Question: Is there anything wrong with this code?

```c
char b[2][10000],*s,*t=b,*d,*e=b+1,**p;
main(int c,char**v)
{int n=atoi(v[1]);strcpy(b,v[2]);while(n--)
{for(s=t,d=e;*s;s++)
{for(p=v+3;*p;p++)if(**p==*s){strcpy(d,*p+2);d+=strlen(d);
goto x;}*d++=*s;x: }s=t;t=e;e=s;*d++=0; }puts(t); }
```
Outline

- Motivation and definition of refactoring
- Playing with real code examples
- Main refactoring strategies
- Practical suggestions
- When refactoring works and when it does not
References Used

**Recommended:**


**Other relevant resources:**

Iteration and Revision as Key to Software Development

As an intellectual (not routine) product, software is created through iterative work, in revisions.

If this were not the case:
- … the programming task could (and should!) be automated…
- … and programmers might need to look for more interesting (and less routine) jobs.
Motivating Question

- After a few revisions and/or after several years many software products get completely rewritten or abandoned.

What might be causing this?
Motivating Question (cont.)

- Many software products get completely rewritten or abandoned after a few versions and/or several years.

- One possible (and correct) cause is:
  - Code evolves to meet *evolving* business needs and *evolving* developer understanding.
  - If code’s structure does not evolve too, it will deteriorate (“rot”) over time, becoming increasingly hard to maintain and extend.
  - Related terms: “code rot”, “spaghetti code”
Motivating Thought Experiment

Case: Imagine you’ve written a piece of code but then accidentally deleted and lost it.

Questions:
- How much time would it take you to reconstruct from scratch what you had – the same amount, or more, or less?
- Would the code have a better design the second time you write it?
What It Takes to Do Effective Software Maintenance

- **Rule of thumb:** It’s harder to maintain (someone else’s) code than it is to write new code.
  - Most developers hope that they won’t have to deal with code maintenance.

- **Reality:** Maintenance is what most developers do most of the time.

- **Advice:** It literally pays to think ahead how you (or someone else) will maintain the code later.
  - Done as an afterthought turns it into a nightmare.
Fact:
- Code evolves
  - Contrary to the popular myth, most software projects cannot be first designed, then coded, then tested...
    - This waterfall lifecycle model does not work well for most software projects.

Corollary:
- (Evolving) code needs to be maintained to keep it from becoming a mess.
- Changes need to be anticipated via the design.
Refactoring Defined

“[Refactoring is] the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.” -- Martin Fowler

- **Note**: Refactoring is not the same as code rewriting; it is more disciplined and structured (as we will see).

- What is the “opposite” of refactoring?

- Why might one want to do that?
Refactoring – Why Do It?

Why is it necessary?

- A long-term investment in the quality of the code and its structure
  - Code structure deteriorates when last-minute fixes are made or unplanned features are added.
- Not doing refactoring may save costs / time in the short term but incurs a huge penalty in the long run
  - “Don’t be penny-wise but hour-foolish!”

Why fix it if it ain’t broken?

Every module has three functions:

- (a) to execute according to its purpose;
- (b) to afford change;
- (c) to communicate to its readers.

If it does not do one or more of these, it is broken.
Examples of What We Don’t Want to Have to Maintain

What is common among the following examples?

1) \[ q = ((p<=1) ? (p ? 0 : 1) : (p==-4) ? 2 : (p+1)); \]

2) \[ \text{while } (*a++ = *b--); \]

3) \[
\text{char } b[2][10000],*s,*t=b,*d,*e=b+1,**p; \text{main}(\text{int } c, \text{char**v})
\{ \text{int } n=\text{atoi(v[1])}; \text{strcpy}(b, v[2]); \text{while}(n--)\{ \text{for}(s=t,d=e; s; s++)\}
\text{for}(p=v+3; *p; p++)\text{if}(**p==*s)\{ \text{strcpy}(d, *p+2); d+=\text{strlen}(d); \}
goto x;\} *d++=*s; x;} s=t; t=e; e=s; *d++=0; \} \text{puts}(t); \}

21 May 2008  CSE403 Spring '08  Valentin Razmov
Examples of What We Don’t Want to Have to Maintain

What is common among the following examples?

1) \( q = ((p<=1) \ ? \ (p \ ? \ 0 : 1) : (p==-4) \ ? \ 2 : (p+1)) \);

2) \( \text{while} \ (\ast a++ = \ast b--) \);

3) \( \text{char b[2][10000],*s,*t=b,*d,*e=b+1,**p;main(int c,char**v)} \)
\( \{ \text{int n=atoi(v[1]);strncpy(b,v[2]);while(n--){for(s=t,d=e;*s;s++)}for(p=v+3;\ast p;p++)if(**p==*s){strncpy(d,*p+2);d+=strlen(d);goto x;}*d++=*s;x;}s=t;t=e;e=s;\}d++=0;}\text{puts(t);}\)

**Hint**: Can each of them:

(a) execute according to its purpose?
(b) afford change?
(c) communicate to its readers?
Communicating to Readers: The (Thorny) Issue of Style

- Remember when you tutored beginning programmers or curious friends…

- … You must have somehow explained to them why style mattered:
  - meaningful variable names
  - naming constants
  - standard indentation
  - etc.
  - even when the code worked as intended without those “extras.”
Let’s Do Some Refactoring!
class Account {
    float principal, rate;
    int daysActive, accountType;

    public static final int STANDARD = 0;
    public static final int BUDGET = 1;
    public static final int PREMIUM = 2;
    public static final int PREMIUM_PLUS = 3;
}

float calculateFee(Account accounts[]) {
    float totalFee = 0;
    Account account;
    for (int i=0; i<accounts.length; i++) {
        account = accounts[i];
        if ( account.accountType == Account.PREMIUM ||
            account.accountType == Account.PREMIUM_PLUS ) {
            totalFee += .0125 * ( account.principal
                * Math.exp( account.rate * (account.daysActive/365.25) )
                - account.principal );
        }
    }
    return totalFee;
}
float interestEarned() {
    float years = daysActive / (float) 365.25;
    float compoundInterest = principal * (float) Math.exp( rate * years );
    return (compoundInterest – principal);
}

float isPremium() {
    if (accountType == Account.PREMIUM || accountType == Account.PREMIUM_PLUS)
        return true;
    else return false;
}

float calculateFee(Account accounts[]) {
    float totalFee = 0;
    Account account;
    for (int i=0; i<accounts.length; i++) {
        account = accounts[i];
        if ( account.isPremium() )
            totalFee += BROKER_FEE_PERCENT * account.interestEarned();
    }
    return totalFee;
}

static final double BROKER_FEE_PERCENT = 0.0125;
Types of Refactoring

- Refactoring to patterns
- Renaming (methods, variables)
- Extracting code into a method
- Changing method signatures
- Performance optimization
- Naming (extracting) “magic” constants
- Extracting common functionality (including duplicate code) into a service / module / class / method
- Splitting one method into several to improve cohesion and readability (by reducing its size)
- Putting statements that semantically belong together near each other
- Exchanging risky language idioms with safer alternatives
- Clarifying a statement (that has evolved over time and/or that is hard to “decipher”)

21 May 2008
CSE403 Spring '08
Valentin Razmov
Language and Tool Support for Refactoring

- **Modern IDEs** (e.g., Eclipse, Visual Studio) support:
  - variable / method / class renaming
  - method or constant extraction
  - extraction of redundant code snippets
  - method signature change
  - extraction of an interface from a type
  - method inlining
  - providing warnings about method invocations with inconsistent parameters
  - help with self-documenting code through auto-completion

- **Older development environments** (e.g., vi, Emacs, etc.) have little or no support for these.
  - Discourages programmers from refactoring their code
When Making Code Changes…

In what order would you do the following?

- **X**: Refactor the code
- **Y**: Write unit tests to ensure that any conditions that need to be met are indeed met
- **Z**: Make the planned code changes

21 May 2008 CSE403 Spring '08 Valentin Razmov
Recommended Actions When Making Code Changes

1. Write unit tests to ensure that any conditions that need to be met are indeed met
   - Both before and after any refactoring or other changes you do

2. Refactor the existing code
   - To accommodate any necessary code changes and to make sure that the tests still pass

3. Make the planned code changes
A Bit of Practical Advice

- **Prioritize what needs to be refactored**
  - Not all parts of your code are equally important at all times – those which are touched more often are more likely to need refactoring sooner
  - This way it won’t feel like a useless, time-consuming exercise – but like something that helps you to more effectively do your job.
Transitioning from Technical to Business Arguments…
Refactoring – When to Do It?

Refactoring is necessary from a business standpoint too

- Helps to increase schedule predictability and achieve higher outputs at lower costs
- In general, ROI for improved software practices is 500% (!) or better
- By doing refactoring a team saves on unplanned defect-correction work

When is refactoring necessary?

- Best done continuously, along with coding and testing
- Very hard to do late, much like testing
  - Often forced before plunging into version 2
Refactoring in Context: Does Company Size Matter?

- Should refactoring be done in *small startups*? Can they afford it?
- Should refactoring be done in *large companies*? Can they afford it?
Refactoring in Context:
Small Startups, Pros

How refactoring may help in small startups:

- It’s an investment in quality, regardless of the size of the company
- Ideas and technologies are typically cutting edge and evolving quickly over time, so the code needs to also evolve at the same pace, to make it easier (not harder) to do the next change when it becomes necessary.
- Even with a small team, if a team member suddenly quits, it will be easier to take over his/her code and be able to maintain and extend it.
  - Also easier to get new team members quickly up-to-speed…
Refactoring in Context: Small Startups, Cons

- How refactoring may not help in small startups:
  - The company may never need to do another version (if the product is unsuccessful).
  - The company may desperately want to get to market as fast as possible, often at the expense of quality.
    - “[They’re] so busy sawing, there’s simply no time to sharpen the saw.” -- Steven Covey
  - The typical customers of v1 products (“early adopters”) want something working, not expecting a solid product yet.
Refactoring in Context: Larger Companies, Pros

How refactoring may help in larger companies:

- Having features is taken for granted. Doing so with good quality is what users demand – or else they may turn to the competition.

- The company aims the product for the long haul, so long-term investments are even more justified.

- More people work on the development of the product, over longer periods of time
  - The original code writer(s) may not be around to explain what they intended with a piece of code.
  - They’ll have saved themselves 5 minutes (by not clarifying) at the expense of 5 days for those who follow.
Refactoring in Context: Larger Companies, Cons

How refactoring may not help in larger companies:

- There’s often less sense of ownership of the code or the product than in smaller companies.
  - You don’t know the “poor” people who will have to maintain your code, so you care less about them.
    - … in contrast to a startup where the maintainer will be either you, or the person sitting next to you.

- Large companies are sometimes just former small companies that never realized they had grown.

- Company culture may not reward programmers for doing it.
  - E.g.: if performance evaluations are mostly based upon the delivery of immediate tangible results.
Food for Thought:
Analyzing the Incentives

Who is supposed to do the refactoring?

(A) programmer
(B) management
(C) maintainer
(D) user
Food for Thought: Analyzing the Incentives (cont.)

Who is supposed to do the refactoring?
(A) programmer
(B) management
(C) maintainer
(D) user

Who benefits from the refactoring?
(A) programmer
(B) management
(C) maintainer
(D) user
Analysis of the Incentives
Shows...

- Those who can do the job often do not have the incentive to do so.
- Those who need the job done can not do it by themselves.

Result:
Classic case of misalignment of incentives that often leads to situations where great ideas get stalled indefinitely.
Conclusion: Top Reasons for Refactoring

- Improving maintainability
  - … and hence productivity!
    - If you are spending much time doing maintenance work, reducing that even by a small fraction will make a big difference

- Responding to changes in the specification / design by improving the code structure
  - Or proactively preparing for (accommodating, anticipating) such changes
A Parting Perspective on Desirable Outcomes of Refactoring

“If bug rates are to be reduced, each function needs to have one well-defined purpose, to have explicit single-purpose inputs and outputs, to be readable at the point where it is called, and ideally never return an error condition.”

-- Steve Maguire, from “Writing Solid Code”