

Linguagem C

Uma revisão com foco em sistemas embarcados

Marcelo Barros e Daniel Carvalho

UFU/FEELT

26 de setembro de 2022

Outline

Fundamentos

Versões utilizadas

- ▶ Revisão C11 (ISO/IEC 9899:2011)
- ▶ Compilador GNU GCC

Linha do tempo da linguagem C

B (1972): Primeira implementação, Dennis Ritchie and Ken Thompson e colegas para o PDP11.

K&R (1978): Primeira especificação informal.

C89/C90 (1989/1990): Adoção como padrão pela ANSI (C89) e depois pela ISO (C90).

C99 (1999): Primeira grande revisão do padrão, amplamente utilizada.

C11 (2011): Segunda revisão da linguagem. Aproximação ao C++. Uso moderado mas crescendo.

C17 (2018): Sem características novas. Apenas correções.

C2x (2023?) ¹

¹<https://en.wikipedia.org/wiki/C2x>

Existe mesmo uma linguagem “Embedded C” ?

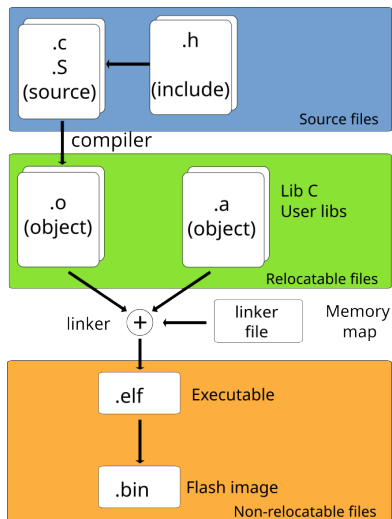
- ▶ Norma: *Programming languages — C — Extensions to support embedded processors* ISO/IEC TR 18037:2008
- ▶ Tecnicamente é apenas uma extensão do C, cobrindo:
 - ▶ Aritmética de ponto fixo
 - ▶ Espaço de endereçamento
 - ▶ Endereçamento de hardware básico para I/O

Palavras Reservadas

- ▶ Não devem ser usadas como nomes de funções ou variáveis
- ▶ São *case sensitives*
- ▶ Em azul, as adições do C11 em relação ao C99

```
auto break case char const continue default  
do double else enum extern float for goto  
if inline int long register restrict return  
short signed sizeof static struct switch  
typedef union unsigned void volatile while  
_Bool _Complex _Imaginary  
_Alignas _Alignof _Atomic _Generic  
_Noreturn _Static_assert _Thread_local
```

Estrutura e organização de uma aplicação



Organização dos arquivos de inclusão

```
#ifndef __DEMO_H__
#define __DEMO_H__

#ifdef __cplusplus
extern "C" {
#endif

// defines

// tipos de dados compartilhados
// (estruturas, unioes, typedefs)

// prototipo de funcoes

#ifdef __cplusplus
}
#endif

#endif /* __DEMO_H__ */
```

- ▶ Pense como API: só exporte interfaces e o que elas precisarem !
- ▶ Não esqueça a proteção de inclusão recursiva !
- ▶ Não é ANSI-C mas `#pragma once` pode ser interessante
- ▶ Evite colocar arquivos de inclusão em `.h` (boa prática)
- ▶ E o `__cplusplus` e `extern "C"` ?

Name mangling ou name decoration

- ▶ É uma forma de adicionar informações adicionais a nomes de funções, variáveis, etc, de forma a remover ambiguidade e gerar informação adicional ao linker².
- ▶ Por exemplo, como o compilador irá gerar nomes únicos para uma função com overloading em C++ ?

```
void accel_read(float &x, float &y, float &x) {}  
void accel_read(accel_t &data) {};
```

- ▶ Nomes decorados gerados pelo GCC (objdump -d <binário>):

```
00000000000001149 <_Z9accel_readRfS_S_>:  
00000000000001160 <_Z9accel_readR7accel_s>:
```

Name mangling ou name decoration

- ▶ Logo, se você linkar um programa em C com outras partes em C++, o compilador C++ irá gerar um padrão de name decoration para as suas funções em C que é diferente dos símbolos gerados pelo C !
- ▶ A convenção de chamada padrão do C (`_cdecl`)³ apenas adiciona “_” antes do nome das funções.
- ▶ Assim, se declarar a função como `extern "C" void func(void);` você está explicitando que a decoração de `func` segue o padrão do C (`_func`).
- ▶ O `__cplusplus` é a forma de identificar se quem está compilando o arquivo é o compilador C ou C++.

³ https://en.wikipedia.org/wiki/Name_mangling

E o extern ?

Para variáveis: diz para o compilador que a variável citada existe em algum lugar, não aloca memória para ela. O linker vai cuidar de “encontrar” onde a declaração foi feita.

```
extern uint8_t buffer[16];
```

Para funções: explicita que a função tem linkagem externa, ou seja, o linker vai deixá-la disponível para ser “encontrada” por qualquer arquivo C. Esse é o default da linguagem. É o inverso do “static”.

Atenção: `extern "C" <nome>` é um dialeto C++ !

Arquivos de inclusão: modelo

```
#ifndef __ACCEL_H__
#define __ACCEL_H__

#ifdef __cplusplus
extern "C" {
#endif

#define ACCEL_NUM_AXIS 3

typedef struct accel_data_s
{
    float axis[ACCEL_NUM_AXIS];
} accel_data_t;

void accel_read(accel_data_t *data);
void accel_init(void);

#ifdef __cplusplus
}
#endif

#endif /* __ACCEL_H__ */
```

Especificadores de classe de armazenamento

~~extern~~ static auto register

- ▶ Existem outros especificadores !
- ▶ No entanto, precisamos entender antes como as variáveis são armazenadas e as seções de código !

Seções de código e dados do programa

Quando o programa é compilado, ele é dividido em **seções**:

- text (flash)**: Onde é armazenado o código executável de fato, geralmente em flash
- rodata (flash)**: Constantes do código (read only)
- bss (RAM)**: Local das variáveis que vão ser inicializadas com zero na partida (o usuário não fez uma inicialização de valor explícita)
- data (RAM)**: Locais das variáveis que tiveram valores iniciais definidos pelo usuário.
- stack (RAM)**: Parte da RAM usada para variáveis temporárias e passagem de parâmetros
- heap (RAM)**: Parte da RAM usada para alocações dinâmicas

Stack (Pilha)

- ▶ Geralmente usado para alocação de variáveis temporárias ou passagem de parâmetros.
- ▶ É uma região linear de memória, gerida por um registro denominado de *Stack Pointer*.
- ▶ O compilador analisa o código e gerar instruções para uso do stack.

Stack (Pilha)

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

uint32_t global_sum = 10;

uint32_t sum(uint32_t *values,
             uint32_t size)
{
    uint32_t sum = 0;

    for(size_t n = 0; n < size; n++)
        sum += values[n];

    return sum;
}

int main(void)
{
    uint32_t data[] = { 1, 2, 3 };
    global_sum = sum(data, 3);
    printf("Sum is %u\r\n", global_sum);
    return 0;
}
```

Stack (Pilha)

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

uint32_t global_sum = 10;

uint32_t sum(uint32_t *values,
             uint32_t size)
{
    uint32_t sum = 0;

    for(size_t n = 0; n < size; n++)
        sum += values[n];

    return sum;
}

int main(void)
{
    uint32_t data[] = { 1, 2, 3 };
    global_sum = sum(data, 3);
    printf("Sum is %u\r\n", global_sum);
    return 0;
}
```

- ▶ `global_sum`: na seção data
- ▶ `data`: temporariamente no stack
- ▶ `sum`: temporariamente no stack
- ▶ variáveis passadas na chamada da função `sum`: temporariamente no stack
- ▶ retorno do valor da função
`sum`: temporariamente no stack

Cuidados no uso da pilha

- ▶ Tamanho de variáveis e estouro de pilha
- ▶ Recursão ou longo encadeamento de chamadas
- ▶ Retorno de valores locais
- ▶ Dimensionamento da pilha

Especificadores de classe de armazenamento

Uso de static em variáveis e funções:

- ▶ static em variáveis locais:
 - ▶ Escopo local (não visível fora da função)
 - ▶ Armazenamento em área global (não usa pilha)
 - ▶ Permite valores de inicialização.
- ▶ static em variáveis globais: linkagem interna para a variável (não “visível” por outros arquivos). Isso permite, por exemplo, ter *nomes de variáveis iguais em arquivos diferentes*.
- ▶ static em funções: linkagem interna para a função, similar a variáveis.

Stack (Pilha)

```
#include <stdint.h>
#include <stdbool.h>

#define DATA_SIZE 128

int drv_init(void)
{
    static bool initialized = false;
    static uint32_t data[DATA_SIZE];

    if(!initialized)
    {
        for(size_t n = 0 ; n < DATA_SIZE,
            n++)
            data[n] = n;

        initialized = true;
    }
}
```

Stack (Pilha)

```
#include <stdint.h>
#include <stdbool.h>

#define DATA_SIZE 128

int drv_init(void)
{
    static bool initialized = false;
    static uint32_t data[DATA_SIZE];

    if(!initialized)
    {
        for(size_t n = 0 ; n < DATA_SIZE,
            n++)
            data[n] = n;

        initialized = true;
    }
}
```

- ▶ `initialized`: na seção de variáveis globais mas sem visibilidade fora de `drv_init`, com valor inicial `false` e mantendo o valor entre chamadas da função
- ▶ `data`: idem, não penalizando o stack

Retorno de valores locais, da pilha, pode ?

```
typedef struct accel_s {
    float x;
    float y;
    float z;
} accel_t;

uint32_t rand_perc(void)
{
    uint32_t val = rand() % 100;
    return val;
}

accel_t accel_axis_get1(void)
{
    accel_t accel = { 0 };
    accel_read(&accel_t);
    return accel;
}

accel_t accel_axis_get2(void)
{
    accel_t *accel = calloc(1, sizeof(accel_t));
    accel_read(accel_t);
    return *accel;
}
```

Retorno de valores locais, da pilha, pode ?

```
typedef struct accel_s {  
    float x;  
    float y;  
    float z;  
} accel_t;  
  
uint32_t rand_perc(void)  
{  
    uint32_t val = rand() % 100;  
    return val;  
}  
  
accel_t accel_axis_get1(void)  
{  
    accel_t accel = { 0 };  
    accel_read(&accel_t);  
    return accel;  
}  
  
accel_t accel_axis_get2(void)  
{  
    accel_t *accel = calloc(1, sizeof(accel_t));  
    accel_read(accel_t);  
    return *accel;  
}
```



Retorno de valores locais, da pilha, pode ?

```
typedef struct accel_s {  
    float x;  
    float y;  
    float z;  
} accel_t;  
  
accel_t *accel_axis_get3(void)  
{  
    accel_t *accel = calloc(1, sizeof(accel_t));  
    accel_read(accel_t);  
    return accel;  
}  
  
accel_t *accel_axis_get4(void)  
{  
    /* auto */ accel_t accel = { 0 };  
    accel_read(&accel_t);  
    return &accel;  
}  
  
accel_t *accel_axis_get5(void)  
{  
    static accel_t accel = { 0 };  
    accel_read(&accel_t);  
    return &accel;  
}
```

Retorno de valores locais, da pilha, pode ?

```
typedef struct accel_s {  
    float x;  
    float y;  
    float z;  
} accel_t;  
  
accel_t *accel_axis_get3(void)  
{  
    accel_t *accel = calloc(1, sizeof(accel_t));  
    accel_read(accel_t);  
    return accel;  
}  
  
accel_t *accel_axis_get4(void)  
{  
    /* auto */ accel_t accel = { 0 };  
    accel_read(&accel_t);  
    return &accel;  
}  
  
accel_t *accel_axis_get5(void)  
{  
    static accel_t accel = { 0 };  
    accel_read(&accel_t);  
    return &accel;  
}
```



Passagem por valor x passagem por referência

- ▶ Passagem por valor: o valor original é copiado e não pode ser alterado pela função
- ▶ Passagem por referência:
 - ▶ O valor original não é copiado e pode ser alterado pela função. É utilizado um ponteiro para essa operação.
 - ▶ Também é interessante quando se passam estruturas muito grandes, economizando memória e processamento
- ▶ Mas antes é preciso falar um pouco sobre ponteiros !

Ponteiros

- ▶ Um ponteiro armazena endereço de memória e não um valor !
- ▶ São referências indiretas para outras variáveis ou funções
- ▶ São declarados com o emprego do asterisco "*"
- ▶ Endereços de variáveis podem ser obtidos com o uso do operador "&"
- ▶ Declaração básica de variável ponteiro:

```
<tipo_de_dado> *variavel;  
uint8_t *pbuffer;  
struct accel_s *paccel;
```

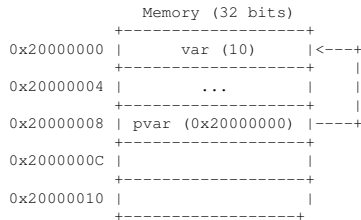
Ponteiros para variáveis simples

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

void add(uint32_t *pv)
{
    *pv = *pv + 1;
}

int main(void)
{
    uint32_t var = 10;
    uint32_t *pvar = &var;

    add(pvar);
    printf("%u\r\n", var);
    add(&var);
    printf("%u\r\n", var);
}
```



Ponteiros para estruturas simples

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
typedef struct frame_s
{
    uint8_t size;
    uint8_t cmd;
    uint8_t payload[32];
} frame_t;
void frame_print(frame_t *cmd)
{
    printf("C:%u S:%u P:%s\r\n",
        cmd->cmd, cmd->size, (char *) cmd->payload);
}
int main(void)
{
    frame_t frame = { 0 };
    frame.cmd = 1;
    frame.size = 6;
    strcpy((char *) frame.payload, "teste");
    frame_print(&frame);
    return 0;
}
```

Ponteiros e vetores

Ponteiros para função

Formas de dimensionamento da pilha

- ▶ Análises estáticas
 - ▶ Usando o próprio compilador (`-fstack-usage` e `-fcallgraph-info` no GCC) e analisando o uso da pilha a partir do `main()` (análise estática)
 - ▶ Usando ferramentas (PC-Lint, valgrind, cppcheck)
 - ▶ Problemas: podem falhar em casos de recursão, interrupções, chamadas indiretas (ponteiros para função)
- ▶ Análises dinâmicas: técnica da marca d'agua na região do stack (verificação dinâmica, útil em caso de recursão, interrupção)

Especificadores de classe de armazenamento (auto)

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

int main(void)
{
    /* auto */ uint32_t val = 0;
    printf("Val is %u\r\n", val);

    { // creating a new scope
        /* auto */ uint32_t val = 10;
        printf("Val is %u\r\n", val);
    }

    val++;
    printf("Val is %u\r\n", val);

    return 0;
}
```

auto

- ▶ É a classe padrão, de escopo local, armazenadas na RAM (pilha), valor inicial pode ser lixo.
- ▶ Pode ser omitida, por simplicidade.

Especificadores de classe de armazenamento

`static`

- ▶ Se usada com variáveis dentro de funções: gera variáveis que mantem valores entre chamadas e são alocadas na área de variáveis globais. São inicializadas com zero, caso não explicitamente especificado.
- ▶ Se usada com variáveis no escopo do arquivo (fora das funções) o comportamento é o mesmo mas existem diferenças em relação a sua visibilidade.
- ▶ `static` também pode ser usada em declarações de função. Novamente, implica em mudança de visibilidade.
- ▶ Novo conceito requerido: **linkagem padrão da linguagem C** !

Especificadores de classe de armazenamento (extern)

```
// file1.c
uint32_t var = 10;
uint32_t var_get(void)
{
    return var;
}
```

```
// file2.c
#include <stdint.h>
#include <stdio.h>

// not recommended!
extern uint32_t var;
extern uint32_t var_get(void);

int main(void)
{
    printf("Val is %u\r\n", var);
    printf("Val is %u\r\n", var_get());

    return 0;
}
```

- ▶ Por default, o compilador C externa todos os símbolos gerados (main,var).
- ▶ Com `extern`, você pode explicitamente dizer que existe algo externo ao seu arquivo (file2.c) e que pretende usar.
- ▶ No entanto, mesmo que não especifique nada, o linker vai tentar encontrar algo na tabela de símbolos que resolva a referência.

Especificadores de classe de armazenamento (static/extern)

```
// file1.h
uint32_t var_get(void);
```

```
// file1.c
static uint32_t var = 10;
uint32_t var_get(void)
{
    return var;
}
```

```
// file2.c
#include <stdint.h>
#include <stdio.h>
#include "file1.h"

int main(void)
{
    printf("Val is %u\r\n", var_get());

    return 0;
}
```

- ▶ Agora `var` tem **linkagem interna**, não é mais vista fora de `file2.c` (o símbolo não é exportado)
- ▶ Conceito também válido para funções `static` !
- ▶ Princípio de encapsulamento ou API.

Especificadores de classe de armazenamento

`register`

- ▶ Se usada com variáveis dentro de funções.
- ▶ Indica ao compilador que deseja o armazenamento da variável em um registro do processador (geralmente para maior performance).
- ▶ Pouco útil atualmente, em geral o compilador resolve bem essas situações.

Arquivos de código fonte

```
// inclusoes da linguagem C
// inclusoes do projeto
// defines internos
// tipos de dados internas
// (estruturas, unioes, typedefs)
// constante internas (const)
// variaveis internas (static)
// prototipos de funcoes internas
// funcoes internas (static)
// ou exportadas (ver .h)
```

- ▶ Externe via funções o acesso a seus dados internos (encapsulamento)
- ▶ Dê escopo de arquivo para as suas funções e variáveis com `static` !
- ▶ Salve RAM colocando como `const` o que for realmente constante.

Arquivos de código fonte

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#include "demo.h"

static bool started = false;

static void accel_init_drv(void)
{
}

void accel_read(accel_data_t *data)
{
}

void accel_init(void)
{
    if(!started)
    {
        accel_init_drv();
        started = true;
    }
}
```



```
// test
#include <stdio.h>
int main(void)
{
    printf("Hello World!");
}
```

Palavras Reservadas

► x

