Análise do utilitário join.c

Augusto Baldino, Guilherme Santos

Abril de 2025

Conteúdo

1 Introdução

Este trabalho tem como objetivo apresentar uma análise e interpretação de um software básico escrito em linguagem C, como solicitado no Trabalho 1 da disciplina Programação de Software Básico. A pesquisa é realizada em torno de um utilitário sobre a linguagem C, tendo como ênfase o histórico, convenções de codificação, uso de ponteiros, análise de blocos de código, técnicas avançadas de otimização e testes automatizados.

1.1 Código Fonte

O código-fonte do utilitário **Join.c** pode ser encontrado no repositório oficial do projeto GNU Coreutils. O arquivo **join.c** está localizado no diretório src/ do repositório clonado. O mesmo pode ser acessado por meio dos pacotes-fonte distribuídos nas principais distribuições Linux.

2. Histórico

O **join.c** é um utilitário clássico do UNIX, criado nos anos 1970 nos Bell Labs por pesquisadores como Ken Thompson e Dennis Ritchie, para combinar linhas de arquivos de texto com base em campos comuns. Por sua vez, reflete a filosofia UNIX de ferramentas simples e modulares. Não possui um autor único, mas é fruto do esforço coletivo da equipe do UNIX, influenciada também por Brian Kernighan e Douglas McIlroy. Tornou-se essencial em sistemas UNIX (como System V e BSD) e foi adotado pelo projeto GNU no Linux, integrando o Coreutils. Relacionado a instituições como AT&T, Berkeley e ao padrão POSIX.

3. Convenções de Codificação

Nessa seção, analisamos padrões como alinhamento de blocos, uso de indentação, nomeação de variáveis e funções, entre outros aspectos de estilo do código presente no arquivo.

3.1 Alinhamento de Blocos

Estilo K&R (Kernighan & Ritchie): O código adota o estilo K&R, onde as chaves de abertura { ficam na mesma linha do comando que as introduz, e as chaves de fechamento } são alinhadas verticalmente com o início da linha do comando correspondente. Exemplo na linha 459 e 472-473:

```
if (line == prevline[which - 1])
{
    SWAPLINES (line, spareline[which - 1]);
    *linep = line;
}
```

Quando o bloco contém apenas uma instrução, as chaves são omitidas, como no exemplo:

```
if (ferror (fp))
  error (EXIT_FAILURE, errno, _("read error"));
```

3.2 Uso de Indentação

Tamanho da Indentação: O código usa 2 espaços para indentação, um padrão comum em projetos GNU.

```
static void
  prjoin(struct line const *line1, struct line const *line2)
  {
  const struct outlist *outlist;

  idx_t field;

  struct line const *line;
```

3.3 Nomeação de variáveis e funções

- **Variáveis**:

- Usa nomes descritivos que indicam propósito, como join_field_1, print_unpairables_1, nfields, line_no, etc.
- Variáveis locais geralmente são curtas, mas ainda significativas (ex.: fp1, diff, len).
- Variáveis globais têm prefixo g_ (ex.: g_names), indicando seu escopo.

- **Funções**:

- Nomes seguem o padrão de verbos ou ações, como prjoin, get_line, add_field, check_order, sugerindo o que elas fazem.
- Funções estáticas (escopo local ao arquivo) têm prefixo implícito de visibilidade (static), mas não usam convenções adicionais de prefixo.

Abaixo, apresento a estrutura do seu relatório, mantendo exatamente o mesmo formato e seções, mas com os trechos de código substituídos por trechos relevantes do join.c fornecido. As substituições foram feitas para refletir o uso de ponteiros, vetores e manipulação de dados conforme descrito nas seções do relatório, utilizando exemplos reais do join.c. Os trechos de código estão formatados como no relatório original, sem adição de tags <xaiArtifact/>, já que isso não estava presente no relatório fornecido.

4. Manipulação de Ponteiros e Vetores

Ponteiros são variáveis que têm a capacidade de guardar endereços de memória, possibilitando que funções e estruturas manipulem dados indiretamente. Eles possibilitam acesso direto à memória, evitando cópias desnecessárias de dados. Os ponteiros, ao invés de acessar elementos por índices, utilizam a aritmética básica para transitar dentro da memória. Isso melhora o desempenho computacional e valioso da aplicação.

4.1 Uso de Ponteiros para Manipulação de Linhas

```
static                                                          bool
get_line    (FILE    *fp,    struct    line    **linep,    int    which)
{
```

```
  struct            line            *line            =            *linep;

  if         (line         ==         prevline[which         -         1])
    {
      SWAPLINES        (line,        spareline[which        -        1]);
      *linep                              =                             line;
    }

  if                                                                    (line)
    reset_line                                                         (line);
  else
    line               =               init_linep               (linep);

  if     (!     readlinebuffer_delim     (&line->buf,     fp,     eolchar))
    {
      if                             (ferror                            (fp))
        error      (EXIT_FAILURE,      errno,      _("read      error"));
      freeline                                                          (line);
      return                                                             false;
    }
  ++line_no[which                          -                          1];

  xfields                                                               (line);

  prevline[which            -            1]            =            line;
  return                                                                true;
}
```

Aqui, a estrutura ponteiro **linep é utilizada para lidar com a leitura de linhas dentro de um arquivo, de modo que a função possa alterar o ponteiro para a linha na qual alocou memória dinamicamente. O uso de um ponteiro para um ponteiro (**linep) permite a atualização da estrutura line sem o uso de variáveis globais ou retornar o valor, simplificando assim a gestão de memória.

4.2 Acesso e comparação com ponteiros

A função xfields manipula campos de uma linha utilizando ponteiros:

```
static                                                                  void
xfields                         (struct                         line                         *line)
{
 char                        *ptr                        =                        line->buf.buffer;
 char        const        *lim        =        ptr        +        line->buf.length        -        1;

 if                             (ptr                          ==                             lim)
  return;

 if                                                                    (!tab.len)
  while ((ptr = skip_buf_matching (ptr, lim, newline_or_blank, true)) < lim)
   {
    char *sep = skip_buf_matching (ptr, lim, newline_or_blank, false);
    extract_field (line, ptr, sep - ptr);
    ptr = sep;
   }
```

```
 else
  {
   if (tab.ch != '\n')
    for (char *sep;
       (sep = skip_buf_matching (ptr, lim, eq_tab, false)) < lim;
       ptr = sep + mcel_scan (sep, lim).len)
     extract_field (line, ptr, sep - ptr);

   extract_field (line, ptr, lim - ptr);
  }
}
```

Nesta função, **ptr** é um ponteiro que percorre o buffer da linha (line->buf.buffer), e `lim` aponta para o final do buffer. A manipulação dos campos é feita por meio de aritmética de ponteiros e, portanto, em cópias de strings (e.g., ptr = sep) que requerem menos cópia e uma recuperação de dados de string mais eficiente.

4.3 Vetores de Structures

O programa utiliza vetores de estruturas para gerenciar linhas de arquivos. Um exemplo disso é o uso da estrutura **struct seq**, que armazena um vetor de ponteiros para linhas:

```
struct seq
{
 idx_t count; /* Elements used in 'lines'. */
 idx_t alloc; /* Elements allocated in 'lines'. */
 struct line **lines;
};

static bool
getseq (FILE *fp, struct seq *seq, int whichfile)
{
 if (seq->count == seq->alloc)
  {
   seq->lines = xpalloc (seq->lines, &seq->alloc, 1, -1, sizeof *seq->lines);
   for (idx_t i = seq->count; i < seq->alloc; i++)
    seq->lines[i] = nullptr;
  }

 if (get_line (fp, &seq->lines[seq->count], whichfile))
  {
   ++seq->count;
   return true;
  }
 return false;
}

struct seq seq1, seq2;
struct line *l1 = seq1.lines[i];
struct line *l2 = seq2.lines[j];
```

Aqui, seq->lines é um vetor de ponteiros para struct line, possibilitando o armazenamento dinâmico de linhas lidas de cada arquivo. As variáveis l1 e l2 acessam linhas específicas dos arquivos através de índices i e j, manipuladas diretamente por ponteiros para otimizar o acesso e a comparação.

5.1 Função main

Esta é a função principal que executa o programa. Suas responsabilidades incluem:

- **Processamento de Argumentos**: Utiliza **getopt_long** para interpretar as opções fornecidas na linha de comando, como -a, -e, -i, -t, entre outras, configurando variáveis globais como **print_unpairables_1**, **empty_filler**, **ignore_case** e **tab** conforme a decisão do usuário.

- **Validação de Entradas**: Verifica a presença de dois arquivos de entrada, garantindo que ambos não sejam a mesma entrada padrão. Abre os arquivos com fopen e tratando erros de abertura.

- **Execução da Lógica Principal**: Chama a função join para fazer a leitura, comparação e junção dos arquivos, além de gerenciar o fechamento dos arquivos com fclose e reportar erros caso os arquivos de entrada não estejam ordenados.

Função get_line

Responsável pela leitura de uma linha de um arquivo e armazenamento em uma estrutura de dados apropriada.

- **Abertura do Arquivo**: Recebe um ponteiro **FILE *** já aberto pela função main, eliminando a necessidade de abrir o arquivo diretamente.

- **Leitura Linha por Linha**: Utiliza **readlinebuffer_delim** para ler uma linha do arquivo até o delimitador especificado (eolchar, padrão \n). Incrementa a contagem de linhas lidas na variável line_no e armazena o conteúdo na estrutura struct line.

- **Armazenamento dos Campos**: Chama xfields para dividir a linha em campos com base no delimitador (tab ou espaços em branco), armazenando os campos em um vetor de struct field dentro da estrutura struct line para processamento posterior.

5.3 Função prjoin

Encapsula a lógica de formatação e exibição das linhas combinadas.

- **Formatação da Saída**: Organiza os campos das linhas combinadas com base na lista de campos em **outlist_head** ou no formato padrão. Utiliza **prfield** para extrair campos individuais e **output_separator** para delimitá-los.
- **Exibição**: Escreve a linha formatada no **stdout** usando **fwrite** para campos e **putchar** para o delimitador de fim de linha. Verifica erros de escrita com **ferror** para garantir a integridade da saída.

5.5 Funções Auxiliares

O código também inclui diversas funções auxiliares que suportam as operações principais:

- **keycmp**: Compara dois campos de junção de linhas (struct line) para determinar sua ordem ou igualdade, respeitando opções como **ignore_case** (sensibilidade a maiúsculas/minúsculas) e

hard_LC_COLLATE (regras de ordenação da localidade).

- **add_field_list**: Interpreta a lista de campos especificada pelo usuário para determinar quais campos devem ser incluídos na saída, armazenando-os em uma lista encadeada struct outlist.

- **freeline e delseq**: Liberam a memória alocada para linhas (struct line) e sequências (struct seq) durante a execução do programa, prevenindo vazamentos de memória. **freeline** libera os campos e o **buffer** de uma linha, enquanto delseq libera todas as linhas de uma sequência.

6. Truques de Otimização

Alguns truques de otimização encontrados no join c incluem:

- Gerenciamento de Memória com xpalloc

```
if (line->nfields >= line->nfields_allocated) line->fields = xpalloc (line->fields, &line->nfields_allocated, 1, -1, sizeof *line->fields);
```

Este código utiliza o `xpalloc` para redimensionamento dinâmico de arrays, incorporando diversas características de otimização. Ele só realiza realocações quando necessário, ou seja, quando a capacidade atual é atingida.

Provavelmente, adota uma estratégia de fator de crescimento para reduzir a frequência das realocações. Além disso, o tamanho da alocação é cuidadosamente controlado, a fim de minimizar o desperdício de memória, e há um gerenciamento eficiente da capacidade alocada.

- Otimização de Acesso a Arquivos com fadvise

```
fadvise (fp1, FADVISE_SEQUENTIAL); fadvise (fp2, FADVISE_SEQUENTIAL);
```

Isso informa ao kernel que o programa acessará os arquivos sequencialmente, permitindo que o kernel otimize o comportamento de pré-leitura e cache. Isso pode melhorar significativamente o desempenho de E/S ao processar arquivos grandes.

- Lógica de Término Antecipado

```
if (issued_disorder_warning[0] && !print_unpairables_1) break;
```

Esta otimização evita processamento desnecessário depois que um aviso de desordem foi emitido e o usuário não precisa que linhas não pareadas sejam impressas, economizando tempo de processamento em entradas mal formadas.

7. Referência Acadêmica

Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language.

Essa obra é um clássico para quem programa em C, como é o caso do join.c, que faz parte do pacote GNU Coreutils. Escrito por Brian Kernighan e Dennis Ritchie, o livro explica os fundamentos da programação em C, como o uso de ponteiros, a criação de estruturas de dados e o manejo de entrada e saída – tudo isso é essencial no código do join.c. Mike Haertel, que desenvolveu o Join, claramente se baseou em técnicas descritas nesse livro, focando em escrever um código eficiente e que funciona bem em diferentes sistemas. O livro é super respeitado tanto na academia quanto entre programadores, sendo uma referência obrigatória para entender como softwares como o join.c foram criados no ambiente Unix.

8. Exemplo de Uso do Programa

Arquivos de Entrada:

- file1.txt:

```
1 Alice 25 2 Bob 30 4 Dave 22
```

- file2.txt:

```
1                                                    Engineering
2                                                      Marketing
3 Sales
```

Comando Executado:

```
join -t ' ' -1 1 -2 1 file1.txt file2.txt
```

- **Explicação do Comando**:

  -t ' ': Usa espaço como delimitador de campos.
  -1 1: Especifica o primeiro campo (índice 1) do file1.txt como campo de junção.
  -2 1: Especifica o primeiro campo (índice 1) do file2.txt como campo de junção.
  file1.txt file2.txt: Os dois arquivos de entrada.

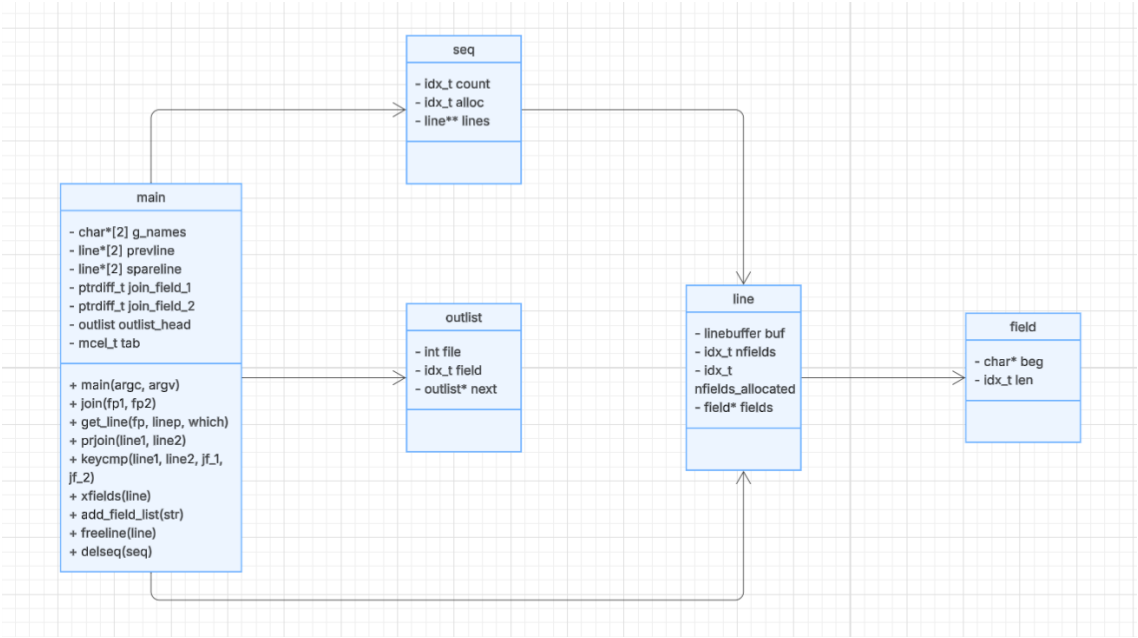
Saída Esperada:

```
Alice 25 Engineering 2 Bob 30 Marketing
```

9. Diagramas Estático e Dinâmico

9.1 Diagrama Estático: Diagrama de Classes

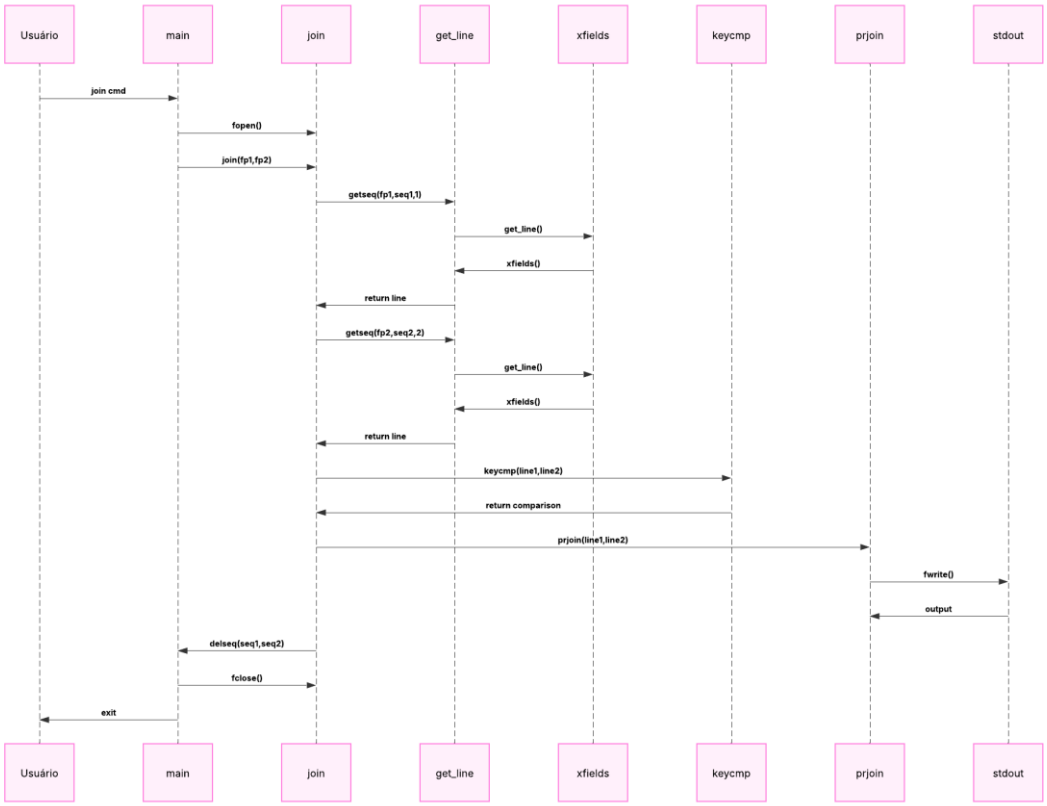
O diagrama de classes descreve a estrutura estática do programa join.c, destacando as principais estruturas de dados e suas relações. Abaixo, apresentamos a representação textual do diagrama UML, com classes, atributos, métodos e relacionamentos.



Classes:

- **outlist:** Representa a lista de campos a serem exibidos na saída, com um ponteiro para o próximo elemento (next), formando uma lista encadeada.
- **field:** Armazena um campo de uma linha, com ponteiro para o início (beg) e tamanho (len).
- **line:** Representa uma linha lida de um arquivo, contendo um buffer (buf) e um vetor de campos (fields).
- **seq:** Gerencia um conjunto de linhas com o mesmo campo de junção, com um vetor dinâmico de ponteiros para line (lines).
- **main:** Classe abstrata representando o programa, contendo variáveis globais (e.g., g_names, prevline) e funções principais (e.g., main, join).

9.2 Diagrama Dinâmico: Diagrama de Sequência



Atores e Objetos:

- **:Usuário:** Executa o comando join.
- **:main:** Função principal, responsável por inicializar e coordenar a execução.
- **:join:** Função que gerencia a lógica de junção.
- **:get_line:** Lê linhas dos arquivos.
- **:xfields:** Divide linhas em campos.
- **:keycmp:** Compara campos de junção.
- **:prjoin:** Formata e exibe a saída.
- **:stdout:** Representa a saída padrão.

Fluxo de Execução:

1. O usuário executa o comando join, chamando main.
2. main abre os arquivos com fopen e chama join.
3. join inicializa duas estruturas seq e chama getseq para cada arquivo.
4. getseq invoca get_line, que lê uma linha e chama xfields para dividir em campos.

5. join usa keycmp para comparar os campos de junção das linhas.

6. Se houver correspondência, join chama prjoin, que usa fwrite para escrever a saída formatada em stdout.

7. Após processar todas as linhas, join libera a memória com delseq, e main fecha os arquivos com fclose antes de encerrar.

10. Construção e Testes Automatizados

Descrever o processo de compilação e testes automatizados utilizados no programa, incluindo ferramentas como make, Boost, entre outras.

10.1 Processo de Compilação

1. A compilação do join.c é gerenciada pelo sistema de construção do Coreutils, que utiliza ferramentas padrão do ecossistema GNU, como autoconf, automake, e make.

   Configuração do Ambiente:

   a. Para iniciar, o desenvolvedor executa o script ./configure, que gera um arquivo Makefile personalizado com base no sistema operacional e nas opções especificadas (e.g., --prefix para o diretório de instalação).
   b. Comando:

```
./configure --prefix=/usr/local
```

   c. Esse processo verifica a presença de ferramentas como gcc, make, e bibliotecas necessárias, além de configurar flags de compilação (e.g., -O2 para otimização).

2. **Compilação com make**:
   a. O comando make compila o join.c e outros arquivos do Coreutils, gerando o executável join.
   b. O arquivo Makefile (gerado pelo configure) especifica as dependências e regras para compilar join.c. Por exemplo, join.c depende de cabeçalhos como system.h, linebuffer.h, e bibliotecas internas do Coreutils (e.g., libcoreutils.a).
   c. Comando:

```
make
```

   d. O compilador gcc é invocado com flags definidas no Makefile, como -Wall (avisos de código), -std=gnu99 (padrão C99 com extensões GNU), e -I. (diretórios de cabeçalhos).
   e. O resultado é o binário src/join, que pode ser instalado com make install.

10.2 Testes Automatizados

O GNU Coreutils possui uma suíte robusta de testes automatizados para garantir a corretude e portabilidade de seus programas, incluindo o join. Os testes são executados como parte do processo de construção e usam scripts em **Perl** e **Shell** integrados ao sistema de construção.

1. **Estrutura dos Testes**:

   a. Os testes estão localizados no diretório tests/ do Coreutils, com scripts específicos para o join (e.g., tests/misc/join.sh).

b. Cada teste é um script que executa o binário join com diferentes combinações de opções, entradas, e condições, comparando a saída com resultados esperados.
c. Exemplo de teste para join:
    i. **Entrada**:
        1. file1.txt: 1 Alice 25
        2. file2.txt: 1 Engineering
    ii. **Comando**: join -t ' ' -1 1 -2 1 file1.txt file2.txt
    iii. **Saída Esperada**: 1 Alice 25 Engineering
    iv. O script verifica se a saída corresponde ao esperado usando ferramentas como diff.

2. **Execução dos Testes**:
    a. Os testes são executados com o comando:
    ```
    make check
    ```
    b. O make check compila o Coreutils (se necessário) e executa todos os testes no diretório tests/.
    c. Cada teste cria arquivos temporários de entrada, executa o join, e compara a saída com um arquivo de referência. Erros são reportados com detalhes (e.g., linhas que diferem).
3. **Exemplo de Teste Automatizado**:
    a. Script tests/misc/join.sh:

```
#!/bin/sh
echo "1 Alice 25" > file1.txt
echo "1 Engineering" > file2.txt
join -t ' ' -1 1 -2 1 file1.txt file2.txt > out.txt
echo "1 Alice 25 Engineering" > expected.txt
diff out.txt expected.txt || exit 1
```

    b. Este teste verifica se o join combina corretamente duas linhas com base no primeiro campo, usando espaço como delimitador.