UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

LEONARDO VEIGA E NICOLAS DA SILVA

COMPILANDO PROGRAMAS EM C

Rio de Janeiro
Abril de 2023

LISTA DE ILUSTRAÇÕES

Figura 1 – Diagrama de funcionamento do compilador 7
Figura 2 – Etapas da compilação 9

LISTA DE TABELAS

Tabela 1 – Tipos de dados básicos do C. 6

LISTA DE ABREVIATURAS E SIGLAS

AMD	Advanced Micro Devices
ANSI	American National Standards Institute
ARM	Advanced RISC Machine
gcc	GNU Compiler Collection / GNU C Compiler
ISO	International Organization for Standardization
MAC OS	Macintosh Operational System
MinGW	Minimalist GNU for Windows
MSYS	Minimal System
RISC	Reduced Inctruction Set Computer

SUMÁRIO

1 CAPÍTULO 1 5
1.1 CARACTERÍSTICAS DA LINGUAGEM C **5**
1.2 O QUE É UM COMPILADOR **7**
1.3 GNU COMPILER COLLECTION **8**

2 CAPÍTULO 2 11
2.1 COMPILANDO UM PROGRAMA EM C **11**
2.2 TRABALHANDO COM MÚLTIPLOS ARQUIVOS **12**
2.3 TRABALHANDO COM BIBLIOTECAS EXTERNAS **13**
2.4 USANDO MAKEFILE **13**

3 CAPÍTULO 3 16
3.1 BANDEIRAS DE AVISO DO COMPILADOR **16**
3.2 USANDO O PRÉ-PROCESSADOR **17**
3.3 COMPILANDO PARA DEBUGAR (NOTAS POR ENQUANTO) . . **18**

4 CAPÍTULO 4 19
4.1 COMPILANDO PARA DEPURAÇÃO **19**
4.2 DEPURANDO PROGRAMAS **20**

5 COMPILANDO PARA A DEPURAÇÃO 21
5.1 EXAMINANDO ARQUIVOS CENTRAIS **21**
5.2 EXIBINDO UMA PILHA DE CHAMADAS **21**

6 COMPILANDO COM OTIMIZAÇÃO 22
6.1 OTIMIZAÇÃO EM NÍVEL DE CÓDIGO-FONTE **22**
6.2 ELIMINAÇÃO DE SUBEXPRESSÕES COMUNS **22**
6.3 INCLUSÃO DE FUNÇÃO **22**
6.4 TRADE-OFFS DE VELOCIDADE E ESPAÇO **23**
6.5 DESENROLAMENTO DE LOOPS **23**
6.6 AGENDAMENTO **23**
6.7 NÍVEIS DE OTIMIZAÇÃO NO GCC: **23**
6.8 6.6 OTIMIZAÇÃO E DEPURAÇÃO **24**
6.9 6.7 OTIMIZAÇÃO E AVISOS DO COMPILADOR **24**

GLOSSÁRIO 26

1 CAPÍTULO 1

Esse capítulo visa estabelecer conceitos importantes que envolvem a linguagem C, compiladores e programação no geral.

1.1 CARACTERÍSTICAS DA LINGUAGEM C

C é uma Linguagem de Programação compilada, isto é, ao escrevermos o Código Fonte na própria linguagem, no caso em C, um programa chamado compilador, que tem como entrada um arquivo com Código Fonte da linguagem e que gera como saída um Arquivo Objeto, com Código Objeto, que é ligado à outros arquivos objeto, para gerar um Arquivo Executável. Sendo assim, o compilador da linguagem C é um programa que recebe um arquivo com Código Fonte escrito em C, e, gera a partir disso, um arquivo que pode ser executado no computador alvo. Na próxima seção, serão dados mais detalhes sobre o processo de compilação de um arquivo em C. Abaixo seguem algumas características importantes da linguagem C:

- **Estruturada** → A programação estruturada (sucedida pela programação orientada a objeto) é um paradigma formado por três componentes:
  - Sequência: Uma tarefa é executada logo após a outra;
  - Decisão: A tarefa é executada logo após um teste lógico;
  - Iteração: A partir de um teste lógico, um trecho de código pode ser repetido finitas vezes.
- **Imperativa** → Em contraste com a programação imperativa, o paradigma imperativo descreve ações/instruções que o programa deverá tomar/executar. Ou seja, linguagens imperativas são programadas com uma sequência de comandos ordenada pela programador;
- **Procedural** → O paradigma procedural permite a linguagem construir procedimentos que podem ser compartimentados e reutilizados de forma a tornar partes do códigos mais independentes entre si;
- **Padronizada** → A padronização é uma característica que dita a regularidade da linguagem, de modo que um mesmo código gere sempre o mesmo resultado, seja ele compilado e executado ou interpretado;
- **Fortemente e estaticamente Tipada** → Em C, o tipo das variáveis/funções precisam ser bem definidos e são mantidos durante toda a execução do programa. Porém, com ponteiros do tipo *void*, é possível driblar essa questão, sendo possível,

mas não aconselhável, mudar implicitamente o tipo da variável que ponteiro está apontando.

Abaixo segue uma tabela com tipos de dados básicos da linguagem, onde a palavra-chave é usada para definir as variáveis e o formato indica a forma de capturar (por meio de uma função como *scanf*) ou de imprimir (por exemplo, com a função *printf*):

PALAVRA-CHAVE	TIPO	BYTES	INTERVALO	FORMATO
char / signed char	Caracter	1	−128 a 127	%c
unsigned char	Caracter sem sinal	1	0 a 255	%c
short / short int / signed short / signed short int	Inteiro curto com sinal	2	−32768 a 32767	%hi ou %hd
unsigned short / unsigned short int	Inteiro curto sem sinal	2	0 a 65535	%hu
signed / int / signed int	Inteiro com sinal	2	−32768 a 32767	%i ou %d
unsigned / unsigned int	Inteiro sem sinal	2	0 a 65535	%u
long / long int / signed long / signed long int	Inteiro longo com sinal	4	−2147483648 a 2147483647	%li ou %ld
unsigned long / unsigned long int	Inteiro longo sem sinal	4	0 a 4294967295	%lu
long long / signed long long / long long int / signed long long int	Inteiro muito longo com sinal	8	-2^{63} a $2^{63}-1$	%lli ou %lld
unsigned long long / unsigned long long int	Inteiro longo sem sinal	8	0 a $2^{64}-1$	%llu
float	Ponto flutuante simples	4	3.4×10^{-38} a $3.4 \times 10^{+38}$	%f ou %F
double	Ponto flutuante em precisão dupla	8	1.7×10^{-308} a $1.7 \times 10^{+308}$	%lf ou %lF
long double	Ponto flutuante em precisão estendida	16	3.4×10^{-4932} a $3.4 \times 10^{+4932}$	%Lf ou %LF

Tabela 1 – Tipos de dados básicos do C.

Vale notar que esses tipos podem variar de máquina para máquina, sendo interessante imprimir os limites dos tipos que estão no cabeçalho *limits.h*. Além disso, as padronizações (como ANSI e ISO, por exemplo) da linguagem também podem afetar certos tipos e, consequentemente, o funcionamento do código. Além disso, como C é muito popular, e por isso muitos compiladores foram construídos, com diferentes características. As próximas seções introduzirão o processo de compilação.

1.2 O QUE É UM COMPILADOR

O compilador é um programa de computador responsável por reescrever o código fonte em código de máquina que poderá ser executado no computador. Sendo assim, o compilador recebe como entrada um arquivo com o código fonte e gera na saída um arquivo que pode ser executado no computador, como mostra a seguinte imagem:

Figura 1 – Diagrama de funcionamento do compilador

Fonte: https://blog.betrybe.com/tecnologia/compilador-o-que-e/

Em outras palavras, o compilador traduz o código fonte de uma linguagem da qual os seres humanos entendem melhor para outra, que o computador consiga entender. Vale notar que hoje em dia, o compilador possui muito mais funcionalidades além de traduzir códigos. Ele permite enclausurar diversas instruções de máquina em uma única linha do código fonte, otimizar o código fonte, gerar arquivos intermediários, tratar erros na programação e dispor de ferramentas de depuração. Os primeiros compiladores eram focados em traduzir o código fonte e reunir as bibliotecas e rotinas necessárias para a execução do código objeto num processo chamado de ligação. Esses primeiros compiladores foram escritos inicialmente em Assembly, e, atualmente, existem diversas ferramentas para a construção de compiladores, o que facilitou a proliferação de novas linguagens.

Com a evolução das linguagens de programação e a necessidade de novas funcionalidades, os compiladores passaram a ter mais características e a funcionar de maneiras diferentes, e com isso foram desenvolvidos diversos tipos de compiladores. Abaixo seguem os principais tipos de compiladores:

- **Compilador *Ahead-of-time*** → Compilador padrão que compila o código fonte antes da execução do programa. A saída do compilador é uma arquivo objeto com instruções de máquina nativas;
- **Compilador *Just-in-time*** → A compilação aqui ocorre durante o tempo de execução do programa. Na primeira vez, compilador passa por cada linha do código fonte e traduz para instruções de máquina (ou para uma linguagem intermediária

que depois será traduzida para linguagem de máquina) que serão executadas imediatamente após a tradução, geralmente em uma máquina virtual. Na segunda vez que o programa será executado, ele já estará compilado (ou em uma linguagem intermediária de fácil tradução) tendo uma execução mais rápida. Diferente do caso anterior, onde o programa sempre será executado após a geração do arquivo objeto;

- **Compilador Cruzado** → Esse tipo de compilador gera um arquivo executável, através do código fonte, capaz de ser executado em outras máquinas. É interessante para aplicações como sistemas embutidos ou para uso em múltiplas máquinas;

- **Compilador *Source-to-source*** → Esse compilador tem como saída um arquivo contendo um código fonte de alto nível, ao invés de um arquivo com instruções de máquina. Isso permite que uma linguagem possua extensões sintáticas de interesse ao programador que podem ser escritas de maneira mais flexível, e que serão reescritas equivalentemente para o código fonte alvo. TypeScript é uma linguagem que utiliza esse tipo de compilação.

Existe um outro programa bastante comum que permite a execução do código fonte ou do *Bytecode* diretamente: o interpretador. Ao contrário do compilador, que gera um arquivo objeto com instruções de máquina, o interpretador traduz cada linha do código fonte em uma linguagem intermediária ou a executa diretamente, em uma máquina virtual. Isso torna o processo de execução de uma linguagem interpretada (Python, R, JavaScript, PHP, etc) mais lento pois cada linha do código precisa ser interpretada, para depois ser executada. Porém, a linguagem interpretada não precisa ser recompilada por inteiro por conta de algumas alterações no código.

Por fim, os programas que traduzem código em assembly para linguagem de máquina e vice-versa são chamados de montador (*assembler*) e desmontador (*disassembler*), respectivamente. O mesmo vale para o processo de construção de um código de alto nível a partir da linguagem de máquina, chamado de descompilação. Esse último possui aplicações em segurança, já que, com acesso ao código objeto, é possível identificar vulnerabilidades mais facilmente nas linguagens de alto nível geradas a partir da descompilação.

1.3 GNU COMPILER COLLECTION

O GNU Compiler Collection é uma coleção de compiladores do tipo *Ahead-of-time* do projeto GNU criada em 1987. Essa coleção possui compiladores para linguaguens como: ADA, C++, Fortran, Java, Objective-C e Pascal. Além disso, possui compatibilidade com muitas arquiteturas, como ARM, x86 e AMD64 (x86-64), sendo considerado favorito pelos desenvolvedores. Essa ferramenta vem como padrão na maioria dos sistemas Linux e é o compilador principal para o MAC OS, mas também pode ser utilizado em sistemas

Windows partir de ferramentas como MSYS2 e MinGW. A grande parte desses compiladores, atualmente, está escrita em C, inclusive o próprio compilador C, num processo chamado de *Bootstrapping*.

O foco se dará em um dos compiladores da GNU, o GNU C Compiler (gcc). Abaixo segue as etapas de compilação do gcc:

Figura 2 – Etapas da compilação

Fonte: https://guialinux.uniriotec.br/gcc/

1. **Pré-processamento** → Tarefa realizada pelo pré-processador que trata de todas as linhas que começam com "#". Duas diretivas principais são tratadas nessa fase: *include* e *define*. A primeira diretiva envolve a inclusão de arquivos de cabeçalho (.h) que possuem as definições e declarações de protótipos, enquanto que a segunda diretiva é utilizada para definir macros e constantes simbólicas. Ainda existem outras diretivas, como a diretiva para compilação condicional "#if...#else...#endif"e "#error", que pausa a compilação imprimindo uma mensagem de erro;

2. **Compilação** → Nessa fase que efetivamente começa, onde o código gerado é traduzido para assembly. A compilação do código feita em três níveis:

   a) Análise léxica: O compilador analisa os símbolos verificando se os nomes das variáveis, funções e palavras reservadas foram escritas corretamente, além de retirar os espaços em branco e comentários. Variáveis não definidas, nome de variável escrito de maneira incorreta, operadores inexistentes, strings e valores mal formados geram erros de compilação nesse nível;

   b) Análise sintática: Essa análise é feita em cima das expressões do C, que devem seguir a gramática formal. Uma expressão é formada por um ou mais símbolos, e, passada a fase anterior, todos esses símbolos foram escritos corretamente, porém podem estar organizados da maneira incorreta dentro de uma expressão em C. Como, por exemplo, utilizar operadores binários (var1 op var2) com um dos símbolos faltando ou abrir um parêntese/colchete/chave mas não fechar;

   c) Análise semântica: No nível final, o compilador analisa o sentido das expressões a partir de uma validação lógica. Uma das tarefas dessa análise consiste na checagem da consistência dos tipos nas expressões, regras de visibilidade e de contexto;

   d) Otimização de alto nível: Aqui, o compilador buscará otimizar o código fonte, geralmente, excluindo código redundante ou desnecessário.

3. **Montagem** → tradução de cada linha de assembly em código de máquina;

a) Otimização de baixo nível: Essa otimização é feita considerando o código objeto. Geralmente, se considera propriedades das operações envolvidas, deslocamento de código nos laços de repetição, substituição de chamadas de funções, que podem ser muito repetitivas, para o código *inline* (ou seja, substitui a chamada da rotina pelo próprio código), entre outros.

4. **Ligação** → A fase final da compilação ocorre na ligação, onde as bibliotecas e todo o código necessário para a execução do programa serão carregados e incluídos ao código objeto gerado na fase anterior.

Vale ressaltar que grande parte das otimizações estão desabilitadas por padrão, sendo necessário o uso de *flags* para habilitar essas otimizações. Depois dessa introdução aos conceitos iniciais acerca da linguagem C e do gcc, os próximos capítulos discutirão as formas de utilização desse compilador e as ferramentas que ele disponibiliza.

2 CAPÍTULO 2

2.1 COMPILANDO UM PROGRAMA EM C

Um dos primeiros programas que muitos programadores aprendem a escrever é o famoso "Hello, World!". A seguir, mostraremos como compilar esse programa em C usando o compilador gcc.

```c
#include <stdio.h>

int main()
{
    printf("Hello, World!");
    return 0;
}
```

helloworld.c

No entanto, o código-fonte ainda não pode ser entendido pelo computador, sendo necessário realizar a compilação do código para gerar um arquivo executável que possa ser executado. Para isso, usamos um compilador de C, como o gcc.

O processo de compilação do código em C é realizado pelo compilador, que transforma o código-fonte em um arquivo executável contendo as instruções que o computador deve seguir para executar o programa.

Para compilar o programa, é necessário informar o nome do arquivo que contém o código-fonte e o nome do arquivo executável que será gerado. No caso deste exemplo, o arquivo com o código-fonte tem o nome "helloworld.c" e o arquivo executável terá o nome "hello". A compilação será feita pelo terminal:

```
$ gcc helloworld.c -o hello
```

O parâmetro -o" indica que queremos criar um arquivo executável com o nome "hello", enquanto "helloworld.c" é o nome do arquivo que contém o código-fonte.

Além disso, é recomendável utilizar a flag -Wall" durante a compilação de programas em C. Essa opção habilita uma checagem mais rigorosa do código-fonte e gera avisos adicionais caso detecte possíveis problemas no código, como variáveis não inicializadas ou operações com ponteiros inválidos. Para utilizar a flag -Wall", basta incluí-la no comando de compilação:

```
$ gcc -Wall helloworld.c -o hello
```

Por fim, para executar o programa, basta digitar "./hello" no terminal. Esse comando informa ao sistema operacional que desejamos executar o arquivo "hello" que foi gerado pela compilação do programa "helloworld.c". Se tudo ocorrer bem, o programa exibirá a mensagem "Hello, World!" no terminal.

2.2 TRABALHANDO COM MÚLTIPLOS ARQUIVOS

Quando se está trabalhando em um projeto em C, é comum que este seja dividido em múltiplos arquivos. Essa prática permite uma melhor organização e compartimentalização do projeto, além de economizar tempo de compilação.

Por exemplo, digamos que queremos criar um programa que calcule o quadrado de um número. Podemos dividir o projeto em dois arquivos: um arquivo main.c, que contém a função principal do programa, e um arquivo quadrado.c, que contém a função que calcula o quadrado do número. Além disso, criaremos um arquivo quadrado.h, que contém somente a declaração da função que calcula o quadrado.

```c
#include <stdio.h>
#include "quadrado.h"

int main() {
    double x;
    printf("Digite um númeroç: ");
    scanf("%lf", &x);
    printf("O quadrado de %lf é %lf\n", x, quadrado(x));
    return 0;
}
```

main.c

```c
int calc_quadrado(int x) {
    return x * x;
}
```

quadrado.c

```c
int calc_quadrado(int x);
```

quadrado.h

É importante notar que, se incluíssemos diretamente o arquivo quadrado.c em main.c, haveria duas definições da função quadrado(double x), uma em cada arquivo, o que causaria um erro. Por isso, incluímos o arquivo de cabeçalho quadrado.h, que contém apenas a declaração da função, sem sua definição.

Podemos compilar o programa de uma vez só, escrevendo no terminal do Linux:

```
$ gcc -Wall main.c quadrado.c -o programa
```

E rodar o programa com:

```
$ ./programa
```

Para evitar a necessidade de compilar o programa inteiro sempre que houver uma alteração, podemos primeiro compilar cada arquivo em um arquivo de objeto. Isso pode ser feito através dos seguintes comandos:

```
$ gcc -Wall main.c -c
$ gcc -Wall quadrado.c -c
```

Isso irá gerar dois arquivos .o, main.o e quadrado.o.

Agora, podemos ligar os dois arquivos em um executável, sem precisarmos compilar o programa inteiro novamente:

```
$ gcc main.o quadrado.o -o programa
```

Com isso, podemos fazer alterações no arquivo quadrado.c, por exemplo, e compilar apenas esse arquivo, ligando-o depois aos demais arquivos já pré-compilados, sem a necessidade de recompilar o arquivo main.c.

2.3 TRABALHANDO COM BIBLIOTECAS EXTERNAS

explicar mais sobre o #include, por que preciso escrever -lm quando uso a <math.h>, etc

2.4 USANDO MAKEFILE

Ao trabalhar em projetos grandes com muitos arquivos, é difícil gerenciar e entender as dependências entre esses arquivos. É comum que apenas alguns arquivos precisem ser recompilados após alterações, e recompilar todos os arquivos novamente pode ser demorado e desnecessário. Para lidar com esses problemas, é possível usar o Make e o Makefile.

O Make é uma ferramenta que automatiza a compilação de programas a partir de arquivos-fonte. Ele trabalha a partir de um arquivo chamado Makefile, que especifica como os arquivos-fonte devem ser compilados em arquivos objeto e como esses objetos devem ser ligados para criar o programa final.

O Makefile é composto por regras. Cada regra especifica um alvo e suas dependências, ou pré-requisitos, seguido por comandos para criar o alvo. Quando um alvo é requisitado, o Make verifica se ele já existe e se é mais antigo que seus pré-requisitos. Se o alvo não existe ou é mais antigo que seus pré-requisitos, o Make executa a regra dos pré-requisitos primeiro, em ordem de dependência, e depois a regra do alvo para criar o alvo. Se o alvo já existe e não é mais antigo que seus pré-requisitos, o Make não executa a regra.

A estrutura básica de um Makefile é a seguinte:

```
alvo: dependencia
    comando

dependencia:
    comando
```

O alvo é o nome do arquivo que queremos criar ou atualizar, e a dependência é o nome do arquivo ou arquivos que precisam estar atualizados antes de criarmos o alvo. O comando é uma linha de código executada na linha de comando que compila os arquivos.

Vamos dar um exemplo simples:

```
saudacao:
    echo "Ola, mundo!"
```

Se o arquivo saudacao não existe, o comando echo "Olá, mundo!"será executado. Caso contrário, nada será feito. Mas se quisermos executar novamente o comando, podemos simplesmente chamar make saudacao na linha de comando. Mais em cima, escrevemos que o comando deve criar o arquivo alvo. No entando, o comando do arquivo saudacao somente escreve uma frase no terminal. Assim, como nenhum arquivo será criado, toda vez que Make for chamado, o mesmo comando será executado.

Agora, vamos dar um exemplo mais complexo:

makefile

```
hello: hello.o
    gcc hello.o -o hello

hello.o: hello.c
    gcc -Wall hello.c -c
```

O arquivo hello.c possui o seguinte código:

```c
#include <stdio.h>

int main()
{
    printf("Hello, World!\\n");
    return 0;
}
```

hello.c

Nesse exemplo, o objetivo é criar o arquivo executável hello. Ele depende do arquivo objeto hello.o, que por sua vez depende do arquivo fonte hello.c.

Quando chamamos make na linha de comando, o Makefile verificará o arquivo hello para ver se ele precisa ser recompilado. Se ele não existir ou for mais antigo que hello.o, o Make executará o comando gcc hello.o -o hello.

Antes de executar esse comando, o Make verificará o arquivo hello.o para ver se ele precisa ser recompilado. Se ele não existir ou for mais antigo que hello.c, o Make executará o comando gcc -Wall hello.c -c para gerar o objeto hello.o.

Portanto, o passo-a-passo seguido pelo Make é:

Verificar se hello precisa ser recompilado. Verificar se hello.o precisa ser recompilado. Executar o comando gcc -Wall hello.c -c se necessário. Executar o comando gcc hello.o -o hello para gerar o executável.

Ao usar um Makefile, podemos garantir que apenas os arquivos que precisam ser recompilados serão atualizados, economizando tempo e esforço. Além disso, a estrutura do Makefile ajuda a gerenciar as dependências de forma mais clara e organizada.

Assim, o uso de Makefiles é uma prática recomendada para projetos grandes e complexos em que a gestão de dependências e a eficiência na compilação são essenciais.

3 CAPÍTULO 3

3.1 BANDEIRAS DE AVISO DO COMPILADOR

O GCC compila programas usando o dialeto GNU da linguagem C como base, que incorpora o padrão ANSI C e várias extensões do GNU C. Essas extensões incluem recursos como declarações de variáveis no meio de um bloco de código, expressões com efeitos colaterais e construtores de atributos, entre outros. No entanto, programas válidos escritos em ANSI C podem conflitar com algumas extensões do GNU C.

Para lidar com esses conflitos, o GCC oferece várias opções de linha de comando, conhecidas como flags, que permitem aos desenvolvedores controlar o comportamento do compilador ao compilar o código fonte. A flag -ansi"é usada para desabilitar as extensões do GNU C que conflitam com o padrão ANSI C. A flag -pedantic"desabilita todas as extensões do GNU C, não apenas aquelas que conflitam com o padrão ANSI.

No capítulo anterior, usamos a flag -Wall na compilação do nosso programa. Essa flag é uma combinação de várias flags de aviso especializadas que detectam erros comuns de programação. Cada uma das flags contidas em -Wall pode ser usada individualmente. Algumas dessas flags são:

- Wcomment: avisa sobre problemas de formatação em comentários, como comentários dentro de comentários.
- Wformat: avisa sobre o uso incorreto de formatação em strings em funções como printf e scanf.
- Wunused: avisa sobre variáveis que foram declaradas mas não foram usadas no programa.
- Wimplicit: avisa sobre funções que foram usadas sem serem declaradas, o que pode acontecer se esquecer de incluir o arquivo de cabeçalho.
- Wreturn-type: avisa sobre funções que não retornam nenhum valor mas que não foram declaradas como "void".

O GCC também inclui outras flags de aviso úteis, como:

- W: uma flag geral, semelhante a -Wall, que avisa sobre diversos erros comuns.
- Wconversion: avisa sobre conversões implícitas de tipo, como entre float e integer, que podem causar resultados inesperados.
- Wshadow: avisa sobre a declaração de variáveis em um escopo em que elas já foram declaradas.

- Wtraditional: avisa sobre partes do código que seriam interpretadas de forma diferente por um compilador ANSI/ISO e um pré-ANSI.

Se você quiser que o programa pare de compilar se houver qualquer aviso das flags que você aplicou, use a flag -Werror.

Usar flags ao compilar um programa é uma boa prática, mas a grande quantidade pode tornar a escolha de quais usar difícil. Com isso em mente, recomendamos que, em geral, as seguintes flags sejam usadas:

```
$ gcc -ansi -pedantic -Wall -W
```

3.2 USANDO O PRÉ-PROCESSADOR

Como dito no capítulo inicial, pré-processador é o componente do GCC responsável por processar e manipular o código-fonte antes da compilação começar. Ele lê o código-fonte e realiza um conjunto de operações, incluindo substituição de macros, inclusão de arquivos e compilação condicional. Essas operações são definidas por um conjunto de diretivas, que são comandos especiais que começam com o símbolo "#".

Uma macro é um pedaço de código ao qual é dado um nome e definida por meio do uso da diretiva #define. Essa diretiva permite que você atribua um nome a um valor, uma expressão ou até mesmo a um bloco de código, tornando mais fácil e conveniente usar esse nome em vez de repetir o código completo sempre que for necessário. Abaixo está um exemplo de código que utiliza macros:

```
1 #include <stdio.h>
2
3 #define NUM 10
4
5 int main()
6 {
7     printf("The value of NUM is: %d\n", NUM);
8     return 0;
9 }
```

O pré-processador substitui as ocorrências da macro pelo seu conteúdo correspondente. Assim, quando compilarmos e executarmos o arquivo acima, será exibida a mensagem:

```
The value of NUM is: 10
```

Podemos querer executar uma parte do programa somente quando certa macro estiver definida. Para isso, delimitamos tal seção do código com as diretivas #ifdef e #endif:

Nesse caso, como a macro NUM não está definida no código fonte, nada será exibido ao executar o programa. No entanto, podemos definir a macro NUM quando compilarmos o programa. Para isso, usamos a opção '-DNAME' para definiremos a macro com nome NAME e, se quisermos designar um valor a ela, escrevemos '-DNAME=valor':

```
$ gcc -Wall -DNUM=5 programa.c -o programa
$ ./programa
The value of NUM is: 5
```

3.3 COMPILANDO PARA DEBUGAR (NOTAS POR ENQUANTO)

Existe uma forma de fazer debugging com a flag -g tal que, se o programa crashar, a parte problemática do código pode ser facilmente encontrada.

Esse capítulo é relativamente grande e usar exemplos vai ser bem importante. Escrever isso em um outro capitulo (chapter 4)

4 CAPÍTULO 4

4.1 COMPILANDO PARA DEPURAÇÃO

Durante o desenvolvimento de software, a depuração (debugging) desempenha um papel fundamental na identificação e correção eficiente de erros no código. Ao compilar programas em C, existe uma prática recomendada para facilitar a depuração: a utilização da opção -g ao chamar o GCC. Essa opção instrui o compilador a incluir informações de depuração no executável gerado. Essas informações, como símbolos de função, variáveis locais e localizações de linha, permitem uma análise detalhada do código durante a depuração. Por exemplo, ao executar o programa em um depurador, é possível definir pontos de interrupção, inspecionar valores de variáveis e rastrear a execução do código passo a passo.

Aqui está um exemplo de código que podemos usar para ilustrar como compilar programas em C para depuração:

```c
int foo (int *p);

int main (void)
{
    int *p = 0;
    /* ponteiro nulo */
    return foo (p);
}

int foo (int *p)
{
    int y = *p;
    return y;
}
```

exemplo.c

Neste código, temos duas funções: `main()` e `foo()`. A função `main()` inicializa um ponteiro `p` com o valor nulo (0) e, em seguida, chama a função `foo()` passando esse ponteiro como argumento. A função `foo()` recebe um ponteiro como parâmetro e tenta acessar o valor apontado por ele. No entanto, como p é nulo, essa operação resulta em um erro.

Para compilar esse código com a opção de depuração, você pode usar o seguinte comando:

```
$ gcc -Wall -g exemplo.c -o exemplo
```

Ao executá-lo, receberemos uma mensagem de erro, indicando que houve uma violação de segmentação (*segmentation fault*). Essa mensagem é exibida quando ocorre uma tentativa de acessar uma área da memória que não é permitida, como no caso em que o ponteiro `p` aponta para o valor nulo.

```
$ ./exemplo
segmentation fault (core dumped)
```

Vamos falar sobre o arquivo `core` mencionado na mensagem de erro. Ele é um arquivo de despejo de memória que pode ser gerado quando ocorre uma falha grave em um programa e contém informações sobre o estado da memória no momento da falha, sendo útil para analisar e depurar o problema. Nem todos os sistemas geram automaticamente o `core` por padrão. Se ele não for gerado, essa funcionalidade pode ser habilitada executando o comando abaixo:

```
$ ulimit -c unlimited
```

Esse comando define temporariamente o tamanho máximo do arquivo core como ilimitado, permitindo a geração do `core` em caso de falha no programa. É importante ressaltar que, se ele não tiver sido gerado durante a primeira execução do programa, `exemplo` terá que ser executado novamente.

4.2 DEPURANDO PROGRAMAS

5 COMPILANDO PARA A DEPURAÇÃO

Geralmente, os arquivos executáveis não incluem informações ou referências ao código-fonte original do programa. Isso pode ser inadequado para fins de depuração, já que não teremos meios de identificar a causa de um erro caso o programa pare de funcionar inesperadamente (crash).

O GCC fornece a flag de compilação '-g' responsável por armazenar informações adicionais sobre a depuração do programa em arquivos executáveis e de objeto. Desse modo, podemos retornar à linha do programa que originouno erro.

5.1 EXAMINANDO ARQUIVOS CENTRAIS

Além de permitir que programas sejam executados sob o depurador, um benefício importante da opção '-g' é a capacidade de examinar a causa de uma falha de programa a partir de um 'despejo de núcleo'. Quando um programa termina de forma anormal (ou seja, falha), o sistema operacional pode criar um arquivo de núcleo (geralmente chamado de 'core'), que contém o estado na memória do programa no momento da falha. Esse arquivo é frequentemente chamado de despejo de núcleo. Combinado com informações da tabela de símbolos produzida pelo '-g', o despejo de núcleo pode ser usado para encontrar a linha onde o programa parou e os valores de suas variáveis nesse ponto.

Isso é útil tanto durante o desenvolvimento de software quanto após a implantação, pois permite investigar problemas quando um programa falha 'no campo'.

5.2 EXIBINDO UMA PILHA DE CHAMADAS

O depurador também pode mostrar as chamadas de função e os argumentos até o ponto atual de execução, isso é chamado de pilha de chamadas e é exibido com o comando backtrace:

```
1 #0 0x080483ed em foo (p=0x0) em null.c:13
2 #1 0x080483d9 em main () em null.c:7
```

(gdb) backtrace

Neste caso, a pilha de chamadas mostra que a falha ocorreu na linha 13 após a chamada da função foo a partir de main com um argumento de p=0x0 na linha 7 em 'null.c'. É possível mover-se para diferentes níveis na pilha de chamadas e examinar suas variáveis usando os comandos up e down do depurador.

6 COMPILANDO COM OTIMIZAÇÃO

O GCC é um compilador otimizador que pode gerar arquivos executáveis mais rápidos e/ou menores, levando em consideração as características do processador alvo e a ordem das instruções.

A otimização é um processo complexo que envolve a escolha da melhor combinação de instruções de máquina para cada comando de alto nível no código-fonte. Diferentes códigos devem ser gerados para processadores distintos, devido ao uso de linguagens de montagem e máquina incompatíveis. Além disso, cada tipo de processador possui características próprias, como o número de registradores disponíveis, que afetam a forma como o código é gerado. Ao compilar com otimização, o GCC leva em consideração todos esses fatores.

Aqui estão alguns dos fatores considerados pelo GCC durante a otimização do código:

- O tipo de processador no qual o código será executado.
- O número de registradores disponíveis no processador alvo.
- A velocidade das diferentes instruções no processador alvo.
- A ordem de execução das instruções.

Ao levar em conta esses fatores, o GCC é capaz de gerar código otimizado para um processador específico, visando uma execução mais rápida.

6.1 OTIMIZAÇÃO EM NÍVEL DE CÓDIGO-FONTE

A otimização em nível de código-fonte melhora o desempenho de um programa por meio de alterações no código-fonte. Duas otimizações comuns são a eliminação de subexpressões repetidas e o inline de funções.

6.2 ELIMINAÇÃO DE SUBEXPRESSÕES COMUNS

A eliminação de subexpressões repetidas evita a reavaliação de uma mesma expressão várias vezes. Por exemplo, a expressão $x = \cos(v).(1 + \sin(u/2)) + \sin(w).(1 - \sin(u/2))$ pode ser reescrita como $t = \sin(u/2); x = \cos(v).(1 + t) + \sin(w).(1 - t)$, evitando a avaliação duplicada de $\sin(u/2)$.

6.3 INCLUSÃO DE FUNÇÃO

O inline de funções substitui uma chamada de função pelo seu próprio corpo, reduzindo a sobrecarga das chamadas de função. Por exemplo, a função sq(x) pode ser inlineada neste loop:

```
1 for (i = 0; i < 1000000; i++)
2   sum += sq(i + 0.5);
```

função sq(x)

Isso substitui o loop interno pelo corpo da função sq(x), melhorando o desempenho ao evitar chamadas de função.

O GCC utiliza heurísticas para decidir quais funções devem ser inlineadas. A palavra-chave "inline"pode ser usada para solicitar explicitamente a inlineação de uma função específica.

6.4 TRADE-OFFS DE VELOCIDADE E ESPAÇO

Algumas formas de otimização podem aumentar a velocidade e reduzir o tamanho do programa simultaneamente, enquanto outras produzem código mais rápido em troca de um executável maior. Isso é conhecido como trade-off de velocidade e espaço. Essas otimizações também podem ser usadas ao contrário, diminuindo o tamanho do executável em detrimento da velocidade de execução.

6.5 DESENROLAMENTO DE LOOPS

O desenrolamento de loops é uma otimização que aumenta a velocidade dos loops eliminando a condição de "fim do loop"em cada iteração. Ele permite atribuições diretas, sem a necessidade de testes, resultando em uma execução mais rápida. O desenrolamento de loops pode aumentar o tamanho do executável, exceto em loops muito curtos.

6.6 AGENDAMENTO

O agendamento é o nível mais baixo de otimização, onde o compilador determina a melhor ordem de execução das instruções individuais. Ele melhora a velocidade do executável sem aumentar seu tamanho, mas requer memória adicional e tempo durante o processo de compilação.

6.7 NÍVEIS DE OTIMIZAÇÃO NO GCC:

O GCC oferece diferentes níveis de otimização (0 a 3) para controlar o tempo de compilação, uso de memória do compilador e o trade-off entre velocidade e espaço no executável resultante. Os níveis de otimização são:

- '-O0' (padrão): Sem otimização, compilando de forma direta para depuração.
- '-O1': Otimizações comuns sem trade-offs de velocidade e espaço.
- '-O2': Otimizações adicionais sem aumentar o tamanho do executável.

- '-O3': Otimizações mais custosas que podem aumentar o tamanho do executável.
- '-funroll-loops': Desenrolamento de loops, aumentando o tamanho do executável.
- '-Os': Otimizações para reduzir o tamanho do executável.

É importante considerar os custos das otimizações, como maior complexidade na depuração e maior tempo/memória de compilação. Geralmente, '-O0' é usado para depuração e '-O2' para desenvolvimento e implantação.

6.8 6.6 OTIMIZAÇÃO E DEPURAÇÃO

Com o GCC, é possível usar otimização em combinação com a opção de depuração '-g'. Muitos outros compiladores não permitem isso. Ao usar depuração e otimização juntas, as reorganizações internas feitas pelo otimizador podem dificultar a compreensão do que está acontecendo ao examinar um programa otimizado no depurador. Por exemplo, variáveis temporárias geralmente são eliminadas e a ordem das instruções pode ser alterada. No entanto, quando um programa trava inesperadamente, qualquer informação de depuração é melhor do que nenhuma, portanto, o uso de '-g' é recomendado para programas otimizados, tanto para desenvolvimento quanto para implantação. A opção de depuração '-g' é habilitada por padrão para versões dos pacotes GNU, juntamente com a opção de otimização '-O2'.

6.9 6.7 OTIMIZAÇÃO E AVISOS DO COMPILADOR

Quando a otimização é ativada, o GCC pode produzir avisos adicionais que não aparecem ao compilar sem otimização.

Como parte do processo de otimização, o compilador examina o uso de todas as variáveis e seus valores iniciais - isso é chamado de análise de fluxo de dados. Isso serve como base para outras estratégias de otimização, como agendamento de instruções. Um efeito colateral da análise de fluxo de dados é que o compilador pode detectar o uso de variáveis não inicializadas.

A opção '-Wuninitialized' (incluída em '-Wall') avisa sobre variáveis que são lidas sem serem inicializadas. Ela só funciona quando o programa é compilado com otimização, para que a análise de fluxo de dados seja ativada. A seguinte função contém um exemplo de tal variável:

```
int sign(int x)
{
    int s;
    if (x > 0)
        s = 1;
    else if (x < 0)
```

```
        s = -1;
    return s;
}
```

A função funciona corretamente para a maioria dos argumentos, mas tem um bug quando x é zero - nesse caso, o valor de retorno da variável s será indefinido.

Compilar o programa apenas com a opção '-Wall' não produz nenhum aviso, porque a análise de fluxo de dados não é realizada sem otimização:

```
$ gcc -Wall -c uninit.c
}
```

Para gerar um aviso, o programa deve ser compilado com '-Wall' e otimização simultaneamente. Na prática, o nível de otimização '-O2' é necessário para obter bons avisos:

```
$ gcc -Wall -O2 -c uninit.c
uninit.c: In function 'sign':
uninit.c:4: aviso: 's' might be used uninitialized in this function
}
```

Isso detecta corretamente a possibilidade de a variável s ser usada sem ser definida.

Observe que, embora o GCC geralmente encontre a maioria das variáveis não inicializadas, ele faz isso usando heurísticas que ocasionalmente podem perder alguns casos complicados ou emitir falsos avisos sobre outros. Nessa última situação, muitas vezes é possível reescrever as linhas relevantes de maneira mais simples que remove o aviso e melhora a legibilidade do código-fonte.