Fundamentos da Ciência da Computação (CSc 101)

Resultados esperados

O egresso de um curso de Ciência da Computação, em relação a aspectos de programação, deverá ser capaz de:

GERAL

01. Explicar tópicos gerais de computação:

a. os componentes fundamentais de hardware e software de um sistema de computador;

b. o processo de tradução e execução de programa em linguagem de alto nível;

c. as tendências históricas na economia do desenvolvimento de software (hardware mais barato, software mais caro).

02. Compreender as responsabilidades sociais e éticas do profissional de software.

RESOLUÇÃO DE PROBLEMAS

03. Seguir processo sistemático para desenvolver um programa completo:

a. analisar a proposição de problema para identificar os elementos-chave: requisitos de informação (que dados serão transformados) e requisitos funcionais (o que deverá fazer);

b. criar projeto de alto nível de uma solução para um problema usando decomposição funcional;

c. escrever projeto de baixo nível de uma solução usando uma notação independente de linguagem para representar um algoritmo, como pseudocódigo, diagrama N-S, PDL etc.

d. Executar passo a passo o algoritmo (usando papel e lápis, se necessário). (Também conhecido como rastreamento de código, verificação manual ou rastreio de execução).

e. Traduzir o algoritmo para linguagem de programação seguindo certo padrão de codificação.

04. Seguir uma prescrição de projeto simples, mas sistemática, para desenvolver uma função funcional:

a. decidir como representar os dados do problema usando os valores presentes no sistema;

b. identificar as entradas e saídas da função (número e tipo, incluindo condições para ambas);

c. fornecer uma Definição de Propósito resumindo a operação da função sucintamente;

d. fornecer exemplos do comportamento correto da função;

e. representar esses exemplos como casos de teste que possam ser executados automaticamente;

f. definir a função, seguindo a estrutura dos dados de entrada;

g. quando necessário, decompor programas complexos em programas mais simples, seguindo essa prescrição para cada subcomponente;

h. Testar a solução em relação aos exemplos fornecidos;

i. Detectar, isolar e corrigir defeitos na implementação;

j. Conhecer e aplicar técnicas gerais para resolução de problemas:
abstração,
refinamento gradual,
decomposição,
modularização,
implementação incremental,
isolamento de falhas
etc.

05. Usar comandos e utilitários apropriados do sistema operacional para desenvolvimento de software (editor, compilador, comandos de manipulação de arquivos).

06. Ser capaz de inserir código-fonte em uma linguagem de programação de alto nível e realizar as etapas necessárias para compilar e executar programa em um computador específico.

DADOS SIMPLES

07. Identificar e conhecer as definições

de todos os tipos de dados primitivos fornecidos pela linguagem:
inteiros, números de ponto flutuante, lógicos, caracteres, *strings*;

das operações que possam ser realizadas com esses e como selecionar o mais apropriado para representar um item de dados em um problema.

08. Entender como a precisão numérica limitada do computador pode afetar os cálculos.

09. Projetar e implementar programas que operem sequencialmente sobre dados primitivos.

10. Projetar e implementar programas que usem expressões/instruções condicionais simples e aninhadas para distinguir categorias de entrada.

11. Projetar e implementar algoritmos contendo expressões que usem (avaliem) o valor oriundo de subprograma com retorno de valor ou "função auxiliar".

12. Entender que funções lógicas possam ser usadas para controlar condicionais e também como valores.

DADOS COMPOSTOS:

13. Entender que dados/estruturas/tuplas/objetos compostos encapsulam vários valores como dado único.

14. Projetar e implementar definições de dados compostos para representar valores compostos.

15. Ser capaz de construir valores compostos e acessar seus campos.

16. Projetar e implementar programas que operem com valores compostos.

SEQUÊNCIAS:

17. Estar familiarizado com as estruturas de dados que uma linguagem fornece para modelar sequências (matriz, tabela, lista etc.).

18. Ser capaz de usar operações básicas de sequência: criar, mostra, ler, alterar.

19. Projetar e implementar algoritmos que usem repetições controladas por contagem:

    a. funções do tipo mapeamento para acessar/converter uma sequência em outra;

    b. funções do tipo filtrar para retornar uma subsequência da sequência fornecida;

    c. funções do tipo reduzir para acumular/produzir um valor resultante da execução de uma sequência.

20. Projetar e implementar algoritmos que usem repetições controladas por eventos:

    a. funções do tipo procurar/buscar capazes de interromper a execução ao encontrar certo elemento.

21. Projetar e implementar funções que aceitem sequências como entradas.

ENTRADA/SAÍDA:

22. Ser capaz de usar operações básicas de saída (mostrar em console, gravar em arquivo).

23. Estar familiarizado com operações para ler dados de arquivos de texto sequenciais ou de console.

24. Ser capaz de executar análise sintática rudimentar (extrair um número, dividir em espaços ou delimitadores) em cadeias de caracteres

25. Projetar e implementar programas que realizem repetições sobre sequências de entrada.

ABSTRAÇÃO

26. Identificar redundâncias em um projeto que se prestem à modularização e identificar módulos de um projeto que possam ser reutilizados a partir de outro desenvolvido de forma independente.

Como estudar para Ciência da Computação

Aprender uma linguagem de programação é considerada uma tarefa que envolve mais trabalho do que as comumente encontradas em outras áreas, porque requer conhecimento bastante detalhado, bem como resolução de problemas abstratos bem elaborados.

Os hábitos de estudo que funcionam (ou funcionaram) em outras situações poderão não ser suficientes para ter sucesso nesse curso.

A seguir estão algumas sugestões de técnicas de estudo e estratégias de aprendizagem para ajudá-lo a ser fluente em uma nova linguagem de programação.

Como ler livro didático

1. Ler cada capítulo do livro didático pelo menos três vezes.

Primeira leitura ("leitura para orientação")

Esta primeira leitura é apenas para se orientar quanto ao conteúdo do capítulo.
A primeira leitura é superficial. É possível pular exemplos.
Não tentar compreender todos os detalhes.
Apenas se familiarizar com as ideias gerais que estão sendo discutidas.
Familiarizar-se com o propósito geral e os objetivos, os pontos principais que estão sendo apresentados e a maneira como o capítulo é apresentado.
Em seguida, deixá-los de lado e colocá-los em "incubação" em seu cérebro durante a noite.

Segunda leitura ("leitura para compreensão")

A segunda leitura é para se ler com objetivo de compreender e entender os conceitos e ideias que estão sendo apresentados.
Tente entender cada nova ideia que encontrar.
Não apenas mover os olhos pela página.
Parar após cada parágrafo e perguntar a si mesmo "isso faz sentido?"
"O que acabei de aprender naquele parágrafo?"
Se não se sentir confiante em sua compreensão, ler novamente.
Se ainda não entender, colocar um ponto de interrogação na margem e voltar após terminar a seção.
Se ainda não fizer sentido, fazer pergunta em sala de aula, perguntar a um monitor ou colega, ou perguntar ao instrutor/professor.

Observar os exemplos para ver como eles se relacionam com os conceitos ou princípios.
Certificar-se de que consegue ver como o exemplo é uma instância específica do conceito geral que está sendo apresentado.

Toda vez que o autor introduzir um novo termo de vocabulário ou conceito,
dizer a palavra em voz alta. Falar o trabalho em voz alta ativa uma parte diferente do seu cérebro e o ajudará a reter a palavra e a se lembrar dela mais prontamente.
Lembrar-se, metade de ser um especialista é falar como um especialista.
Se você usar a palavra adequada, estará na metade do caminho para dominar o conceito.

No final de cada seção, reservar um minuto para relembrar conscientemente os principais pontos aprendidos.
Talvez anotá-los.
Então, voltar às páginas e certificar-se de que se lembrou de todos eles.

Terceira leitura ("leitura para compreensão detalhada")

Na terceira leitura, é para envolver-se com o texto.

A terceira leitura é a mais importante e difere das duas anteriores porque deve haver uma participação ativa com o conteúdo.
Ler o livro novamente, mas dessa vez concentrar-se na compreensão detalhada.
Ler cada frase com o objetivo de compreensão completa. O objetivo é dominar cada detalhe.
Isso é importante na programação porque um programa bem-sucedido
requer que cada detalhe seja empregado corretamente.
Mesmo um pequeno erro fará com que seu programa falhe e, às vezes,
poderá falhar de forma catastrófica.
Você não precisa memorizar todos os detalhes, mas precisa entender o que eles significam e como usá-los de forma correta.
Em um teste de laboratório, poderá usar outros recursos,
não é necessário memorizar todos os detalhes de sintaxe.
Por exemplo, caso precisar saber que há um operador aritmético que retornará o resto inteiro após uma divisão.
É preciso, primeiro, saber como isso funciona e em quais situações poderá ser usado.
Mas não precisar memorizar que o símbolo é um sinal de porcentagem.
(Poderá ser diferente em outras linguagens. Poderá procurar o símbolo quando precisar.)

Estudar os exemplos.

Rastrear cada etapa de cada exemplo.
Certificar-se de compreender as transformações em cada etapa na solução de um problema.
Uma técnica é inspecionar cada linha de um exemplo e descrever em voz alta para si mesmo ou para um colega de estudo o que cada etapa está realizando.
Em um programa de exemplo, deverá ser capaz de explicar o propósito de cada caractere ou símbolo.
Cada detalhe está lá por um motivo e é preciso ser capaz de explicá-lo.

Outra técnica para trabalhar com o programa de exemplo é fazer um rastreamento manual ou verificação passo-a-passo do programa e ver se consegue calcular os mesmos resultados mostrados no exemplo.

Explorações - interagir com programas

Uma das estratégias de aprendizado mais poderosas é usar o próprio computador para interagir com programas de exemplo.
Essa é uma estratégia tão importante que merece ser explicada em detalhes.
Ler Estratégias de Exploração: como aprender uma linguagem de programação.

Deverá realizar várias explorações usando os programas de exemplo fornecidos no capítulo.
O objetivo é obter uma compreensão completa de cada nova construção, conceito ou técnica apresentada no capítulo.

Autoteste - Avaliar o conhecimento

A maneira mais simples de avaliar se aprendeu o material é concluir todos os exercícios de autoverificação ou exercícios de revisão de capítulo fornecidos pelo autor.
Frequentemente, respostas poderão ser fornecidas em apêndices.
Se tiver dificuldades com um exercício, revisar a seção apropriada no capítulo.
Se ainda assim não conseguir, anotar e pedir ajuda mais tarde.
Normalmente, o instrutor poderá responder a esses tipos de perguntas por mensagem ou *e-mail*.

Em seguida, trabalhar os "problemas de previsão" fornecidos pelo instrutor.

Esses são programas, em geral, serão curtos para se examinar.

Estudar cada programa para tentar compreender quais tarefas ele irá executar.

Em seguida, executar um rastreamento manual para acompanhar a execução do programa usando os dados de entrada fornecidos. Registrar a execução passo-a-passo à parte.

Ao executar o rastreamento manual, poderá encontrar uma declaração sobre a qual não tem certeza.
Reservar um minuto para consultá-la no capítulo e ver se consegue determinar como ela irá operar.

Quando tiver concluído o rastreamento manual, deverá ter uma "previsão" sobre qual saída será produzida pelo programa.

Depois, compilar e executar o programa em computador e observar os resultados reais produzidos.

Examinar cuidadosamente os resultados reais e compará-los caractere por caractere com sua previsão.

Se houver uma discrepância, isso indicará algum mal-entendido em relação ao enunciado.
Revisar o programa, consultar novamente o enunciado, até que possa explicar a discrepância.
Fazer uma anotação para enfatizar o entendimento correto.
Se não puder explicar a discrepância, fazer uma anotação e pedir ajuda mais tarde.

De forma semelhante às explorações acima, poderá criar pequenos experimentos para verificar sua compreensão.

Prever o efeito de alguma mudança em um programa de exemplo, executar a mudança no programa e verificar sua previsão.

Estudar com um ou dois colegas é uma estratégia excelente.

Ter parceiros é especialmente útil nessa fase de "autoteste".
Poderão inventar perguntas de teste para fazer um ao outro, com base nas perguntas de autoverificação.
Poderão criar problemas de previsão um para o outro, com base nos programas de exemplo no texto.

Finalmente, o teste mais completo de sua compreensão é escrever um programa completo.

Começar com os problemas de programação no final do capítulo.
Ler todos os problemas e formular um plano de como resolvê-los.
Deverá ser capaz de pelo menos elaborar uma estratégia para tratar a maioria dos problemas.
Pode haver um ou dois problemas avançados ou "desafiadores" que resistam a uma solução fácil; deixar esses para mais tarde.

Em seguida, selecionar dois ou três problemas que pareçam interessantes, escrever soluções e tentar fazê-las funcionar no computador.

Seu objetivo é obter "fluência", ou seja, ser capaz de trabalhar na solução do início ao fim sem grandes problemas em um período de tempo razoável.
Claro, o que é "razoável" varia de problema para problema; pedir orientações ao seu instrutor em caso de dúvidas.
Se não puder prosseguir, fazer uma anotação para pedir ajuda mais tarde.
Caso tropeçar em uma solução, ainda que ligeiramente, isso indicará a necessidade de mais prática.

Recursos

Seus colegas de classe.
Amigos em outras classes.
Grupos de estudo.
Monitores/tutores.
O instrutor/professor
(em seu horário de expediente!)

Livros didáticos alternativos.
Buscar outros livros didáticos pertinentes.
O instrutor poderá indicar outros disponíveis.
Visitar a biblioteca ou uma livraria.
Ler um livro didático alternativo e compará-lo ao seu livro didático principal é um excelente exercício.

Como construir um programa

Construir um programa não é uma arte, é um processo em boa parte artesanal, mas sistemático. Como tal, há um processo a seguir com etapas claramente definidas.

01. Analisar (Entender os requisitos)

Ao receber os requisitos do problema, e isso não é diferente do mundo real, precisará levantar seus requisitos.

O primeiro passo será ler o problema e ter certeza de ter entendido bem o que o programa deverá fazer.

Resumir sua compreensão escrevendo os dados de entrada, dados de saída e funções (operações ou transformações nos dados).

02. Criar um plano de teste.

Deverá ser capaz de verificar se seu programa funcionará corretamente depois de escrito. Escolher alguns valores de dados reais e calcular manualmente o resultado esperado.

03. Elaborar uma solução

Essa é a parte criativa e exploratória do projeto, onde deverá decidir como resolver o problema.

04. Apresentar a estratégia

1. Resolver o problema manualmente, da mesma maneira como mostraria a outra pessoa.
2. Prestar atenção às operações que executar e anotar cada etapa.
3. Procurar um padrão nas etapas que executar.
4. Determinar como esse padrão pode ser automatizado usando os três blocos básicos para construção do algoritmo (sequência, seleção, repetição).

05. Projetar (formalizar sua solução)

06. Organizar sua solução em componentes. Isso é chamado de arquitetura.

07. Escrever o algoritmo para cada componente.

Refinar seu algoritmo de forma gradual, se necessário.

08. Determinar os tipos de dados e restrições para cada item de dados.

09. Revisar

10. Realizar um rastreamento manual da solução e simular como o computador executará seu algoritmo.

11. Certificar-se de que seu algoritmo funcionará corretamente antes de colocá-lo no computador.

12. Implementar (codificar)

    Traduzir seu algoritmo para uma linguagem de programação e inseri-lo no computador.

13. Compilar seu código-fonte para produzir um programa executável.

    É possível compilar e testar cada subprograma individualmente antes de combiná-los em um programa completo.

14. Testar

    Executar o programa usando os Planos de Testes criados inicialmente ou compostos durantes as etapas anteriores.

15. Corrigir quaisquer erros caso necessário.

ANÁLISE INFORMAL

Esclarecer os requisitos do problema.

A análise é a primeira fase do processo de desenvolvimento de software.

Objetivo:

Entender o problema antes de tentar resolvê-lo.

Certificar-se de que os requisitos sejam totalmente especificados e compreendidos pelo desenvolvedor.

Analisar a declaração do problema (ou requisitos) e escrever um documento que especifique o comportamento do software a ser construído. Esse documento será chamado de "especificação de software". É uma declaração precisa de exatamente **o que** o software deverá realizar. (Frequentemente, a especificação é o contrato legal que vinculará o desenvolvedor e o solicitante do software).

Método:

Como a linguagem natural é notoriamente imprecisa e sujeita a má interpretação, o trabalho do analista é examinar a declaração do problema para descobrir quaisquer possíveis ambiguidades.

Ler a declaração do problema e anotar todas as perguntas que lhe ocorrerem, quaisquer declarações que sejam confusas ou sujeitas a má interpretação.

Suas perguntas deverão abordar quaisquer ambiguidades, omissões, inconsistências, imprecisão, confusão ou falta de clareza nos requisitos do problema. Apresentar suas perguntas ao solicitante (ou instrutor) e usar o *feedback* dele para escrever uma declaração de problema mais esclarecedora.

Se não conseguir obter *feedback*, fazer o melhor que puder para determinar as respostas por si mesmo e escrever uma lista de todas as suposições que fizer.

Depois de estudar completamente o problema e sentir que entendeu completamente o que está sendo perguntado, poderá reescrever os requisitos do problema na forma de uma especificação.

Os componentes de uma especificação são uma descrição dos dados de entrada, os dados de saída e as funções que o software deverá executar.

Um método é sublinhar todos os substantivos na proposição do problema — esses são os itens de dados, e sublinhar todos os verbos — essas são as funções.

Em seguida, escreva uma descrição para cada palavra sublinhada como um item separado (ou "requisito") na especificação. O ideal é que cada declaração na especificação esteja em uma linha separada com um número de linha para fácil referência em documentos posteriores (como o plano de teste).

Um cuidado:

Não tentar descrever COMO resolver o problema, apenas tentar descreva O QUE o software fará. Descrever apenas o comportamento observável externamente. Esforçar-se para obter clareza!

Uma boa especificação é clara, precisa, consistente, compreensível, inequívoca e verificável.

Fazer perguntas ao analisar

A linguagem natural geralmente não tem o tipo de precisão de expressão necessária para descrever operações computacionais e lógicas.

Abaixo está um exemplo mais realista de um requisito que pode ser encontrado na declaração do problema para um sistema de folha de pagamento:

"Ler o dia da semana e o número de horas trabalhadas."

Perguntas para análise:

Presume-se que "ler" significa entrar no teclado digitando, e não selecionar em um menu.
O programa deverá exibir algum tipo de mensagem solicitando a entrada?
O que exatamente a mensagem deverá dizer?

O dia da semana deverá ser escrito por extenso, por exemplo, "terça-feira",
ou poderá ser abreviado? Ou ambos? Qual abreviação, exatamente?
Deverá ser sensível a maiúsculas e minúsculas?

O número de horas é um inteiro?
Um número decimal? Quantas casas decimais?

Há apenas 24 horas em um dia. E se as horas inseridas forem maiores que 24?
Menores que zero?

O dia e as horas devem ser inseridos na mesma linha?
Linhas diferentes? Ambos?

Como se pode ver, é fácil desenvolver uma longa lista de perguntas sobre até mesmo uma declaração simples.

É verdade que um analista experiente poderá fazer algumas suposições razoáveis para a maioria dessas perguntas, mas essas suposições deverão sempre ser explicitadas e comunicadas ao solicitante.

No caso de uma tarefa, é uma boa ideia verificar novamente com seu instrutor para verificar se foram feitas as suposições corretas sobre o problema que deverá resolver.

É quase impossível remover **todas** as incertezas em uma especificação em linguagem natural, então os cientistas da computação desenvolveram métodos mais formais de especificação de software que estão além do escopo de um curso introdutório.

Para este curso, se considerará análise completa quando se tiver uma especificação escrita com clareza suficiente (em linguagem natural) para que nenhuma pessoa tenha dúvidas sobre o que o problema requer.

Isso exigirá um pouco de julgamento de sua parte para avaliar quando suas especificações estão escritas com precisão suficiente para comunicar efetivamente os requisitos a outra pessoa.

Consultar ar exemplo a seguir, bem como os estudos de caso no livro didático para ter uma ideia do que constitui uma boa especificação.

Exemplo de análise

Declaração do problema

Em uma recente competição de corrida de tartarugas, seu animal de estimação "Speedy" ficou em primeiro lugar.

Os juízes da corrida determinaram que a taxa média de deslocamento de Speedy é de um pé a cada dez segundos.

Só por diversão, isso despertou-lhe a curiosidade saber sobre quanto tempo Speedy levaria para viajar ao redor do mundo.

Escrever, portanto, um programa que permita ao usuário inserir uma distância em milhas e calcular o tempo de viagem de Speedy.

Suposições

A taxa de deslocamento de Speedy é uma constante dentro do programa. A distância inserida em milhas deverá ser um número inteiro positivo.

A distância máxima permitida é de 99.999 milhas.

Dados de entrada inválidos deverão fazer com que uma mensagem de erro seja exibida e o programa seja encerrado.

A saída deverá ser um número decimal em horas, com duas casas decimais.

O programa deverá exibir uma mensagem de prompt apropriada, ler a entrada do usuário no teclado e exibir os resultados na tela com uma observação explicativa.

Nenhuma verificação será realizada para ver se os cálculos excederão o intervalo numérico permitido da máquina e, consequentemente, resultados inválidos poderão ocorrer.

Entrada

Distância a ser percorrida (em milhas), um número inteiro positivo menor que 99.999.

Saída

Sugestão de mensagem:
"Digitar a distância para Speedy percorrer (em milhas)."

Uma observação explicativa:
"A uma velocidade de um pé a cada dez segundos,
Speedy levaria _____ horas para percorrer _____ milhas."

O número de horas (número decimal com 2 casas decimais).

A distância da viagem (mesmo valor da entrada).

Funções

1. Exibir a mensagem.
2. Ler a entrada do usuário sobre a distância a ser percorrida e verificar se a distância é número válido.
3. Calcular o tempo de viagem (distância dividida pela taxa)

Dados:

velocidade = 1 pé / 10 segundos
1 hora = 3600 segundos
1 milha = 5280 pés

entrada do usuário: distância

fórmula para o tempo de viagem:

(distância * 5280) / (3600 * (1 / 10))

4. Exibir a distância, o tempo de viagem e a mensagem explicativa.

Problemas de análise prática

Poderá praticar seu questionamento de análise e redação de especificações nas proposições de problemas de exemplos abaixo.

Escrever um programa que receba suas leituras do medidor elétrico (em quilowatts-hora) no início e no final de cada mês do ano.

O programa determinará seu custo anual de eletricidade com base em uma cobrança de 6 centavos por quilowatt-hora para os primeiros 1000 quilowatts-hora de cada mês e 8 centavos por quilowatt-hora além de 1000.

Após imprimir suas cobranças anuais totais, o programa também determinará se houver uso de menos de 500 quilowatts-hora durante o ano inteiro e, se sim, imprimirá uma mensagem agradecendo por poupar eletricidade.

Escrever um programa que determinará o imposto estadual adicional devido por um funcionário. O estado cobra um imposto de 4% sobre a renda líquida.

Determinar a renda líquida subtraindo uma mesada de $ 500 para cada dependente da renda bruta. O programa lerá a renda bruta, o número de dependentes e o valor do imposto já deduzido.

Então calculará o imposto real devido e exibirá a diferença entre o imposto devido e o imposto deduzido seguido pela mensagem "PAGAR" ou "REEMBOLSAR", dependendo se essa diferença for positiva ou negativa.

A *New Telephone Company* tem a seguinte estrutura de tarifas para chamadas de longa distância:

a. Qualquer chamada iniciada após as 18:00 (18:00 horas),
   mas antes das 8:00 (08:00 horas), terá desconto de 50%.
b. Qualquer chamada iniciada após as 8:00 (08:00 horas),
   mas antes das 18:00 (18:00 horas), terá o preço integral cobrado.
c. Todas as chamadas estarão sujeitas a um imposto federal de 4%.
d. A taxa normal para uma chamada será de US$ 0,40 por minuto.
e. Qualquer chamada com mais de 60 minutos receberá um desconto de 15% em seu custo
   (após qualquer outro desconto ser subtraído, mas antes do imposto ser adicionado).

Escrever um programa para ler o horário de início de uma chamada com base em um relógio de 24 horas e a duração da chamada.

O custo bruto (antes de quaisquer descontos ou impostos) deverá ser mostrado, seguido pelo custo líquido (após os descontos serem deduzidos e o imposto ser adicionado).

(De Programação e resolução de problemas em Ada, por Michael Feldman, Addison-Wesley, 1992).

ESTRATÉGIAS DE EXPLORAÇÃO: COMO APRENDER UMA LINGUAGEM DE PROGRAMAÇÃO

Aprender uma linguagem de programação é semelhante a aprender uma linguagem natural como outro idioma.

Embora o "vocabulário" de uma linguagem de programação geralmente tenha menos de 100 palavras, há muitas regras que governam a formação correta de frases ou "declarações" na linguagem.

Essa "gramática" de uma linguagem de programação é bastante complexa, e as regras gramaticais são muito detalhadas e precisas.

Durante uma conversa em linguagem natural, é possível haver ambiguidades consideráveis ou imprecisões gramaticais e ainda ser compreendido.

Não é assim com linguagens de programação.

As inúmeras regras que ditam o que constitui uma declaração de programa correta deverão ser seguidas com precisão, rigorosamente.

Qualquer pessoa que tenha aprendido a falar uma segunda língua concordará que a melhor maneira de ganhar fluência é visitar um país onde essa língua é falada.

Ao se cercar de outras pessoas que estão falando e ouvindo nessa língua, cria-se uma situação que oferece *feedback* abundante para o aprendizado.

Ao conversar em um novo idioma diariamente, ganha-se muita prática e recebe-se muito *feedback* sobre suas tentativas.

Da mesma forma, pode-se aprender melhor uma linguagem de programação criando uma situação que ofereça muito *feedback*.

Felizmente, o próprio computador oferece tal situação, já que sua natureza interativa fornece muito *feedback* sobre nossas tentativas de "falar" esse novo idioma.

É possível tentar aprender todas as regras para formular instruções corretas do programa somente em livros, mas isso seria como tentar aprender a gramática correta de uma língua sem nunca falar ou ouvir o idioma.

O método empírico

Ao aproveitar a natureza interativa do computador, acabará se envolvendo nos mesmos métodos empíricos em que todas as ciências confiam.

A maneira como se aprende uma linguagem de programação é análoga a um cientista estudando o mundo natural.

Um físico, por exemplo, realiza "experimentos" jogando bolas na tentativa de entender como a gravidade funciona. Dessa forma procura reunir evidências sobre certo fenômeno para obter ideias sobre os princípios físicos subjacentes que governam o comportamento que está sendo observado.

No caso de estudar uma linguagem de programação, o fenômeno observado é um programa de computador.

Quando executado no computador, esse programa tem certos comportamentos observáveis externamente.

Ao experimentar sistematicamente o programa, reunindo evidências sobre seu comportamento, se poderá construir inferências sobre os princípios ou regras que ditam como ele opera.

Eventualmente, o cientista irá conceituar suas observações em uma teoria a partir da qual se pode fazer previsões sobre o comportamento do fenômeno em diferentes situações.

Da mesma forma, conforme for "descobrindo" como a linguagem de programação se comporta, e sua compreensão da mesma poderá ser medida pelo grau em que poderá fazer previsões precisas sobre a maneira como os recursos dessa operam em diferentes circunstâncias.

Provavelmente, isso permitirá construir programas a partir de seus próprios projetos.

Anotações de laboratório

Um cientista trabalhando em um laboratório procede de forma organizada e sistemática, fazendo anotações exaustivas de seus procedimentos e observações. As razões para isso são duas.

Primeiro, ao explorar uma área desconhecida, nunca se sabe quais detalhes poderão ser significativos, então é importante registrar tudo.

Segundo, a documentação detalhada dos procedimentos seguidos e dos resultados obtidos será necessária para permitir que outros repliquem as descobertas do cientista.

Da mesma forma para suas explorações, proceder de forma cuidadosa e organizada é a chave para ser eficaz.

Se seus hábitos de trabalho forem desleixados, desorganizados ou aleatórios, seu trabalho não fará sentido e tentar chegar a qualquer entendimento significativo será difícil.

A precisão e exatidão implacáveis exigidas pelo computador exigem de nós um grau incomum de cuidado e atenção aos detalhes. Sem uma abordagem sistemática, detalhes minúsculos, mas significativos, da linguagem de programação podem escapar e levar a mal-entendidos e confusão.

Embora possa não ter que compartilhar suas descobertas com outros pesquisadores, sair-se bem nas práticas de laboratório se tornam uma referência valiosa à qual poderá retornar quando encontrar situações semelhantes no futuro.

Muitas vezes poderá encontrar-se diante de uma mensagem de erro obscura do computador que pareça familiar, mas não conseguirá se lembrar do que significa.

Se mantiver boas anotações de laboratório, então é uma questão simples procurar como já lidou com esse problema em um momento anterior.

Se você acabar buscando assistência externa com problemas de computador, terá um registro detalhado de suas dificuldades e isso tornará muito mais fácil para um especialista diagnosticar seus problemas.

É altamente recomendável guardar os registros de suas explorações.

Usar o bloco de notas é recomendável para se registrar tudo o que você faz no computador.

Escrever de forma legível e incluir detalhes suficientes para que outra pessoa lendo suas anotações possa replicar seu experimento.

Para cada experimento que realizar, documentar o propósito do experimento, os resultados esperados, os procedimentos passo a passo que foram seguidos, os resultados reais observados e as conclusões — o que se aprendeu. (Também poderá manter um registro de datas e tempo gasto como um auxílio no gerenciamento de tempo).

Exemplos de anotações de laboratório estão incluídos no final deste texto.

No início, poderá se sentir relutante em manter anotações de laboratório, pois parece um trabalho tedioso. No entanto, à medida que aprender o quão "exigente" o computador é, se fizer um esforço para manter boas anotações, achará isso extremamente útil.

Em laboratório, é dito que "não há erros, apenas surpresas".

Cada coisa que fizer no computador é uma chance de aprender algo, independentemente de sair ou não como você esperava.

Então, não tente fazer suas anotações de laboratório parecerem que tudo funcionou perfeitamente.

As anotações de laboratório não são destinadas à publicação, elas são o lugar onde se documentará seus aprendizados.

É importante explicar claramente cada "surpresa" que encontrar e o que aprendeu com ela. Eventualmente, todas essas ideias começarão a se acumular em uma compreensão coerente do computador e do software que o controla.

Construir experimentos

A maneira mais eficaz de aprender programação é colocar as mãos no computador, praticar como "exprimir-se" nessa nova linguagem e observar o feedback que receberá.

Se estiver fazendo um curso formal de programação, o instrutor sem dúvida fornecerá exercícios e atividades para executar no computador como uma forma de praticar.

Alguns livros didáticos são publicados com "livro de exercícios" de laboratório que o acompanha.

Se tal livro de exercícios estiver disponível, realizar as atividades sugeridas seria um bom lugar para começar suas explorações.

As atividades do livro de exercícios geralmente são muito bem estruturadas e acabará simplesmente trabalhando com elas como se fosse um manual ou um livro de receitas.

Mas se for um aluno independente, poderá não ter o luxo (ou o fardo, dependendo do seu ponto de vista) das atividades atribuídas pelo instrutor.

Poderá não ter um livro de exercícios com atividades já preparadas para executar. Sem essas diretrizes prescritas externamente para prosseguir, por onde começar?

Talvez a melhor abordagem para explorar uma linguagem de programação seja aproveitar os programas de amostras fornecidos em um livro didático.

A maioria dos livros didáticos, tutoriais e outros manuais de programação apresentam vários programas como exemplos para ilustrar os principais recursos da linguagem.

Esses exemplos por si só, mesmo sem as explicações que os acompanham, são recursos valiosos para o aprendizado.

Infelizmente, o livro didático não explica como aproveitar esses programas de exemplos.

Os exemplos são apresentados como se apenas a leitura bastasse para o entendimento.

O que o livro didático não diz é que deverá interagir com esses exemplos.

Ler não é suficiente.

Para realmente entender esses exemplos, precisará inseri-los no computador e explorá-los.

O que está sendo sugerido é um tipo de investigação aberta, não um método ou procedimento rígido.

Não há maneira certa ou errada de prosseguir. Assim como muitas crianças curiosas puxaram a parte de trás de um brinquedo novo para tentar ver como ele funciona, os exemplos do livro didático não deverão ser vistos como "tábuas de sabedoria", mas como brinquedos clamando por serem desmontados e experimentados.

Infelizmente, esse tipo de exploração lúdica requer uma estrutura mental inquisitiva que a maioria das experiências escolares não incentiva.

Depois de uma dúzia de anos de escola pública, sua curiosidade pode ter se recolhido em um armário na sua mente. Então, o desafio de prosseguir independentemente para explorar o computador poderá fazer com que precise desenterrar estratégias antigas que não usa há muito tempo.

Outra característica da educação formal é a redução de ideias complexas em questões simples de múltipla escolha e preenchimento de lacunas.

Ainda que possa ter se tornado hábil em descobrir como ter sucesso nesses testes, ainda assim, muitas vezes, poderá sentir-se inseguro de ter compreendido todo o quadro.

Escrever um programa de computador, por outro lado, é um esforço integral e criativo, e a abordagem fragmentada para a compreensão pode não funcionar muito bem.

Poderá ter que abandonar sua abordagem habitual de "copiar e colar" para a resolução de problemas.

A próxima seção apresentará algumas estratégias de exploração específicas para ajudá-lo a entrar no estado de espírito para interagir com os exemplos do livro didático.

Poderá usar essas estratégias ao construir seus próprios experimentos.

Essas diretrizes não dirão exatamente o que fazer, em vez disso, descreverão estratégias e táticas.

Cabe a você criar os experimentos reais que realizará para cada novo tópico que investigar.

Para cada programa que explorar, precisará considerar quais novos recursos ou técnicas importantes da linguagem estão sendo ilustrados e construir um experimento que se concentrará no comportamento de interesse particular.

Esse pode ser um novo desafio, já que a maioria dos professores não exige que os alunos inventem suas próprias tarefas. Mas com linguagens de programação, uma habilidade essencial de aprendizagem é ser capaz de projetar seus próprios experimentos para aprender sobre as regras da linguagem que você não entende.

As diretrizes abaixo fornecem algumas estratégias poderá usar, mas aplicar essas estratégias exigirá alguma criatividade e imaginação de sua parte, pois cada programa que encontrar será diferente. Sua própria curiosidade natural e inquisitividade geralmente serão toda a centelha de que precisará para fazer sua criatividade fluir.

ESTRATÉGIAS DE EXPLORAÇÃO DE LABORATÓRIO

Digitar programas de exemplos.

A primeira atividade muito simples, mas surpreendentemente útil, é inserir um programa no computador digitando de uma listagem de programas fontes no livro didático. Especialmente quando estiver começando, descobrirá que há regras de pontuação muito precisas para a linguagem que estiver estudando.

Cada símbolo deverá aparecer exatamente na relação correta com outros símbolos.

Não há melhor maneira de se familiarizar com os pequenos detalhes da sintaxe de programação do que sentar-se ao teclado e reproduzir um programa exatamente como ele é mostrado em exemplo de livro didático.

Selecionar qualquer um dos primeiros exemplos do livro didático, inseri-lo no computador com um editor de texto e compilá-lo.

Se não cometer erros, o programa deverá compilar sem erros.

Se você cometer um erro de digitação ou não copiar com precisão o código do livro didático, poderão ocorrer erros de compilação.

Deverá ser capaz de resolver esses erros facilmente verificando seu trabalho, comparando o que foi copiado do livro didático.

Depois de digitar seus primeiros programas, começará a se sentir confortável com as regras de pontuação da linguagem, e digitar programas à mão se tornará menos útil.

Muitos livros didáticos fornecem códigos prontos para testar os programas de exemplos.

Se isso estiver disponível, poderá evitar a digitação manual todos os exemplos.

Se não houver tal recurso, entre em contato com o instrutor para descobrir se há outro disponível. Há autores de livros didáticos que disponibilizam os exemplos de livros didáticos de forma particular ou através das editoras.

Executar programas de exemplos.

Selecionar um programa que seja um exemplo interessante do tópico que estiver estudando.

Se seu texto incluir "estudos de caso", eles geralmente são excelentes exemplos.

O estudo de caso geralmente contém uma execução de amostra parcial ou completa do programa.

Ler o estudo de caso para determinar o que o programa deverá realizar.

Tentar localizar uma execução de amostra que apresente quais dados de entrada serão fornecidos ao programa e como deverá ser a saída.

Em seguida, compilar o código-fonte real e executá-lo.

Inserir os mesmos dados de entrada fornecidos na execução de amostra no livro didático.

Obviamente, a expectativa é que sua execução produzirá os mesmos resultados.

Se o programa não compilar, talvez não tenha sido seguido o procedimento correto.

Verificar novamente se está usando os comandos corretos.

Também é possível que o código-fonte, se fornecido, seja um pouco diferente do que estiver no texto.

Examinar o código e compará-lo com o texto. Muitas vezes, será possível descobrir a discrepância.

Se a execução não produzir os mesmos resultados do livro, talvez tenham sido inseridos dados de entrada diferentes.

Novamente, é possível que o livro-texto tenha um erro tipográfico ou que o código-fonte distribuído para os programas de exemplo não seja exatamente o mesmo que está impresso no livro-texto. Se não conseguir isolar a discrepância facilmente, poderá simplesmente ignorá-la e tentar um exemplo diferente.

Se ainda não tiver sucesso após revisar cuidadosamente seus recursos, poderá ser necessário solicitar assistência externa de alguma pessoa experiente.

Testar programas de exemplo.

O próximo passo para aprender sobre um programa de exemplo é estudar como ele se comporta em diferentes circunstâncias.

Ao observar como ele responde a várias condições de entrada, poderá obter uma melhor compreensão do que o programa é capaz de realizar, bem como suas limitações.

Na exploração anterior, se conseguido executar o programa usando dados de entrada fornecidos no livro didático. Então, realizar seus próprios testes no programa.

Primeiro, deverá ter uma compreensão clara do que o programa **pretende** realizar.

Ler a explicação do livro didático novamente e certificar-se de ter uma imagem clara em sua mente do propósito do programa.

Em seguida, determinar que tipos de dados de entrada serão necessários e quais saídas serão produzidas.

Escrever um "plano de teste" em suas anotações de laboratório, escolhendo itens de dados de entrada apropriados e calculando manualmente os resultados esperados. Construir quantos casos de teste puder para criar uma ampla variedade de condições de entrada.

Poderá usar um formato de tabela como nos exemplos em anexo, para registrar o propósito de cada caso de teste, os dados de entrada e a saída esperada.

Em seguida, compilar e executar o código do programa.

Executar seu plano de teste inserindo os dados de entrada e registrando os resultados reais fornecidos pelo computador.

Se a saída real for diferente da prevista, anotar isso como uma "discrepância" (talvez marcando com um ponto de interrogação).

Quando tiver executado todos os seus casos de teste, revisar os resultados para avaliar se o programa está funcionando corretamente.

Escrever um "resumo de teste" no qual poderá comentar quais testes foram bem-sucedidos e quais falharam.

Tentar explicar a natureza ou a causa de cada "discrepância" da melhor forma possível.

Resumir o que você aprendeu sobre os recursos e limitações do programa.

Se houver aspectos do programa que precisem de reparo ou melhoria, poderá anotá-los como possíveis aprimoramentos para futuras explorações.

Examinar o código-fonte.

Quando estiver completamente familiarizado com o comportamento observável externamente do programa, deverá estudar o código-fonte responsável por produzir esse comportamento.

À primeira vista, isso poderá parecer sem sentido, mas, na verdade, cada faceta do código-fonte de alguma forma contribui para a operação geral do programa.

Por fim, se tiver um entendimento completo do programa, deverá ser capaz de explicar o papel que cada declaração desempenha na obtenção dos resultados do programa.

Começar lendo o código-fonte e ver o quanto dele faz sentido para você. Se quiser, poderá escrever uma observação na qual se tentará explicar cada declaração no código.

Escrever um breve resumo do que for capaz de compreender.

Não se preocupar se fizer pouco sentido no momento.
As explorações a seguir ajudarão a separar as peças do quebra-cabeça.

Poderá tentar identificar quais partes do programa correspondem ao seu comportamento observável.
Poderá determinar onde as entradas são obtidas e as saídas exibidas?
Quais partes serão responsáveis pelos principais cálculos?
Onde estão as estruturas de controle ou lógica?
Por enquanto, basta fazer uma lista de perguntas sobre tudo o que não estiver claro ou intrigante, ou sobre o que lhe parecer simplesmente curioso.

Introduzir erros.

Tentar intencionalmente introduzir erros em um programa de exemplo do livro.
Usar um editor de texto para fazer alguma pequena alteração superficial na sintaxe das instruções da linguagem, como omitir um parêntese, mudar uma vírgula para um ponto e vírgula, escrever uma palavra-chave incorretamente ou alterar a ordem dos elementos no programa.
As perguntas geradas na etapa anterior podem lhe dar ideias de coisas a serem tentadas.
Compilar o programa e observar se uma mensagem de erro é gerada pelo compilador.
Registrar cuidadosamente cada etapa em suas anotações, particularmente a mensagem de erro.
Sem dúvida, encontrará essas mesmas mensagens no futuro e, se as tiver escritas em suas anotações de laboratório, será muito mais fácil diagnosticar o problema.

Se não houver erros, executar o programa e observar qualquer alteração no comportamento.
Descrever os efeitos em suas anotações.
Nessa atividade e nas que se seguem, tentará fazer alterações sistemáticas no código e anotar cuidadosamente os resultados.
Poderá pensar nisso como um tipo de trabalho de detetive onde se estará reunindo pistas a partir das quais poderá começar a fazer inferências sobre os mecanismos subjacentes no programa.

Mudar a sintaxe.

"Sintaxe" significa a maneira como os símbolos são colocados juntos e se relacionam entre si em uma declaração. Refere-se à ortografia, pontuação e outros aspectos da forma das declarações, não ao seu significado subjacente.
À medida que começar a tentar entender o funcionamento do programa, os problemas mais óbvios a serem explorados serão os aspectos sintáticos do recurso de linguagem que estiver estudando.
Normalmente, as linguagens de programação não permitem muito desvio das regras para declarações corretas. Quase qualquer variação das regras precisas causará um erro, como deverá ter descoberto durante os experimentos de introdução de erros.
No entanto, às vezes é possível alterar a sintaxe para que **não** causar erro.
Um primeiro passo importante é determinar quais tipos de alterações de sintaxe serão legais e quais não serão.

Pensar em uma maneira de experimentar a sintaxe da declaração, alterando-a de alguma forma, mas ainda produzindo o mesmo resultado. A intenção é encontrar uma maneira diferente, mas ainda correta, de produzir o resultado desejado.

Por exemplo,

```
sum = count + 2 * last;
```

deverá dar os mesmos resultados que

```
sum = count + last * 2;
```

Similarmente,

```
for (int count =  1; count <= 10; count = count+1) { }
```

será repetido 10 vezes, e assim será

```
for (int count = 21; count <= 30; count = count+1) { }
```

Outro exemplo:

```
printf ("How now ");

printf ("brown cow.")
```

produzirá a mesma saída que

```
printf ("How now brown cow.")
```

Frequentemente, seu experimento poderá envolver a remoção de uma linha inteira de código.

Um atalho para fazer isso é transformar a linha de código em um comentário, usando a pontuação apropriada.

Por exemplo, em C/C++, colocar duas barras (//) no início de uma linha serve para indicar que o resto da linha não deverá ser lido pelo compilador. Mais tarde, poderá facilmente excluir as barras para restaurar o código à sua forma original.

Mudar semânticas.

"Semântica" se refere ao significado subjacente de uma declaração.

Na comunicação em linguagem natural, os humanos geralmente conseguem inferir o significado subjacente mesmo se a sintaxe estiver errada.

Normalmente, poderão ser ignorados erros de ortografia e pontuação e ainda ser possível entender a essência da frase.

Os compiladores insistem que a sintaxe esteja correta.

Em linguagem natural, é possível criar frases que são gramaticalmente corretas, mas cujo significado é ambíguo.

Felizmente, as declarações de programação são interpretadas apenas de uma maneira pelo compilador.

Depois de aprender as regras de sintaxe corretas para a linguagem, poderá avançar para aprender o que essas declarações significam.

Explorar a semântica da linguagem é como se obtém compreensão da maneira como o compilador interpreta cada declaração e qual efeito essa declaração tem na execução do programa.

Experimentar alterar uma declaração para que, embora ainda sintaticamente válida, possa produzir resultados diferentes.

Prever a saída e executá-la para observar o resultado real.

Certificar-se de registrar seus resultados, pois é crucial entender os tipos de alterações que o compilador **não** sinalizará como erros.

Por exemplo, alterar

```
Celsius = 5.0 / 9.0 * (Fahrenheit - 32.0);
```

para

```
Celsius = 5.0 / 9.0 * Fahrenheit - 32.0;
```

Uma implicação importante desses experimentos é que o compilador não tem como verificar se o significado semântico é o que se pretendia.

A segunda fórmula acima é sintaticamente correta e produzirá um programa funcional, mas a semântica não realizará a conversão de temperatura adequada

É importante ter em mente que um programa que não produz erros do compilador não está necessariamente correto.

Usar sinônimos de código.

"Sinônimo" implica ter o mesmo significado.

Nessa exploração, o objetivo é mudar a semântica do programa para que uma abordagem diferente seja usada para produzir os mesmos resultados.

Apesar do fato de que o compilador é incrivelmente exigente sobre a sintaxe, geralmente é possível encontrar mais de uma maneira de atingir um resultado pretendido. Por exemplo, às vezes poderá alterar a ordem das instruções sem alterar o resultado geral. Às vezes, poderá pensar em um atalho ou uma maneira alternativa de codificar a solução que fornece resultados equivalentes.

Por exemplo, em C++, poderá encontrar o quadrado de um número multiplicando-o por ele mesmo ou usar a função '**pow**(x,y)' da biblioteca matemática.

```
quadrado = número * número;
```

ou

```
quadrado = pow(número,2);
```

Outros exemplos:

```
volume = 3.14 * raio * raio;
```

ou

```
volume = (3,14 * (raio*2.0) ** 2.0) / 4.0;
```

Observar que os sinônimos de código são semelhantes às explorações iniciais de "mudanças de sintaxe". A diferença é que os sinônimos de código usam uma semântica diferente, mas chegarão à mesma saída. Se essa distinção não estiver clara, não se preocupar. Não importará muito. Contanto que esteja criando ideias para seus próprios experimentos, siga em frente.)

Modificar código.

Inventar alguma variação menor que envolva alterar o código (sem adicionar nada novo).

Poderá simplesmente seguir sua curiosidade ou consultar a lista de perguntas criadas anteriormente. Não tem problema em experimentar, mudar as coisas sem nenhum propósito específico, apenas para ver o que acontece.

Tentar ter uma noção de quais mudanças serão cosméticas ou não essenciais e quais serão mais significativas.

Algumas de suas mudanças poderão produzir erros de sintaxe, outras não.

Ao fazer uma modificação, registrar sua previsão de qual efeito ela terá na compilação ou na execução do programa.

Em seguida, comparar os resultados reais com sua previsão.

As mudanças semânticas feitas anteriormente também são modificações de código, embora provavelmente sejam estreitas em escopo. A ideia com modificações de código é ampliar um pouco seu foco e lidar com declarações inteiras ou mesmo sequências de declarações. (Se já fizer isso, tudo bem.)

Alguns exemplos podem incluir alterar a aparência ou o formato da exibição de saída ou a ordem em que a entrada é inserida. Poderá alterar o valor inicial atribuído às variáveis ou alterar os cálculos em uma fórmula.

Depois de aprender sobre estruturas de controle, poderá alterar o número de vezes que uma repetição executa ou alterar uma declaração de decisão para que ela fazer o oposto do que é pretendido.

Poderá experimentar alterar o tipo de dados numéricos de inteiro para flutuante.

Exemplos de modificações de código:

Alterar: if (HoursWorked > 40) { }

para: if (40 > HoursWorked) { }

Alterar: while (Today /= Saturday) { }

para: while (Today < Saturday) { }

Alterar: scanf(%s %d", Name, &Age);

para: scanf("%d %s", &Age, Name);

Experimentar soluções alternativas.

"Quem não tem cão, caça com gato."

Como a velha máxima indica, geralmente pode haver várias soluções para certo problema.

Depois de começar a aprender sobre diferentes recursos de linguagem, tentar imaginar uma abordagem diferente que alcance o mesmo resultado. (Novamente, a diferença entre "sinônimos de código" e "soluções alternativas" não é importante. A intenção aqui é tentar mudanças mais ambiciosas para envolver diferentes recursos de linguagem.)

Por exemplo, usar uma repetição **while** em vez de uma com **for** para controlar o número de repetições.

Experimentar melhorias no código.

Inventar alguma pequena melhoria no programa que envolva adicionar novas instruções, mas que não afete a estrutura geral do programa. Por enquanto, evitar mudanças que alterem a lógica subjacente da solução. Claro que há muitas melhorias possíveis, então deverá se concentrar em experimentos que o ajudem a entender o recurso de linguagem demonstrado pelo programa de exemplo.

Abordar esta tarefa de forma sistemática, registrando em suas anotações de laboratório o que pretende realizar, como escolheu fazer isso, quais problemas encontrou e quais resultados observou. Certificar-se de ter um plano de teste, para que saiba se sua solução produzirá os resultados corretos.

Exemplos:

Melhorar um programa que peça a altura de uma pessoa, para que ele obter também o peso da pessoa.

Melhorar um programa que calcule a área de um círculo, para que ele também calcule o volume de uma esfera com o mesmo raio.

Melhorar um programa que determine o dia mais quente do ano a partir de uma lista de temperaturas, para que ele também calcule o dia mais frio.

Melhorar um programa que calcule um total de alguns valores para que ele também calcule a média desses.

Melhorar um programa para que ele seja mais flexível, para que valores previamente codificados como constantes possam ser inseridos pelo usuário.

Melhorar um programa que dependa de entrada em letras minúsculas, para que ele não faça distinção entre maiúsculas e minúsculas.

EXPLORAÇÕES AVANÇADAS

À medida que começar a ganhar confiança em suas habilidades de programação, poderá empreender explorações mais sofisticadas.

Adicionar melhorias no programa.

Melhorar o programa para que alguma nova funcionalidade seja adicionada. Isso geralmente requer retrabalho da lógica da solução, alterando ou aprimorando o algoritmo subjacente.

Certificar-se de planejar essas mudanças no papel primeiro e verificar sua lógica cuidadosamente.

Em seguida, traduzir algoritmos em código de programa.

Criar um plano de teste para verificar se suas mudanças estarão corretas.

Exemplos:

Alterar um programa que aceita um número fixo de itens de dados de entrada para que o usuário possa especificar quantos itens de dados devem ser inseridos.

Executar a verificação de validade nos dados de entrada para garantir que seja o intervalo adequado.

Alterar um programa que recebe todas as entradas de um arquivo de dados para um que interaja com o usuário para obter a entrada.

Alterar um programa que depende de entradas "codificadas" (como 23456SD para dias da semana) para poder aceitar o nome real, por exemplo, "segunda-feira".

Melhorar um programa para que, quando os cálculos forem concluídos, pergunte ao usuário se ele gostaria de executar outra execução antes de terminar.

Melhorar um programa para que o usuário tenha a opção de salvar os resultados em um arquivo de dados.

Converter um programa que exibe gráficos de barras verticais para exibi-las horizontalmente.

Adicionar um "menu" interativo a um programa para que, em vez de executar suas funções de maneira sequencial, o usuário possa selecionar quais operações executar.

Reformular o programa.

"Reformular" um programa é transformá-lo de forma que ele execute uma tarefa diferente. Isso implica em uma grande revisão. É bem provável que a estrutura geral e a lógica do programa sejam alteradas.

Exemplos:

Reformular um programa que calcula uma conta de água para que calcule a conta de luz.

Reformular um programa que paga um vendedor por hora para um em que o pagamento é baseado em comissão.

Reformular um programa que determina as notas dos alunos com base em uma curva para um em que a nota é baseada em uma porcentagem direta.

Reformular um programa que auxilia na preparação de formulários de impostos federais do usuário para que ele prepare os formulários de impostos estaduais.

Reformular um programa que procura itens em um arquivo de inventário para que ele possa atualizar itens no arquivo.

Remontar o programa.

Ao construir programas maiores, é desejável não "reinventar a roda". Quando possível, é bom aproveitar o código já escrito. A "reutilização" de software é uma estratégia para montar partes de programas diferentes para criar um novo programa. Componentes de software pré-escritos geralmente existem em bibliotecas para tarefas comuns, como funções matemáticas, manipulações de texto, gráficos e assim por diante.

Para essas explorações, poderá experimentar incorporar rotinas pré-escritas em seus próprios programas.

Poderá simplesmente invocar funções de uma biblioteca ou poderá "recortar e colar" código de programas de exemplos no livro didático.

CONCLUSÕES

Quando terminar suas explorações, é importante reservar um tempo para refletir sobre o que aprendeu.

É útil escrever um parágrafo conclusivo que resuma os principais pontos investigados e o que aprendeu com suas explorações.

O propósito dessas explorações de laboratório é ajudá-lo a dominar a sintaxe e as regras semânticas para programas escritos corretamente.

Não há um número necessário de exercícios para se trabalhar, ou um conjunto específico de problemas atribuídos para completar.

Em vez disso, essas diretrizes serão abertas para permitir que possa explorar os tópicos mais interessantes, ou investigar ideias que achar difíceis.

Cada programa de exemplo que investigar é projetado para ilustrar certos recursos de linguagem.

Suas explorações deverão se concentrar nos principais recursos do exemplo, e deverá continuar experimentando até entender o uso adequado do recurso de linguagem que estiver sendo ilustrado.

INSTRUÇÕES PARA RASTREAMENTO MANUAL

Objetivo:

Um rastreamento de código é um método para simular manualmente a execução do seu programa para verificar se ele funciona corretamente antes de compilá-lo. Também é conhecido como "rastreio de código" ou "verificação manual".

1. Desenhar uma tabela com cada variável no programa mostrada no topo da coluna.
   Desenhar uma coluna para o número da instrução à esquerda.
   Desenhar também uma coluna para a saída.

2. Numerar cada instrução executável no código.

3. Começar com a primeira instrução executável, simular a ação que o computador tomará ao executar cada instrução.
   Na coluna apropriada, escrever o valor atribuído a essa variável (ou saída produzida).
   Continuar dessa maneira, escrever a ação de cada instrução em uma nova linha.

4. Se dados de entrada forem necessários, usar as entradas de um dos seus casos de teste dos seus planos de teste.
   Rotular sua tabela para indicar qual caso de teste estará usando.

5. Quando o rastreamento estiver concluído, verificar se a saída produzida corresponderá à saída esperada em seu plano de teste.

Exemplo:

Abaixo está uma tabela de rastreamento de código para o programa Ounces.c mostrado a seguir.

Linha	*ounces*	*cups*	*quarts*	*gallons*	Saída
16					Inserir o número de onças:
17	9946				
20		1243			
21			310		
22				155	
25					9946 *ounces*
					1243 *cups*
					310 *quarts*
					155 *gallons*

```
01  /* Programa: Ounces.c
02   * Resumo  : Converter oncas em xicaras, quartos e galoes
03   *                O usuário devera' inserir o numero de oncas.
04   * Autor       : J. Dalbey 11/10/1997
05   */
06  #include <stdio.h>
07
08  int main (void)
09  {
10      int ounces;  /* entrada - numero de oncas       */
11      int cups;      /* saída    - convertida em xicaras */
12      int quarts;   /* - convertida em quartos          */
13      int gallons;  /* - convertida em galoes           */
14
15   /* obter dados de entrada */
16      printf("Inserir o numero de oncas: ");
17      scanf("%d", &ounces);
18
19   /* calcular conversoes */
20      cups    = ounces  /  8;
21      quarts  = cups      /  4;
22      gallons = ounces / 64;
23
24   /* exibir resultados */
25      printf("%d ounces = \n%d cups\n%d quarts\n%d gallons\n",
26                    ounces,          cups,       quarts,       gallons);
27      return 0;
28  }
```


Referências

https://users.csc.calpoly.edu/~jdalbey/101/resources.html
https://users.csc.calpoly.edu/~jdalbey/101/Lectures/HowToStudy101.html
https://users.csc.calpoly.edu/~jdalbey/101/Resources/exploring.html
https://users.csc.calpoly.edu/~jdalbey/101/Resources/analysis.html
https://users.csc.calpoly.edu/~jdalbey/101/Resources/codetrace.html