in medical treatment and diagnosis – yet there is still an objective functional criteria that determines whether a treatment or explanation is effective.

This view entails a certain degree of determinism regarding explanation, that given a certain set of concepts and prior knowledge, there is an explanation or set of functionally equivalent explanations which is maximally efficient. Great instructors may vary greatly teaching the same material, just like chefs vary greatly in producing nutritionally equivalent food, but a full functional understanding of each part of instruction would show their different approaches equally valid, capable of producing concepts that are "built to the same spec".

I hope that this view seems at least marginally more reasonable than it did at our beginning, and perhaps it may even seem true. At the very least, I hope that some of the mysticism regarding great teaching has been diminished, so that novice instructors of Dondero's material know why he does what he does and what exactly they need to learn in order to achieve his level of mastery, though it may take many years for them to do so. And finally, I hope Dondero himself appreciates this recognition of the immense care he brings to his craft, far beyond what is expected by his institution, and respects this attempt to document for posterity the facts about human learning he has determined practically throughout a long and passionate career.