Checklist voltado ao C/C++

Guilherme Braga Pinto
Universidade de
Brasília
Engenharia de Computação
Brasília, DF
Matrícula: 17/0162290

***Resumo*—Este trabalho tem como finalidade fazer uma "checklist" para projetos de C e C++ em diferentes níveis. Aborda-se conceitos relacionados às formas de programação (aproveitamento de tempo e métodos de se evitar "bugs") e bons costumes para programas bem otimizados. Segundo trabalho da matéria "Métodos de Programação"(201600), 2/2018, Professor Jan M. Correa. Entrega do trabalho prevista para dia 26/09/2918. Documento editado via LaTeX.**

I. Como fazer levantamento de requisitos

A. Compreensão da área de aplicação

É de vital importancia entender o contexto de onde será executado um programa, o software de uso doméstico e o software diretamente aplicado em uma empresa podem ser diferentes. E podem existir lógicamente diferenças nas aplicações entre empresas que necessitariam de um mesmo programa.

B. Interação com os stakeholders do sistema

Stakeholder significa "público estratégico", e traduzindo a palavra livremente do inglês, estes são os "possuidores de risco". São as partes (pessoas ou empresas) cujo o programa produzido afetará diretamente. Definir estratégias e objetivos com esse grupo é crucial para que o programa seja útil, é uma medida que prevê a otimização da funcionalidade do programa na teoria.

C. Definir prioridades

Lógicamente esse ato prevê a boa alocação de recursos na hora de se fazer um programa. Depenendo da tarefa e do tempo que se tem disponível para programar, definir prioridades irá otimizar o uso do tempo. É importante gastar mais tempo nas partes mais complexas ou mais relevantes de um código, afim de se evitar bugs.

D. Assegurar clareza nos requisitos

Requisitos ambíguos impossibilitam o programador de fazer um código que atenda às necessidades desejadas. Sem requisitos bem definidos, o programador se verá obrigado a assumir por conta própria diversos aspectos do programa, e o que pode ser formulado à partir disso difícilmente será o objetivo idealizado palo formulador que não fez uma lista de requisitos decente.

E. Analisar os diferentes perfis de usuários do programa

O método VORD (viewpoint-oriented requirements definition, em português, definição de requisitos orientada a ponto de vista) foi projetado como um framework (provê diferentes códigos para uma funcionalidade generalizada) orientado à serviço da análise de requisitos. Em suma, diferentes usuários tendem à utilizar o mesmo programa de formas variadas, e a idéia é atender a todos de forma funcional.

F. Workshops

O Workshop se diferencia de uma reunião comum entre os participantes de um projeto por propor a interação entre diferentes partes, assim mais pontos de vista são difundidos se comparado com uma simples reunião em moldes antigos (um palestrante e uma apresentação de slides pode não manter a atenção de todas as partes, uma apresentação de teor mais técnico pode não ser relevate para o usuário).

G. Brainstorming

Muitas vezes sendo a técnica abordada dentro de um workshop, o brainstorm (livremente traduzido para "toró de idéias") é uma maneira de se agregar novas idéias para a produção de um programa. Se leva em consideração todas as idéias de uma equipe, por exemplo, para assim se ter a noção de que as melhores sugestões podem chegar a serem usadas.

H. Mapa mental

É uma forma de se estruturar uma linha de pensamento. Partindo de uma idéia específica, uma linha de raciocínio na criação do software não terá as mesma implicações de qualquer outra idéia. Essa estrutura ajuda a visualizar implicações ao longo prazo, não só ajudando a estruturar os passos do desenvolvimento mas também ajudando na tomada de decisões. Pode ser muito relevante se o trabalho em equipe é um foco, o mapa mental não só apresenta como também justifica uma decisão com base em uma melhor alternativa. Explicar decisões ajuda um programador a elaborar o código.

I. Criação de versões de teste do programa (prototipagem)

Testar o programa (com o usuário, de preferência) significa fazer uma checagem constante do rumo do projeto. Implementações equivocadas podem ser rápidamente corrigidas antes

de se atingir um estágio avançado de desenvolvimento, e a percepção da necessidade de novas funcionalidades ocorre em estágios iniciais de desenvolvimento. Nesse caso, é uma medida de se corrigir ambiguidades geradas por um levantamento de requisitos mal formulado.

J. Questionários

Questionários são uma forma direta de definir as necessidades de um código buscando diretamente na fonte do sucesso, a base de usuários. Se o usuário do programa em questão não está satisfeito, logo não há razão para se promover seu trabalho como programador (que está sendo representado por uma aplicação incoerente, não necessariamente mal produzida).

K. Avaliação de cenários dinâmicos

Avaliar diferentes cenários representa estar um passo à frente de um problema, ou seja, avaliar as possibilidades do uso de um software mesmo que não represente algo diretamente previsto. O elemento que não foi previsto não representa simplesmente um problema independente de sua complexidade, mas sim uma situação de adaptabilidade que pode ou não vir a ocorrer.

L. Definição de requisitos não-funcionais

Basicamente, essa etapa tem como finalidade definir com stakeholders ou usuários uma série de requisitos do programa que não se relacionam diretamente com a sua funcionalidade (requisitos funcionais, que no geral são o maior foco no trabalho executado). Por exemplo, o programa deve ser portado ou compilado em diferentes sistemas (diferentes Windows, diferentes distribuições de Linux ou difertentes versões do MacOS)? O programa deve ser utilizado por quanto tempo? Qual deve ser o número médio de falhas? A máquina onde o programa será utilizado deve ter um requisito mínimo de hardware? Pois elaborar um programa muito pesado pode não ser a melhor solução. Arquivos de texto devem ser entregues no formato .pdf para melhor leitura ou .docx (Microsoft Word) para melhor edição com as ferramentas do Office? Os usuários necessitam de outros programas para fazer o seu programa principal funcionar? O uso de software proprietário pode aumentar os custos para o usuário, podendo ou não ser a melhor solução.

M. Pesquisa direta

Tentar entender como outras pessoas lidaram com a idéia do código, como outras pessoas aplicaram diferentes idéias para uma mesma solução. Não resulta em necessariamente ver outros códigos a um baixo nível de abstração, ou buscar outros programas que façam exatamente a mesma coisa que se deseja que seu programa faça. Ver programas parecidos ou que apresentam apenas partes relevantes para a sua aplicação podem ajudar na elaboração do projeto.

N. Conhecimento de um sistema antigo

Conhecer uma antiga forma de se aplicar um programa pode ser crucial para não se repetir os mesmos erros. Assim não é necessário "reinventar a roda" a cada aplicação da mesma idéia. Caso se trabalhe com uma atualização ou reformulação do software de uma empresa, é bom checar a solução que se usava antes de se encomendar o software.

O. Estudo etnográfico

É uma análise do "componente social" das tarefas desempenhadas numa dada situação. Inclui verificar a necessidade de requisitos que não são exatamente perceptíveis com o uso de outros métodos, se busca entender a rotina de uso do usuário de um programa do ponto de vista de um observador para possivelmente facilitar uma funcionalidade.

II. Como fazer a especificação de requisitos

A. Análise do levantamento de requisitos

Préviamente desejou-se levantar motivações e resultados, na elaboração da especificação dos requisitos a prioridade é definir os meios que serão utilizados. A especificação é a forma de comunicação direta entre um analista e os envolvidos no desenvolvimento do software.

B. Lidar corretamente com domínios desconhecidos

Se a aplicação do software será feita em um ambiente específico, deve-se usar ferramentas para tais meios. Por exemplo, se um software lida com imagens de raio-x, deverá ser utilizado um formato para esta aplicação em especial, e não qualquer ".png" que seja melhor do ponto de vista do programador. O usuário deve ser o foco independente de sua necessidade específica.

C. Definição de papéis

Em uma equipe, deve-se definir os papéis de cada programador em base especificação do projeto. É normal que cada participante tenha uma preferência (aptidão) específica ao se escolher o que vai programar.

D. Elaboração formal da documentação

Não existe uma forma exatas para a documentação (onde se explica o programa), mas existem diversos modelos. Na documentação, explica-se desde o fluxo dos dados e a abordagen que foi utilizada até as limitações do programa e outros diversas descrições do trabalho (descrição de estados, descrição comportamental, entre outros).

E. Boa conduta do responsável pelos requisitos

É relevante que o encarregado direto dos requisitos seja primeiramente um ouvinte. A pessoa deve estar atenta aos detalhes de elaboração e deve aceitar o diálogo quando necessário, suas decisões afetam o projeto antes mesmo de se começar a programar.

F. Evitar extrapolações

Pedidos exagerados e especificações redundantes não colaboram com o projeto, muito pelo contrário, servem para atrasar o programador e agrega possíveis ambiguidades.

III. Como fazer o design de software

A. Interpretação de protótipos

É vantajoso o teste de protótipos com os usuários, ou mesmo no ambiente de criação do software. É elaborar uma espécie de "ponto de controle" para boa manipulação do resultado final. Nesse caso se deseja analisar a interação entre o usuário e um programa via prototipagem.

B. Criação do melhor modelo conceitual

Deve-se elaborar como o programa deve funcionar na teoria, montando uma espécie de "esqueleto" de sua estrutura, para então poder executar o trabalho necessário.

C. Pesquisa com o usuário

Design de software esta relacionado diretamente com o ato de definir a experiência que será entregue ao usuário, logo ouvir suas reclamações e pedidos é crucial para saber como será a rotina do programa. A pesquisa desde o início do projeto é crucial para elaborá-lo.

D. Definir as estruturas de dados

Previamente definir as estruturas utilizadas faz parte do design de software, por uma questão de planejamento para otimizar a programação.

E. Definir os algoritmos

Previamente definir os algoritmos que serão utilizados ajuda o programador a elaborar com mais detalhes os tipos de operação que serão implementados.

F. Definir relações entre os recursos utilizados

Deve-se ter bem claro como diferentes funções e como diferentes registros trabalham entre si. É vantajoso definir uma hierarquia se necessário. Por exemplo, em um programa de uma Pilha, definimos que as funções dependem diretamente uma struct que guarda os atributos necessários para armazenar o obejto de interesse.

G. Elaboração da Interface

O design tem também a finalidade de definir como o usuário de relacionará com a aplicação final, ou seja, o meio de interação do usuário. Para isso é necessário elaborar esse conceito a partir do perfil de quem utiliza o programa (não adianta elaborar um programa que executa suas necessidades básicas, mas de forma ruim, é basicamente uma forma de se solucionar um problema criando outro problema). Muitas vezes, algo mais simples e funcional pode ser melhor do qu euma aplicação complexa e lotada de recursos.

IV. Como fazer o código

A. Domínio da linguagem

Executar um código grande em uma linguagem que o programador recém descobriu pode não ser uma boa idéia. Muitas vezes simplesmente pensar um código em uma linguagem e escreve-lo em outra pode implicar em resultados horríveis e totalmente não esperados. Essa idéia é uma das bases para o Linux (kernel) ser escrito em C e não C++. Linus Torvalds diz que o uso do C++ para essa aplicação fará com que o código dependa de um modelo de abstração maior. Se esse modelo se mostrar menos eficiente, corrigir o problema será algo mais complicado do que deveria.

B. Modularização

Modularizar o código pode implicar em diferentes resultados dependendo do contexto (até mesmo da linguagem). Para o C/C++, modularizar representa a criação de funções e diferentes arquivos bem estruturados para a aplicação de maior relevância, o programa final. Módulos podem ser compilados de forma independente, não só facilitando o entendimento como agilizando o processo em caso de correção de bugs. Uma modularização básica e bem executada se dá por separar as funções auxiliares usadas na função "main" em outro arquivo (".cpp") junto à um arquivo adicional com os protótipos das funções (".hpp"), podendo até agrupar as funções de diferentes finalidades em diferentes arquivos (acentuando a hierarquia e a ordem entre as partes do programa).

C. Reutilização de códigos que já foram testados

Reutilizar códigos não deve servir de base para um novo código, porém, o uso de algoritmos especificos que já foram previamente testados é uma medida de diminuir possíveis bugs. Por exemplo, se desejamos organizar números em ordem crescente, pode não ser lógico inventar um novo método de organização se já é amplamente conhecido e debatido métodos como o BubbleSort e o InsertionSort. Nesse exemplo, existem diversos algoritmos, e a escolha de um algortimo já conhecido pode representar uma otimização significativa no tempo de execução, dependendo do conjunto de números que se deseja organizar.

D. "Defensive Programming"

É uma forma de se programar onde é de vital importância garantir o funcionamento do software em cenários emprevisíveis. Porém, exagerar nesse princípio pode resultar em má otimização do código (já que se programou um grande número de exceções).

E. "Offensive Programming"

Uma forma e um conjunto de práticas de "programação defensiva" visando especificamente erros externos ao programador. Ou seja, esse método de programação tenta minimizar erros gerados pelo input do usuário. Muitas vezes se parte do princípio que o código já está funcionando bem, já que apoenas queremos tratar externalidades. Por exemplo, se estamos trabalhando com um programa que calcula a velocidade

de um objeto no contexto de uma aceleração constante e invariável, não deixar que o usuário trabalhe com medidas de tempo negativas é uma medida de "Offensive Programming", solucionando um erro antes de um bug vir a ocorrer.

F. "Secure Programming"

Uma forma e um conjunto de práticas de "programação defensiva" visando especificamente a otimização da funcionalidade do código, visando continuamente checar a segurança do computador em si. Todo tipo de Programação Defensiva tem como objetivo evitar a criação de bugs, mas nesse caso a motivação é o princípio de que o programa será executado da pior forma possível, para gerar novos bugs que podem ser explorados de formas ilícitas. Por exemplo, uma função que pode utilizar toda a memória da máquina com alocação dinamica (resultando no congelamento das funções do sistema) não representa uma boa prática de "Secure Programming".

G. Idealizar um projeto em Open Source

Idealizar um projeto em Open Source ("Software Livre") pode ser benéfico dependendo da aplicação do programa que se deseja criar, e dependendo de quem já se encontra na equipe (uma equipe pequena que trabalha po rconta própia pode se beneficiar). Se a aplicação final do programa é um bem em comum com outros usuários e programadores, trabalhar com o código aberto pode ser crucial para agilizar o produto final que se deseja estebelecer. Uma pequena equipe de programadores pode não executar um trabalho tão bem quanto uma comunidade dedicada à acabar com um problema. O Software Livre acaba nascendo mais de uma necessidade grande de um grupo de pessoas do que de o pedido de uma aplicação específica, logo para uma empresa pode não ser tão lucrativo, ou não ser lucrativo de forma alguma. Todavia é importante ressaltar que esse método não implica em distribuir trabalho sem algo em retorno, ou trabalhar de forma anônima já que o código se torna público para qualquer pessoa, logo é de vital importância definir uma licença de distribuição adequada. As diferentes licenças (GPL e MIT, por exemplo) resultam em diferentes resultados, diferentes aplicações e diferentes redistribuições do seu trabalho.

H. Especificar corretamente suas variáveis

Criar variáveis que não são "auto-explicativas" é um péssimo costume. Não só para o programador (que eventualmente irá se perder em meio à diferentes variáveis sem nome lógico) como também para outras pessoas que lerão o código (que terão que adivinhar o que os nomes abstratos representam). Nomes ruins nesses casos são, por exemplo, variáveis de apenas uma letra (variável X, variável Y) ou pior, uma mesma letra com diferentes números para diferenciação (variável X1, variável X2, variável X3). Um código com variáveis assim, dependendo de sua complexidade, será praticamente impossível de se corrigir por alguém que não seja efetivamente o programador.

I. Nomear funções corretamente

O nome de uma função deve ser autoexplicativa. Ela deve ser clara ao falar o que a função faz, por exemplo, uma função que organiza números deveria explicar o algoritmo que foi utilizado. Chamar uma função de "bubblesort" ao invés de "organizanum" agrega ao entendimento do código.

J. Não alocar memória excessivamente

O uso de alocação dinamica é uma boa prática de progrmação, é como reservamos um espaço de memória ainda não definido para uma variável. Porém, se usado em excesso, toda essa memória que foi reservada poderá atrapalhar a execução do programa. E também, é sempre necessário checar se ao final do código todo o espaço alocado já foi liberado (comando free).

K. Tomar cuidado com a quantidade de variáveis auxiliares criadas

Normalmente não existe a necessidade de se criar uma variável de "contador" para cada loop "for" que existir no programa. O hábito de se criar muitas variáveis desnecessárias resultará em um código ambíguo e mal formulado, de difícil leitura e certamente mal otimizado.

L. Identar o código

A identação tem como objetivo ressaltar uma estrutura lógica no código, para sabermos que linhas de código estão em quais estruturas (amplamente usa-se a tecla TAB em editores de código para esta finalidade). Um código "embaralhado" dificilmente será corrigido propriamente antes de ser arrumado para uma devida leitura. Algumas linguagens de programação-são sensíveis à identação, como o Python, diferentemente do C/C++ (dispensando o uso de colchetes). Logo muitas vezes não só é uma questão de manter um bom costume, mas a identação pode ser crucial para o funcionamento decente do código.

M. Evitar comando "goto"

O comando "goto" basicamente acaba por criar diversos atalhos no código. Para um programador, isso implica em ir para frente e para trás no código dependendo dos diversos "goto" que foram usados, dificultando o entendimento e dificultando uma possível otimização do código posteriormente. A idéia de se evitar esse comando em C é amplamente difundida pela comunidade.

N. Avaliar o sinal das variáveis

Algumas variáveis não apresentam a necessidade de serem negativas (uma variável que representa valores absolutos, como distância). Uma declaração "unsigned" é estratégicamente melhor nesse caso.

O. Avaliar o tipo das variáveis

Diferentes declarações de variáveis implicam em diferentes tamanhos de memória ocupados. Caso não seja necessário o uso de muitas casas decimais, declarar uma variável como "float" e não como "double" é melhor para se otimizar o código.

P. Evitar repetições que poderiam ser implementadas com loops

Muitas partes parecidas e repetidas em um código podem ser adaptadas em um loop (uma grande estrutura de "while" ou "for" com divisões de condicionais, por exemplo).

Q. Valores que não mudam ao longo da execução

Podemos mudar esses valores para variáveis constantes, porém, se evitando o uso do "define" (global). Se determinada variável é criada, porém, não utilizada de forma alguma, ela deve ser deletada.

V. Como comentar o código, "Design by Contract"

A. Idéia de "Design by Contract"

É uma abordagem em que se deve ter "contratos" pré-estabelecidos entre componentes do programa, ou seja, uma espécie de planejamento de software onde se estabelece pré-condições, resultados e hierarquia entre meios utilizados.

B. Evitar redundância ao explicar funções

Funções devem explicar sozinhas sua finalidade, sem a necessidade de um comentário. Comentários nesse caso são relevantes para explicar "como" se aplica uma idéia, mas não o que é a idéia em si. Por exemplo, um comentário que diz "essa função converte números binários para números decimais" ao lado de uma função chamada "convertebinpradecim" é algo redundante que não ajuda o entendimento do programa.

C. Evitar explicar cada variável criada

Variáveis devem ser nomeadas de forma que dispensem de uma linha de explicação. Se cada variável apresentar uma respectiva explicação, muito espaço será ocupado por nada de realmente relevante.

D. Evitar "poluir visualmente" o código com comentários

O código fonte não é o lugar apropriado para escrever longos textos explicativos sobre o trabalho feito, por isso a documentação se faz em arquivos independentes (com a ajuda de softwares propriamente elaborados para esta aplicação).

E. Manter os comentários simples e diretos

Se a explicação do comentário for redundante, então o mesmo deve ser desconsiderado. Comentários devem explicar estruturas de código de forma direta e simples unicamente para se otimizar o entendimento do que foi escrito.

F. Comentar códigos em inglês para alcançar maior público

Dependendo de onde que o código irá circular, comentar o código em ingles facilitará a colaboração de outros programadores, e poderá servir de referência futura em trabalhos internacionais.

G. Uso de delimitadores para comentários: / */*

Caso o comenário tenha um tamanho considerável, escrever repetidamente utilizando "//" pode simplesmente não ser uma boa escolha visualmente, apesar de ter o mesmo resultado de um comentário com delimitadores.

H. Informar dados básicos para referência futura

O arquivo onde se encontra o código não é o lugar correto para se escrever uma documentação, lógicamente, porém dados básicos (nome do programador, idéia resumida do que o código faz, data de início do projeto) são boas informações a se declarar (desde que isso não ocupe um espaço exagerado).

I. Comentar não apenas explicações, mas também trechos antigos do código em partes do desenvolvimento

Ao tentar corrigir uma função, declarar um código ao invés de apagá-lo pode trazer benefícios posteriormente, em especial se a posssível solução (nova implementação) de um bug acabou por piorar um resultado final ao invés de corrigi-lo. Porém, entregar na versão final de um programa vários trechos de códigos antigos não apresenta vantagem alguma.

J. Separar "blocos" de código usando comentários

Se um arquivo apresenta várias funções de diferentes finalidades juntos, separar em blocos pode ser uma boa escolha. Por exemplo, se em um mesmo arquivo tipo "headers" se declara funções de Pilha e de Fila, "dividir em grupos" as declarações com uma linha de comentário pode ajudar visualmente a entender todo o trabalho que foi feito, e ajuda a localizar uma função desejada dependendo de sua aplicação.

VI. Como testar o código

A. Uso do Gcov

O Gcov é uma ferramenta para análisa da execução efetiva do código, ou seja, a ferramenta executa a cobertura do código e gera um documento com as contagens exatas de cada linha feita pelo programador. Uma ferramenta de vital importância para checar a efetividade do código. Gcov vem como um utilitário padrão com o pacote GNU Compiler Collection (GCC).

B. Desenvolvimento orientado à testes: Google Test

O Google Test (gtest) é uma ferramenta usada no desenvolvimento de software orientado a testes. Basicamente a ferramenta checa o resultado de uma função ou um código com um resultado préviamente previsto pelo programador. Assim, se checa se todas as funções de um grande projeto estão funcionando de forma esperada.

C. Desenvolvimento orientado à testes: catch.hpp

Esta também é uma ferramenta de testes, mas diferente do gtest, o Catch é utilizado por meio da adição de um arquivo tipo "headers" ao código, podendo facilitar a portabilidade do código (em comparação, para o Google Test funcionar, é necessária uma instalação prévia de um repositório específico na máquina, o que pode ser difícil para usuários de Windows).

D. Valgrind

Valgrind é um software livre que auxilia programadores à checar vazamentos de memória. Ou seja, checa se a alocação de memória foi bem executada, e se ao final foi liberado o espaço reservado. O Valgrind não faz uso de outras bibliotecas auxiliares para um bom funcionamento, assim seu uso para o programador é simplificado.

E. Compartilhamento local de software (PrivateBin ou PasteBin)

Isso inclui pedir uma segunda opinião sobre determinada aplicação em caso de dúvidas, não necessáriamente abrindo em definitivo o código para que qualquer pessoa possa modificar a fonte. O uso do PasteBin pode encurtar distancias, assim a pessoa que se deseja consultar não está necessariamente presente no ambiente de criação do software. O PasteBin é uma ferramenta de compartilhamento temporário de código, sendo a ferramenta mais usada desde 2002. O programador pode copiar o código e envia-lo usando apenas um link para um terceiro de confiança. A ferramenta de leitura funciona online em, browser, e o fornecedor tem acesso ao número de visualizações do link que foi gerado por tempo limitado. Uma boa alternativa para este serviço é o PrivateBin, onde segurança é a prioridade. Além de ser uma boa alternativa em Open Source, o servidor do site não tem informação sobre o que está sendo compartilhado. Outras ferramentas opcionais estão disponíveis, como o uso de senhas e de funções de "burn after reading", onde o receptor da mensagem pode ler o código apenas uma vez para rápida revisão.

F. Teste de entradas supostamente inválidas

O teste de entradas inválidas se dá para ver se toda a precaução que se teve ao longo do projeto foi bem idealizada. Por exemplo, deve-se testar números inconcebíveis para uma aplicação para entender como o programa irá lidar com isso. Se o seu programa simula um elevador com diferentes lógicas para a entrega dos passageiros em diferentes andares (First in, First Out ou Shortest Job First), como ele irá lidar com um elevador que tem 8 milhões de andares (um número não realista, se não impossível)?

G. Uso de funções que medem o tempo de execução do programa

O uso dessas funções ajudam a entender a complexidade de um programa, assim facilitando na otimização quando possível, e ajudando a entender as estruturas que foram empregadas.

H. Avaliação de variáveis

Checar o comportamento das variáveis ao longo da execução do código. Eventualmete se cria mais variáveis do que o necessário, o que pode gerar em danos à performance geral da aplicação.

I. Execução em outra máquina

Ajuda a identificar as depenências do código. Um programa pode não ser relevante se ele funciona apenas em um ambiente muito específico.

VII. Como depurar o código

A. Sem a ajuda de um programa: funções de "printf"

É a forma mais arcaica e menos funcional de se corrigir um bug. Porém, em aplicações simples, testar se um condicional é satisfeito por meio de mostrar um print pode ser a solução mais rápida. É importante manter em mente que esse método é ruim para programas grandes e é algo de muito baixo nível e pouco "profissional". Problemas complexos exigem medidas complexas.

B. GDB

O GDB (GNU Debugger) é um depurador de código do GNU, usado em sistemas baseados em Unix (funciona nativamente no Terminal). É a ferramenta mais tradicional para se corrigir programas em C/C++, normalmente é a alternativa recomendada caso se tenha acesso à uma máquina com Linux.

C. Ferramenta de depuração nativa do Visual Studio 2017

A IDE da Microsoft apresenta uma ferramenta para correção de códigos que funciona em um ambiente independentemente do acesso à um terminal. O programa apresenta uma versão paga e uma versão aberta à comunidade.

D. Ferramenta de depuração nativa do Code::Blocks

Uma boa ferramenta por conta de seu uso rápido. Conhecida entre iniciantes (também por ser gratuita), essa IDE também tem a opção de depurar códigos. Seu uso pode ser vantajoso caso não se tenha acesso à uma máquina com Linux.

E. Ferramenta de depuração com o Dev-C++

O Dev-C++, após simples configuração, pode ser uma boa ferramenta de depuração de código. Com ele podemos analisar como o código se comporta. Podemos usar os chamados "breakpoints" para esta análise.

Referências

[1] The Cathedral and the Bazaar, Eric S. Raymond, 1999.
[2] Material de aula, Métodos de Programação (201600), UnB 2/2018
[3] An Abbreviated C++ Code Inspection Checklist, John T. Baldwin
[4] "Secure Programming for Linux and Unix HOWTO" by David A. Wheeler
[5] "Técnicas para levantamento de requisitos", devmedia.com.br
[6] https://privatebin.info/