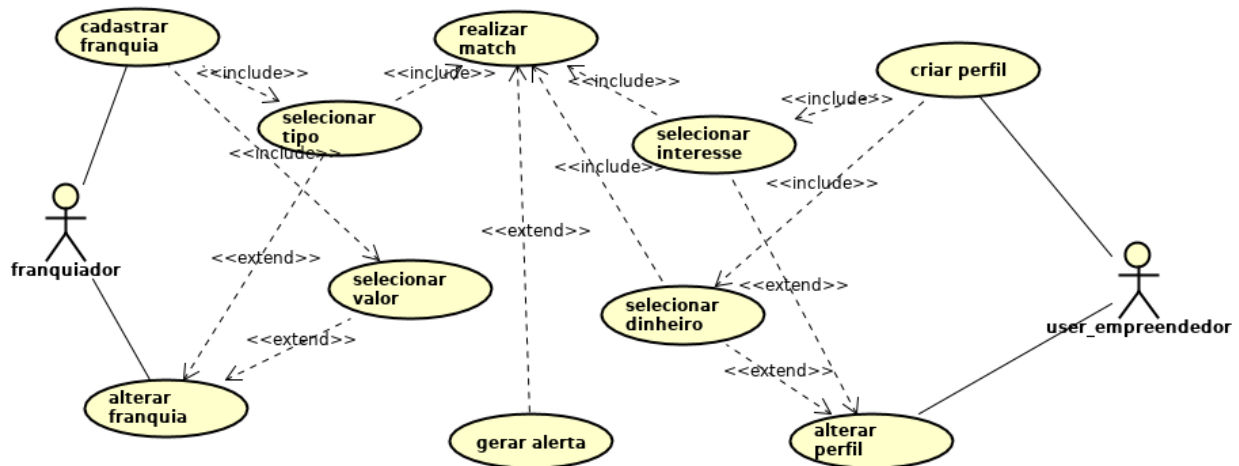


## Trabalho 2

Nome: Gabriel Porto Oliveira  
180058975

### 1-Especificação de requisitos

1.1- Utilizar diagramas de caso de uso para entender como o software será utilizado:



1.2- Organizar como cada tela e função serão executadas.

Tela\_1: Cadastro cliente  
Funções: cadastrarCliente();

1.3- Definir o mínimo projeto viável de acordo com a data.

Dessa forma caso aconteça algum imprevisto, um produto aceitável ainda pode ser entregue.

1.4- Fazer várias reuniões com o cliente antes de definir todos os requisitos.

- 1) Reunião 1 20/05
- 2) Reunião 2 03/06

1.5- Definir o que foi tratado em cada reunião.

- 1) Reunião 1 20/05 - Definido número de telas para o cliente
- 2) Reunião 2 03/06 - Definido a quantidade de usuários simultâneos

1.6- Estimar prazos de entrega.

Deadline: dezembro-2018

1.7- Estabelecer forma de desenvolvimento.

Entrega com prazo fixo e final.

Entrega com prazo aberto e reuniões durante o desenvolvimento.

1.8- Definir número de usuários que podem utilizar o software ao mesmo tempo.

Expectativa: 2000 conexões ao banco diárias.

1.9- Entender todas as ações que o usuário pode fazer e definir casos de exceção.

Cadastrar o cliente(3 campos são obrigatórios)

1.10- Estabeler tipos diferentes de usuários.

Cliente, Administrador, Gerente.

1.11- Estabeler o “poder” de cada tipo de usuário.

Cliente – Só gerencia a sua própria conta.

Administrador – Tem poder sobre todas as contas no sistema.

1.12- Estabeler quando o programa pode parar de ser executado.

Programa deve parar todo dia à noite.

1.13- Estabeler preferências de bibliotecas.

Biblioteca Libre.

Biblioteca proprietária.

1.14- Estabeler necessidade de SGBD ou arquivos txt

SGBD- banco de dados ou um arquivo txt

1.15- Definir o produto mais básico que pode ser entregue.

CRUD mais simples possível.

1.16- Definir custos adicionais para manter o software.

R\$ 500 por mês para bugfixing.

1.17- Estabeler a necessidade essencial do cliente.

Software para padaria, loja, hotel e etc.

1.18- Definir uso de GUI ou CLI e suas vantagens.

CLI- Usa menos recursos mas é menos intuitivo.

GUI- Mais bonito porém utiliza mais recursos.

1.19- Estabeler quanto o cliente esta disposto a pagar a mais caso seja necessário.

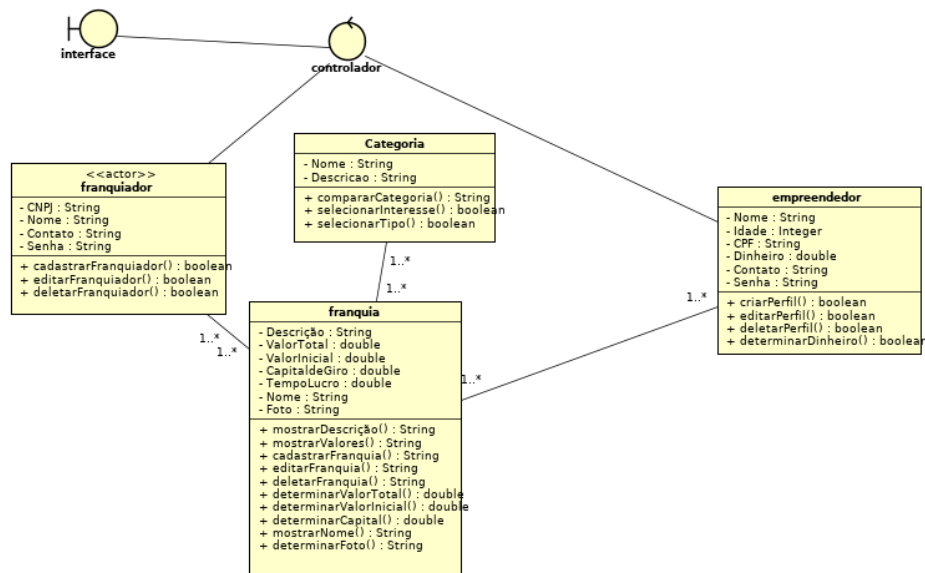
Se o cliente esta disposto a pagar mais R\$ 2000 caso o projeto atrase.

## 1.20- Estabelecer o local onde o software será executado.

Qual SO será utilizado, quanto de memória a máquina tem para otimizar o software.

## 2- Design do software.

### 2.1- Fazer diagrama de classes.



### 2.2- Definir como as telas ou CLI serão organizadas.

Menu CLI, Menu GUI e as suas opções.

### 2.3- Definir qual SGBD será utilizado.

PostgreSQL  
MySQL

### 2.4- Escolher as linguagens necessárias de acordo com a necessidade.

Back-end:

mais low-level mas básica: C

mais funcionalidades porém tem maior complexidade: C++

### 2.5- Definir a equipe necessária e quem vai desenvolver qual parte do software.

João back-end C

Paulo front-end TurboGUI

### 2.6- Dividir o software em funcionalidades e como serão desenvolvidas e implementadas.

Acesso ao Banco – Qual biblioteca será usado.

CRUD – Como o CRUD será feito.

2.7- Esquematizar todas as possíveis ações do usuário.

Tela1: Aqui o usuário pode fazer tal e tal coisa.

Tela2: Aqui ele pode fazer isso.

2.8- Formular soluções para todas as possíveis exceções.

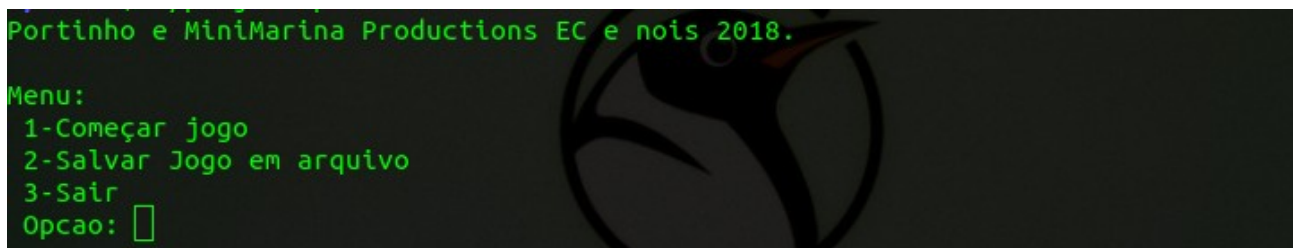
Tela1: Se o usuário tentar inserir um valor inválido, uma mensagem irá informa-lo que não é possível.

2.9- Planejar todas as funções, parâmetros e como esses irão funcionar.

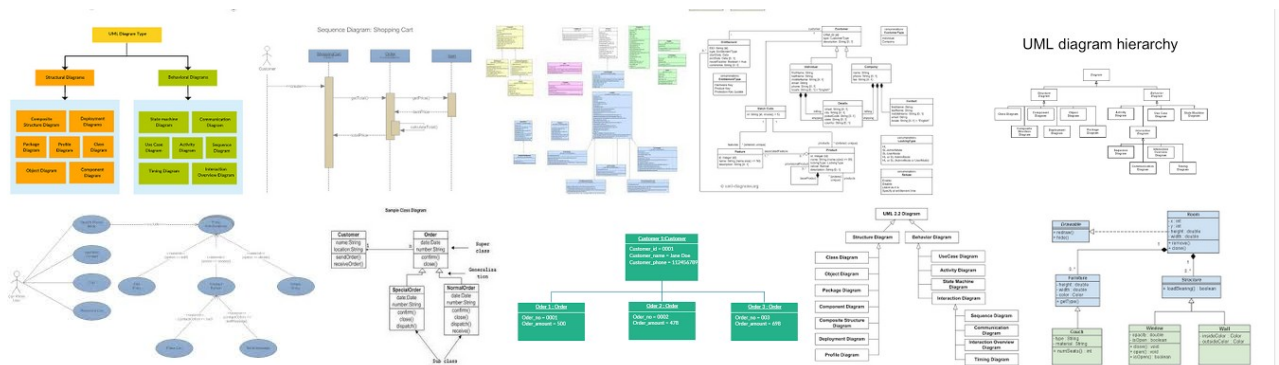
Função criarUser(int identificador, char \* nome)

Essa função vai criar o usuário ao receber o identificador e o nome.

2.10- Fazer esboços de como a GUI ou CLI podem ser antes de implementar.



2.11- Utilizar mais diagramas de modelagem para organizar o máximo possível.



2.12- Pensar em formas de otimizar a ordem de execução das funções para ter maior performance.

Não executar a função 1 dentro do laço se posso executar ela antes e ter o mesmo resultado.

2.13- Decidir a forma de programação.

Utilizando testes, fazendo todo o esqueleto das funções e depois a main ou fazendo a main sem ter as funções prontas.

2.14- Estabeler metas a cada uma ou duas semanas.

Semana 1: Terminar o layout.

Semana 2: Estabelecer toda a conexão com o banco.

2.15- Planejar o software de forma que seja fácil mante-lo mais tarde ou por outra equipe.

Funções bem planejadas e auto-explicativas. Vários comentários.

2.16- Criar padrões de qualidade a ser atingidos.

O layout tem de ser simples mas auto-explicativo.

2.17- Montar objetos de acordo com os usuários do programa.

Gerente: Identificador, CPF, Nome, Idade.

2.18- Utilizar somente estruturas que são necessárias.

Não criar estruturas que não sejam obrigatoriamente necessárias.

2.19- Levar em consideração as limitações físicas do hardware alvo.

Não criar software com grandes necessidades que irá ser executado em um mini-PC com pouca performance.

2.20- Projetar funções que podem ser reutilizadas.

Função 1 pode ser utilizada por várias outras funções.

### 3- Padrões de programação do software.

3.1- Não utilizar nomes genéricos para as variáveis.

```
Int x; //o que é x?  
Int y; //o que é y?
```

3.2- Utilizar nomes de variáveis que façam sentido.

```
Int contador; //será usado para um contador.  
Int statuscompra; //status de uma compra, 1 ou 0.
```

3.3- Comentar o que as funções fazem.

```
//Recebe um número e o insere no banco.  
Int insereNumero(int numero){};
```

3.4- Comentar partes complexas do código.

```
//Recebe o ponteiro da arvore e checa se é NULL se for libera memória.  
if(arvore!=NULL){  
    free(arvore);  
}
```

3.5- Criar funções com nomes auto-explicativos.

```
Void insereArquivo(FILE * arquivo, int numero);
```

3.6- Identar código corretamente.

```
Void main(){  
  
    int valor=0;  
  
    if(valor ==0){  
        printf("teste");  
    }  
  
}
```

3.7- Não criar código confuso.

Se depois de uma semana o programador tiver problemas para entender o próprio código, talvez o problema seja ele.

3.8- Criar padrões de codificação.

Nomes de variáveis sempre em minúsculo, nomes de funções começam com minúscula mas tem uma letra maiúscula.

3.9- Dividir grandes partes do código em funções independentes.

Transformar 100 linhas de código em 4 funções de 30 linhas que podem ser utilizadas várias vezes.

3.10- Comentar as variáveis utilizadas.

```
Int valorrecebido; //armazena o valor que será recebido do usuário.
```

3.11- Codificar de maneira que seja fácil manter.

Seja simples e faça sentido.

3.12- Criar código que outros entendam.

Em algum momento alguém vai precisar utilizar teu código. Ele tem de estar organizado o suficiente de forma que seja fácil saber o que está acontecendo.

### 3.13- Utilizar sistema de gerenciamento de versões.

Utilizar algum sistema como o git para versionar o código.

Se o software quebrou na versão 2.7 é possível voltar a 2.6 para saber qual o problema.

### 3.14- Separar o código em vários arquivos.

Separar as funções do main e utilizar um arquivo .h ou .hpp

### 3.15- Reutilizar funções o máximo possível.

Não criar código que não seja necessário, assim diminuindo o número de linhas.

### 3.16- Usufruir das funcionalidades da linguagem.

Não criar uma estrutura sendo ela já existente na linguagem.

### 3.17- Entender bem os conceitos da linguagem.

Usar ponteiros invés de variáveis globais.

### 3.18- Não utilizar variáveis globais.

Elas podem comprometer a segurança do programa.

### 3.19- Entender como a alocação de memória ocorre.

Muitos problemas de memory leak acontecem por falta de atenção ou conhecimento de métodos de alocação.

### 3.20- Utilizar constantes invés de #define.

Use `static const int var = 5;`  
invés de `#define var 5`

## 4- Padrões para testagem de Software.

### 4.1- Utilizar alguma biblioteca de testes de código.

Algumas bibliotecas são o Catch e o Gtest.

### 4.2- Testar todas as funções existentes no código.

```
unsigned int Factorial( unsigned int number ) {  
    return number <= 1 ? number : Factorial(number-1)*number;  
}  
  
TEST_CASE( "Factorials are computed", "[factorial]" ) {  
    REQUIRE( Factorial(1) == 1 );  
    REQUIRE( Factorial(2) == 2 );  
    REQUIRE( Factorial(3) == 6 );  
    REQUIRE( Factorial(10) == 3628800 );  
}
```

4.3- Checar se os testes fazem sentido.

Checar por um parâmetro que não é possível, não tem necessidade.

4.4- Utilizar todas as ferramentas disponibilizadas pelos frameworks de teste.

4.5- Testar se todas as funções implementadas são utilizadas.

Utilizar ferramentas como o gcov para entender a utilização das funções.

4.6- Checar se parâmetros fora do esperado quebram o software ou se ele consegue seguir funcionando normalmente.

Se alguma variável for NULL o programa consegue seguir sua execução?

4.7- Criar testes que levem em consideração as possíveis opções do usuário e exceções.

Checar se o programa segue funcionando caso o usuário coloque a respostas esperada ou se coloca algo inesperado.

4.8- Utilizar vários frameworks de teste simultaneamente.

Se um framework não reconhece uma falha, talvez outro consiga.

4.9- Testar utilizando input humano, como um colega de trabalho.

Após o software passar por todos os teste automatizados, utilizar teste humano.

4.10- Testar para erros de sintaxe e lógica.

Ter certeza que a implementação utilizada é possível.

4.11- Utilizar diversos compiladores.

O compilador acusa alguns possíveis erros.

4.12- Testar para usabilidade.

O quão responsivo e intuitivo o software é.

4.13- Testar as funções assim que elas são feitas.

É mais fácil achar um erro em uma função recém criada do que após o software estar totalmente pronto.

4.14- Utilizar a ajuda de outros programadores.



Maiores são as chances de outro programador achar uma falha no código do que o programador trabalhando no código.

4.15- Criar unidades de testes independentes do código principal.



4.16- Corrigir erros apontados pelos testes rapidamente e com soluções inteligentes.

Não ignorar erros ou corrigi-los utilizando métodos “alternativos”.

4.17- Entender como os diferentes frameworks funcionam antes de utilizá-los.

Isso ajuda a otimizar suas utilizações e escolher o mais apropriado.

4.18- Usar linux.

Grande maioria dos frameworks de teste são criados em ambientes linux.

4.19- Testar utilizando vários tipos de variáveis.

Um tipo específico de variável pode diminuir linhas de código e facilitar a testagem.

4.20- Usar testes para entender o código.

É mais fácil visualizar quais variáveis estão entrando e saindo utilizando testes.

## 5- Padrões para *debugging* de Software.

5.1- Usar debugger como o gdb.

```
gabriel@linux-porto:~$ gdb --help
This is the GNU debugger.  Usage:

gdb [options] [executable-file [core-file or process-id]]
gdb [options] --args executable-file [inferior-arguments ...]

Selection of debuggee and its files:
  --args                Arguments after executable-file are passed to inferior
  --core=COREFILE       Analyze the core dump COREFILE.
  --exec=EXECFILE       Use EXECFILE as the executable.
  --pid=PID             Attach to running process PID.
  --directory=DIR       Search for source files in DIR.
  --se=FILE             Use FILE as symbol file and executable file.
  --symbols=SYMFIL      Read symbols from SYMFIL.
  --readnow             Fully read symbol files on first access.
  --readnever          Do not read symbol files.
  --write               Set writing into executable and core files.

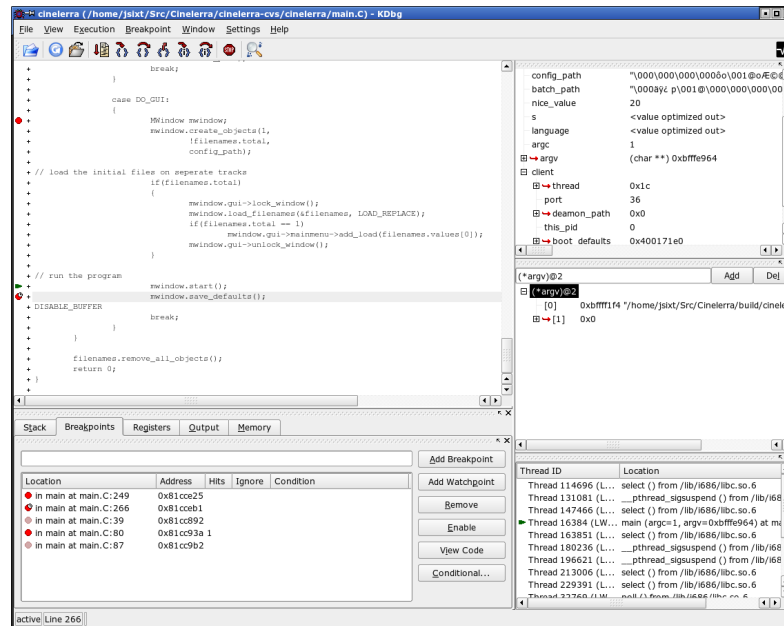
Initial commands and command files:
  --command=FILE, -x    Execute GDB commands from FILE.
  --init-command=FILE, -ix
                        Like -x but execute commands before loading inferior.
  --eval-command=COMMAND, -ex
                        Execute a single GDB command.
                        May be used multiple times and in conjunction
                        with --command.
  --init-eval-command=COMMAND, -iex
                        Like -ex but before loading inferior.
  --nh                 Do not read ~/.gdbinit.
  --nx                 Do not read any .gdbinit files in any directory.

Output and user interface control:
  --fullname            Output information used by emacs-GDB interface.
  --interpreter=INTERP  Select a specific interpreter / user interface
```

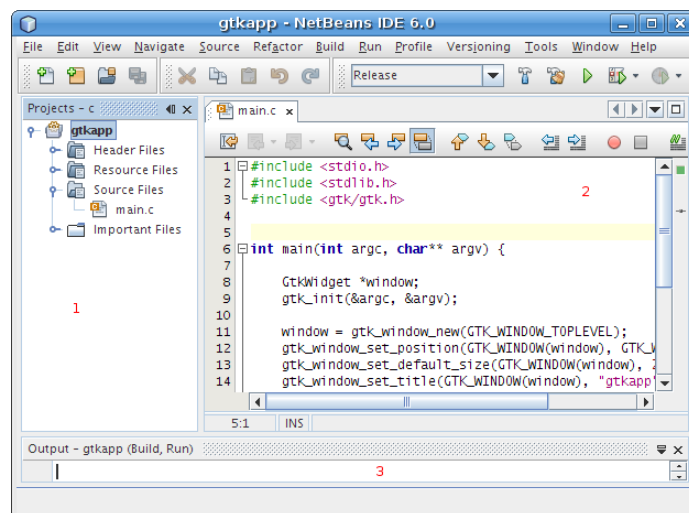
## 5.2- Entender como o debugger funciona para otimizar sua utilização.

Usar todas as ferramentas que o GDB, por exemplo, tem disponível.

## 5.3- Utilizar GUI para facilitar o processo e ficar mais fácil de visualizar erros.



## 5.4- Utilizar alguma IDE, isso pode diminuir as chances de algo dar errado e integrar o desenvolvimento. Facilitando o processo de debugging.



## 5.5- Depurar funções independentes e separadamente.

Assim fica mais fácil achar o erro e ele pode ser corrigido na hora.

## 5.6- Modificar variáveis durante a depuração.

Quando fazemos isso temos controle total sobre os valores das variáveis, para poder descartar a função e examinar como o dado original foi criado.

## 5.7- Visualizar a mudança dos valores em real-time.

Utilizando GUIs é possível ver as variáveis mudar em tempo real, dessa forma fica mais fácil saber o momento exato onde acontece o erro.

#### 5.8- Não depurar o programa inteiro de uma vez.

Depurar o software inteiro de uma vez quase nunca mostra o erro e ainda gera gasto de tempo.

#### 5.9- Não deixe de utilizar GUIs se elas te ajudam.

Não porque utilizar o GDB em um terminal se é possível e disponível uma GUI apropriada.

#### 5.10- Preste atenção em ponteiros, uma pequena diferença pode quebrar o software.

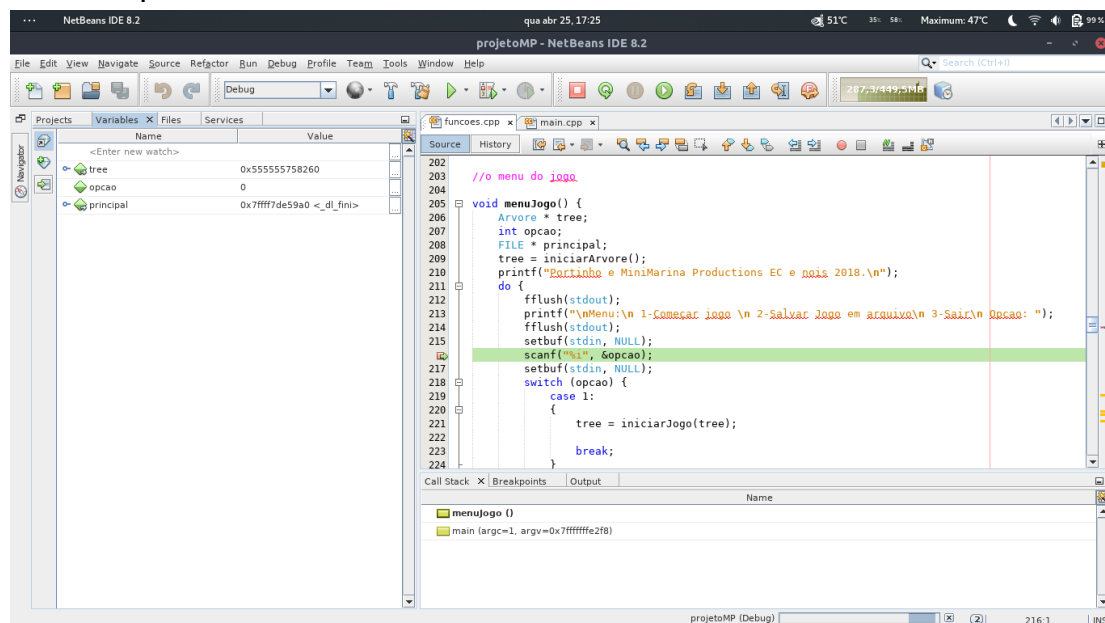
Debuggers permitem que observemos a mudança nos ponteiros e seus valores.

#### 5.11- Comentar códigos já depurados e testados.

Dependendo do tamanho do seu software a depuração pode levar dias ou semanas. Nesses cenários comentar o código já depurado com a data da depuração, se deu certo e os parâmetros que foram testados pode ajudar muito quando tu voltar para continuar mais tarde.

#### 5.12- Checar o comportamento linha por linha.

Quando fazemos isso podemos saber exatamente em qual linha o erro aconteceu e o que o causou.



#### 5.13- Não colocar breakpoints em nada.

A maioria das excessões acontecem na leitura ou manipulação dos dados, não na apresentação deles.

#### 5.14- Entender as mensagens de erro.

Mesmo utilizando um debugger e vendo as variáveis, erros vão acontecer. É importante entendê-los para evitar no futuro e diagnosticar o problema mais cedo.

#### 5.15- Comentar linhas de código.

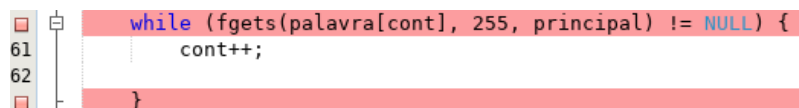
Alguns erros são específicos de uma linha, tente comentar elas e ver se o erro persiste.

5.16- Diminuir os breakpoints com o passar do tempo, assim é possível diminuir o número de possíveis linhas de código com problema.

#### 5.17- Testar os arquivos .c ou .cpp um de cada vez.

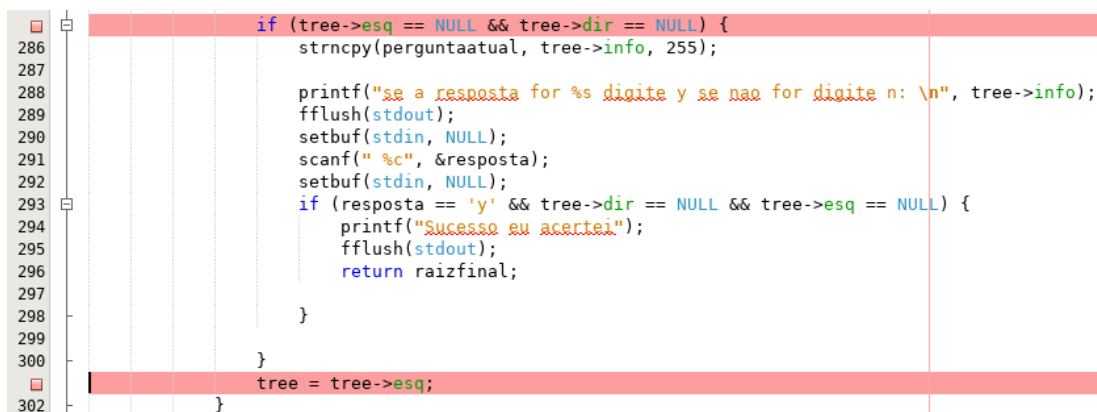
Dessa maneira fica mais fácil saber em qual arquivo está o erro.

5.18- Coloque breakpoints no início e final de laços para saber se o problema está antes, depois ou dentro deles.



```
61 while (fgets(palavra[cont], 255, principal) != NULL) {
62     cont++;
}
```

#### 5.19- Utilizar breakpoints em ações com ponteiros.



```
286 if (tree->esq == NULL && tree->dir == NULL) {
287     strncpy(perguntaatual, tree->info, 255);
288     printf("se a resposta for %s digite y se nao for digite n: \n", tree->info);
289     fflush(stdout);
290     setbuf(stdin, NULL);
291     scanf(" %c", &resposta);
292     setbuf(stdin, NULL);
293     if (resposta == 'y' && tree->dir == NULL && tree->esq == NULL) {
294         printf("Sucesso eu acertei");
295         fflush(stdout);
296         return raizfinal;
297     }
298 }
299 }
300 tree = tree->esq;
302 }
```

#### 5.20- Colocar breakpoints em chamadas de funções.

Dessa forma podemos saber se as respostas das funções estão de acordo com o que é enviado.