Linguagem C

Walter Fetter Lages

w.fetter@ieee.org

Universidade Federal do Rio Grande do Sul

Escola de Engenharia

Departamento de Engenharia Elétrica

Microprocessadores II

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Introdução

- C foi criada e implementada por Dennis Ritchie, em um sistema operacional UNIX
- C foi criada a partir da linguagem B, derivada da linguagem BCPL
- O sistema operacional UNIX system V foi implementado em C
- Implementações bastante compatíveis
  - Portabilidade
- Padrão ANSI

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Características

- C é um linguagem fracamente tipificada
  - As variáveis precisam ser declaradas antes de serem utilizadas
  - Conversões de tipos automáticas
- Pouca verificação de erros em tempo de execução
- Facilidades para manipulação de bits, bytes e endereços
- Apenas 32 palavras reservadas (padrão ANSI)
- É considerada uma linguagem estruturada
  - A rigor, não é, pois não permite a declaração de sub-rotinas dentro de sub-rotinas
- É uma linguagem para programadores

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Fundamentos

- C é sensível ao caso
  - Por convenção, variáveis e sub-rotinas são declaradas em letras minúsculas, constantes são declaradas em letras maiúsculas
- O início do programa é na função `main()`
- Existe uma biblioteca padrão definida pela ANSI
- Existe um pré-processador

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Palavras Reservadas

- Tipos de dados: `char int float double void`
- Modificadores de tipos de dados: `signed unsigned long short`
- Classes de armazenamento: `auto const extern register static volatile`
- Declaração de tipos: `enum struct union typedef`
- Controle de fluxo: `return if else for do while continue switch case break default goto`
- Operador: `sizeof`

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Identificadores

- Nomes de variáveis, funções, estruturas...
- Definidas pelo programador
- Um ou mais caracteres (letras números e subscrito), sem espaços
- O primeiro caractere deve ser uma letra ou subscrito

Declaração de Variáveis

```
int i;
int i,j,l;
short int si;
short si;
unsigned int ui;
unsigned ui;
double soma;
unsigned long int ul;
unsigned long ul;
char c;
unsigned char c;
long double ld;
```

Declaração de Variáveis

```
const long double um=1.0;
register int i;
volatile int i;
static int i;
extern char c;
```

- Variáveis declaradas fora de funções são variáveis globais e são alocadas na área (segmento) de dados
- Variáveis declaradas dentro de funções são variáveis locais e são alocadas na pilha
- Variáveis podem ser declaradas no início de cada escopo
- Variáveis podem ser inicializadas na declaração

Exemplo de Declaração de Variáveis

```
#include <stdio.h>
int sum; /* variavel global */

int main(void)
{
        int count; /* variavel local */
        sum=0; /* inicializacao */

        for(count=0;count < 10; count++) total(count);
        printf("A soma e': %d\n",sum);
        return 0;
}

void total(int x)
{
         sum=sum+x; /* x e' um parametro */
}
```

Caracteres Especiais

Código	Significado
`\b`	retrocesso
`\f`	alimentação de formulário
`\n`	nova linha
`\r`	retorno do carro
`\t`	tab horizontal
`\"`	aspas duplas
`\'`	aspas simples
`\0`	zero
`\\`	barra invertida
`\v`	tab vertical
`\a`	alerta
`\o`	constante octal
`\x`	constante hexadecimal

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Operadores

- Aritméticos: `+ - * / % ++ -- = += -= *= /=`
- Relacionais: `> < >= <= == !=`
- Lógicos: `! && ||`
- *bitwise*: `~ & | ^ « »`
- Outros: `() [] * & sizeof (type) , ? : . ->`

Precedência de Operadores

```
() [] . ->
! ~ ++ -- - (type) * & sizeof
* / %
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
?
= += -+ *= /=
,
```

Atribuição e *Cast*

- Em C o operador de atribuição (=) resulta o valor que foi atribuído

```
if(i=malloc(100)==NULL) return -1;
```

- Para converter tipos de variáveis utiliza-se *casts*

```
int i=1;
int j;
double k;

j=i/3;
k=i/3;
k=i/3.0;
k=(double)i/3;
```

Ponteiros

- O operador & é utilizado para obter o endereço de uma variável, ou seja um ponteiro para a variável
- O operador `*` é utilizado para declarar uma variável do tipo ponteiro e para obter o dado apontado por um ponteiro (derreferenciar o ponteiro)

```
int i=4;
int j;
int *iptr;

iptr=&i;

j=*iptr+1;
```

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Arrays

- Em C, os *arrays* tem como primeiro índice 0
- Não existem matrizes, mas pode-se declarar *arrays* de *arrays*
- Não há verificação de limites
- Uma string é um *array* de caracteres terminado pelo byte 0, ou o caractere `\0`
- O nome do *array* é um ponteiro para o seu início

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Exemplo *Array*

```
int amostras[10];
double x[3]={0.0,1.0,2.0};
int i;
int *s;

amostras[1]=i;

i=amostras[1];
i=*(amostras+1);

s=amostras;
i=*amostras;
i=*(s+1);
i=s[1];
```

Funções

- Funções devem ter um tipo de retorno e parâmetros de algum tipo
- O tipo de retorno e dos parâmetros pode ser `void`
- Funções cujo tipo de retorno não sejam `void` devem terminar com o comando return.

```
int soma1(int a,int b)
{
        int s;

        s=a+b;
        return s;
}
```

Passagem de Parâmetros

```
#include <stdio.h>

int main(int argc,char *argv[])
{
    int a;
    int b;
    int s;

    a=atoi(argv[1]);
    b=atoi(argv[2]);
    s=soma1(a,b);
    printf("%d+%d=%d\n",a,b,s);
    return 0;
}
```

Passagem de Parâmetros

- Em C os parâmetros são sempre passados por valor
- As alterações nos parâmetros feitas dentro das funções não se refletem fora
- Pode-se usar ponteiros para simular passagem de parâmetros por referência

```
void soma2(int a,int b,int *s)
{
        *s=a+b;
}
```

Passagem de Parâmetros

```
#include <stdio.h>

int main(int argc,char *argv[])
{
    int a;
    int b;
    int s;

    a=atoi(argv[1]);
    b=atoi(argv[2]);
    soma2(a,b,&s);
    printf("%d+%d=%d\n",a,b,s);
    return 0;
}
```

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Protótipo de Função

- É a declaração da função, com tipo de retorno, nome e parâmetros
- Especifica a interface para uso da função
- É altamente recomendado que os protótipos das funções sejam declarados antes que elas sejam chamadas
  - Definindo a função antes da chamada
  - Declarando o protótipo antes da chamada
  - Declarando o protótipo em um arquivo *header*
  - Caso contrário, o compilador assume que o tipo de retorno e de todos os parâmetros é `int`

Exemplos de Protótipos

```
int main(int argc,char *argv[]);
int soma1(int a,int b);
void soma2(int a,int b,int *s);
```

Função `main()`

```
int main(int argc,char *argv[])
```

- A função `main()` recebe dois parâmetros e retorna um inteiro
  - Código de final de programa
- `argc` é o número de parâmetros passados na linha de comando de execução do programa
  - O próprio nome do programa é contado como um dos parâmetros
- `argv` é um *array* de *strings*
  - Cada elemento do *array* é um argumento da linha de comando de execução do programa

Comando return

- return;
- return x;

```
int soma(int a,int b)
{
        return a+b;
}

void erro(int codigo)
{
        if(!codigo) return;
        printf("erro %d\n",codigo);
}
```

Comandos `if` e `else`

- `if(condicao) comando; else comando;`
- O `else comando;` é opcional
- O `comando` pode ser um bloco de comandos
- A condicao pode ser qualquer expressão que resulte em zero ou não-zero

```
if(codigo==0) printf("Nao houve erro\n");
else printf("Erro %d\n",codigo);

if(codigo)
{
        printf("Erro %d\n",codigo);
        return -codigo;
}
```

Comando `for`

- `for(inicializacao;condicao;incremento) comando;`
- `inicializacao`, `condicao` e `incremento` não precisam necessariamente estar relacionadas, podem ser três comandos quaisquer

```
for(x=0;x < 100;x++) printf("%d\t",x);

for(;;)
{
        c=getchar();
        if(c==0x2b) break;
        printf("%c",c ^ ('a' ^ 'A'));
}
```

Comando while

- while(condicao) comando;

```
char c='\0';
while(c!=0x2b) c=getchar();

char c=0;
while(c!=0x2b)
{
        c=getchar();
        printf("%c",c ^ ('a' ^ 'A'));
}
```

Comando do while

- do comando while(condicao);

```
do c=getchar() while(c!=0x2b);

do
{
        c=getchar();
        printf("%c",c ^ ('a' ^ 'A'));
} while(c!=0x2b);
```

Comando `continue`

- continue;
- `continue` pode ser utilizado com qualquer tipo de laço: `for`, `while` ou `do while`

```
do
{
        c=getchar();
        if(c < 'a' || c > 'z') continue;
        printf("%c",c ^ ('a' ^ 'A'));
} while(c!=0x2b)
```

Comando `break`

- `break;`
- `break` pode ser utilizado com qualquer tipo de laço: `for`, `while` ou `do while`
- `break` também é utilizado para evitar *fall-through* no comando `switch`

```
for(;;)
{
        c=getchar();
        if(c==0x2b) break;
        printf("%c",c ^ ('a' ^ 'A'));
}
```

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Comandos `switch`, `case` e `default`

```
switch(variavel)
{
        case constante1: sequencia_de_comandos;
        case constante2: sequencia_de_comandos;


        default: sequencia_de_comandos;
}
```

- Podem existir quantos `case` forem necessários
- O `default` é opcional e não precisa ser o último
- Ocorre *fall-through* de um `case` para outro
- Usualmente se utiliza o `break` para evitar *fall-through*

Comandos switch, case e default

```
switch(c)
{
        case 0x2b: return;
        case '\0':
        case '\n':
        case '\t':
                printf("%c",c);
                break;
        default:
                printf("%c", c ^ ('a' ^ 'A');
}
```

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Comando `goto`

- `goto rotulo;`
- Não é um comando necessário, sempre pode-se fazer o mesmo sem utiliza-lo
- Tende a tornar o programa confuso
- Em geral é conveniente apenas para se sair de muitos laços aninhados em uma condição de erro

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Comando goto

```
for(;;)
{
    for(;;)
    {
        while(1)
        {

            if(erro) goto fim;
        }
    }
}
fim: printf("erro");
```

UFRGS
UNIVERSIDADE FEDERAL
DO RIO GRANDE DO SUL

Tipos Definidos pelo Usuário

- Estruturas
- Campos de bit
- Uniões
- Enumerações

Estruturas

```
struct nome
{
        tipo variavel1;
        tipo variavel2;

} variavel;
```

- Pode-se declarar a estrutura sem declarar a variável
- Se for declarada a variável, a estrutura não precisa ter nome
- Estruturas são passadas por valor para funções
- Normalmente é passado um ponteiro para a estrutura

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Estruturas

```
struct FICHA
{
        char nome[30];
        double altura;
} dados;

struct FICHA cadastro[500];

dados.altura=1.8;
cadastro[20].altura=1.8;
imprime(dados);
atualiza(&dados);
atualiza(&cadastro[20]);
```

Estruturas

```
void imprime(struct FICHA x)
{
    printf("nome: %s\n",x.nome);
    printf("altura: %f\n",x.altura);
}

void atualiza(struct FICHA *x)
{
    (*x).altura=1.8;
    x->altura=1.8;
}
```

Campos de Bit

- ```
  struct nome
  {
        tipo variavel1:comprimento1;
        tipo variavel2:comprimento2;

  } variavel;
  ```
- Pode-se declarar o campo de bit sem declarar a variável
- Se for declarada a variável, o campo de bit precisa ter nome
- Pode-se declarar bits sem nome, apenas com o comprimento
- A implementação é dependente de máquina e compilador
- É mais comum utilizar máscaras ao invés de campos de bit
  ```

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Campos de Bit

```
struct ESTADO
{
        unsigned int ativo:1;
        unsigned int:4
        unsigned int pronto:1;
} dispositivo;

struct ESTADO disp2;

dispositivo.ativo=1;
disp2.pronto=0;
imprime(dispositivo);
atualiza(&dispositivo);
```

Campos de Bit

```
void imprime(struct ESTADO x)
{
    if(x.ativo) printf("ativo\t");
    if(x.pronto) printf("pronto\n");
}

void atualiza(struct ESTADO *x)
{
    (*x).ativo=1;
    x->pronto=0;
}
```

Uniões

```
union nome
{
        tipo variavel1;
        tipo variavel2;

} variavel;
```

- Pode-se declarar a união sem declarar a variável
- Se for declarada a união, a união não precisa ter nome
- Uniões são passadas por valor para funções
- Normalmente é passado um ponteiro para a união

Uniões

```
union REG
{
        struct
        {
                unsigned char l;
                unsigned char h;
        } byte;
        unsigned short x;
        unsigned int ex
} a;

union REG b, c, d;

a.byte.l=0x03;
a.byte.h=0x05;
a.x=0x0503;
a.ex=0x00000503;
```

Enumerações

- ```
  enum nome
  {
        constante1=valor,
        constante2=valor

  } variavel;
  ```
- Pode-se declarar a enumeração sem declarar a variável
- Se for declarada a variável, a enumeração não precisa ter nome
- Não é necessário definir um valor para cada constante
- Os valores devem ser inteiros
  ```

Enumerações

```
enum MOEDA
{
        REAL,
        DOLAR,
        EURO=100,
        PESO,
        YEN
} dinheiro;

union MOEDA verba;

dinheiro=REAL;
verba=DOLAR;
imprime(dinheiro);
atualiza(&verba);
```

Enumerações

```
void imprime(union MOEDA x)
{
    printf("Moeda: %d\n",x);
}

void atualiza(union MOEDA *x)
{
    *x=REAL;
}
```

typedef

- `typedef tipo nome`
- Pode-se dar nomes aos tipos e com isto criar novos tipos de dados semelhantes aos tipos de dados nativos

```
typedef double real;
typedef enum MOEDA moeda;
typedef union REG reg;

real a;
moeda dinheiro;
reg r;
```

Operador ?

- `condicao?  expressao1:expressao2`
- É uma forma abreviada de `if(condicao) comando; else comando;`

```
x=x > 10 ? 10:x;
x=(x < 0)? 0:x;
```

Formas Abreviadas

```
i++;
i--;
i+=10;
i-=10;
i*=10;
i/=10;
i=j=5;
```

Operador ,

- O operador `,` é um separador de comandos, enquanto `;` é um terminador de comandos
- O lado esquerdo do operador `,` sempre tem o valor `void`, de forma que o valor da expressão completa é o valor da expressão do lado direito do operador `,`

```
for(x=0,y=0;x < 10 && y < 20;x++,y+=3)
        printf("%d %d\n",x,y);
x=(y=3,y+1);
```

Préprocessador

- Comandos executados em tempo de compilação e não em tempo de execução

```
#if expressao
#ifdef identificador
#ifndef identificador
#else
#elif expressao
#endif
#include <arquivo>
#define identificador string
#undef identificador
#line numero [arquivo]
#error mensagem
#pragma nome
```

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Programas Multi-Módulos

- Programas grandes são organizados em diversos módulos
  - Um módulo é a unidade mínima de linkagem
  - Cada módulo é compilado separadamente dos demais
- Cada arquivo `.c` compilado torna-se um módulo
- Módulos passíveis de reutilização são usualmente agrupados em bibliotecas
- Arquivos *header* são utilizados para definir os protótipos das funções públicas dos módulos e constantes associadas à utilização destas funções
- Não existe uma associação direta entre *headers*, módulos e bibliotecas

Exemplo

- Arquivo `soma.h`

```
#ifndef _SOMA_H
#define _SOMA_H
extern int soma1(int a,int b);
#endif
```

- Arquivo `soma.c`

```
#include <soma.h>
int soma1(int a,int b)
{
        int s;

        s=a+b;
        return s;
}
```

Exemplo

- Arquivo `teste.c`

```
#include <stdio.h>
#include <soma.h>

int main(int argc,char *argv[])
{
    int a;
    int b;
    int s;

    a=atoi(argv[1]);
    b=atoi(argv[2]);
    s=soma1(a,b);
    printf("%d+%d=%d\n",a,b,s);
    return 0;
}
```

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Exemplo

- Arquivo `Makefile`

```
CFLAGS=-O2 -Wall
CINCLUDE=-I. -I${HOME}/include
CLIBDIR=-L${HOME}/lib
CLIBS=

CPPFLAGS=
CPPINCLUDE=-I${HOME}/include/cpp -I../include
CPPLIBDIR=-L../lib
CPPLIBS=

INCLUDE=${CINCLUDE} ${CPPINCLUDE}
FLAGS= ${CFLAGS} ${CPPFLAGS}
LIBDIR=${CLIBDIR} ${CPPLIBDIR}
LIBS=${CPPLIBS} ${CLIBS}

CMP= gcc
CMPFLAGS= ${FLAGS} ${INCLUDE}
LDFLAGS= ${LIBDIR} ${LIBS}
```

Exemplo

- Arquivo `Makefile` (Continuação)

```
all: teste

teste: teste.o soma.o
	${CMP} ${CMPFLAGS} -o $@ $^ ${LDFLAGS}

teste.o: teste.c soma.h
	${CMP} ${CMPFLAGS} -c -o $@ $<

soma.o: soma.c soma.h
	${CMP} ${CMPFLAGS} -c -o $@ $<

clean:
	rm -f *~ *.bak *.o

install:

distclean: clean
	rm -f teste
```

Outro Makefile

```
TARGET=teste
SRCS=$(TARGET).c soma.c

PREFIX=/usr/local
BINDIR=$(PREFIX)/bin

FLAGS=-Wall -g -MMD
INCLUDE=-I. -I$(HOME)/include
LIBDIR=-L$(HOME)/lib
LIBS=

CC=gcc
CFLAGS=$(FLAGS) $(INCLUDE)
LDFLAGS=$(LIBDIR) $(LIBS)
```

Outro Makefile (continuação)

```
all: $(TARGET)

$(TARGET): $(SRCS:.c=.o)
        $(CC) -o $@ $^ $(LDFLAGS)

%.o: %.c
        $(CC) $(CFLAGS) -c -o $@ $<

-include $(SRCS:.c=.d)

clean:
        rm -f *~ *.bak *.o *.d

distclean: clean
        rm -f $(TARGET)

install: $(TARGET)
        install -m 755 $(TARGET) $(BINDIR)

uninstall:
        rm -f $(BINDIR)/$(TARGET)
```