Linguagem C

Uma revisão com foco em sistemas embarcados

Marcelo Barros e Daniel Carvalho

UFU/FEELT

27 de setembro de 2022

Outline

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" ?

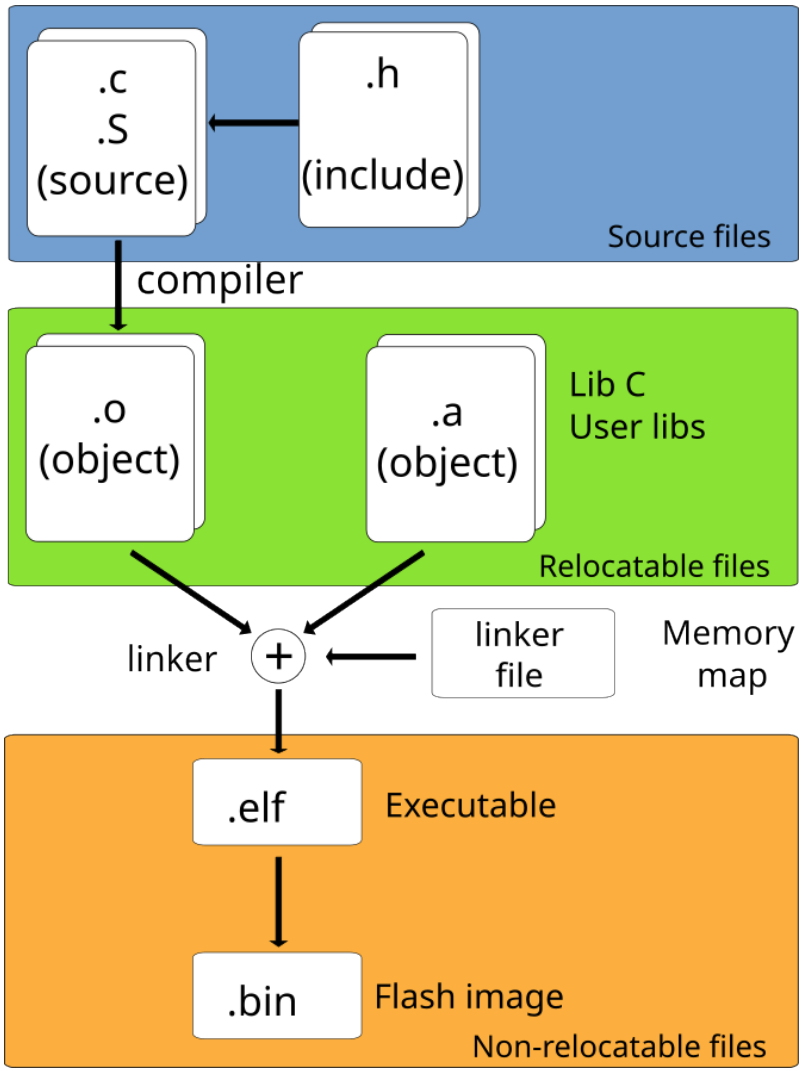
- Norma: *Programming languages — C — Extensions to support embedded processors* ISO/IEC TR 18037:2008
- Tecnicamente é apenas uma extensão do C, cobrindo:
  - Aritmética de ponto fixo
  - Espaço de endereçamento
  - Endereçamento de hardware básico para I/O

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

```c
#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 linker[2].
- 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>):

```
0000000000001149 <_Z9accel_readRfS_S_>:
0000000000001160 <_Z9accel_readR7accel_s>:
```

[2] https://stackoverflow.com/questions/41524956/gcc-c-name-mangling-reference

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 compilador, ele é divido 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)

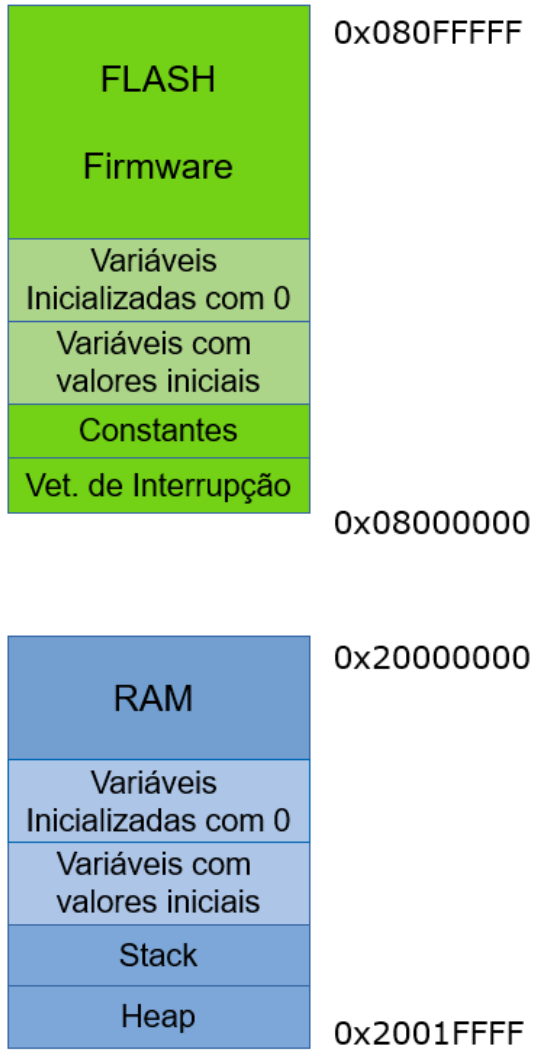
data (RAM): Locas das variáveis que tiveram valores iniciais definidos pelo usuário.

stack (RAM): Parte da RAM usada para variável temporárias e passagem de parâmetros

heap (RAM): Parte da RAM usada para alocações dinâmicas

Exemplo de seções para um microcontrolador STM32F411

- Um arquivo de *linkedição* (.ld) especifica as áreas de memória (flash e RAM) e como elas serão divididas para comportar as seções de text, bss, data, rodata, heap e stack.
- Um código de inicialização (que roda antes do `main()`) é responsável por preparar o terreno:
  - Zerar a seção bss
  - Inicializar a seção data
  - Inicializar o ponteiro da pilha
  - Inicializar o heap



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)

```c
#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)

```c
#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)

```c
#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)

```c
#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 valore 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 ?

```c
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 ?

```c
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, ou seja, deve ser passada o endereço do dado (referência) e não o seu valor.
  - Também é interessante quando se passam estruturas muito grandes, economizando memória e processamento

```
// passagem por valor
void func1(uint32_t v){ v = v + 5; }
// passagem por referencia
void func2(uint32_t *v){ *v = *v + 5; }

int main(void)
{
  uint32_t v = 10;
  func1(v); // v nao sera alterado
  func2(&v); // v sera alterado (15)
  return 0;
}
```

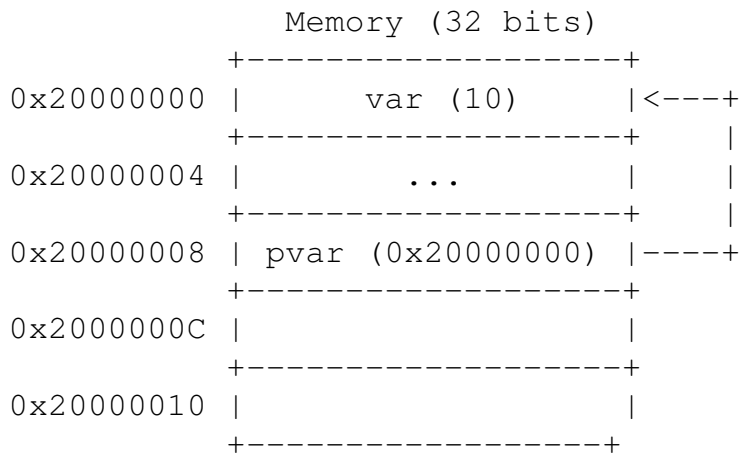
Ponteiros

- 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

```c
#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

```c
#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

```c
#include <stdio.h>
#include  <inttypes.h>

uint32_t v[4];

int main(void)
{
    printf("V     = 0x%08lX\n", (uintptr_t)v);
    printf("&V    = 0x%08lX\n", (uintptr_t)&v);
    printf("&V[0] = 0x%08lX\n", (uintptr_t)&v[0]);
    printf("&V[1] = 0x%08lX\n", (uintptr_t)&v[1]);
    printf("&V[2] = 0x%08lX\n", (uintptr_t)(v + 2));
    printf("&V[2] = 0x%08lX\n", (uintptr_t)(v + 3));
    return 0;
}
```

```
V     = 0x55D2EBB8F020
&V    = 0x55D2EBB8F020
&V[0] = 0x55D2EBB8F020
&V[1] = 0x55D2EBB8F024
&V[2] = 0x55D2EBB8F028
&V[2] = 0x55D2EBB8F02C
```

Ponteiros e arrays de caracteres

```c
#include <stdio.h>
#include <inttypes.h>
#include <string.h>
int main (void)
{
  uint8_t str1[] = { 'a', 'b', '\0' };
  uint8_t *str2 = "ab";
  uint8_t *str3[] = { "ab", "12" };
  printf("STR1 = 0x%08lX\n", (uintptr_t) str1);
  printf("STR2 = 0x%08lX\n", (uintptr_t) str2);
  printf("STR3 = 0x%08lX\n", (uintptr_t) str3);
  printf("STR1    %s,%lu\n",str1,strlen(str1));
  printf("STR2    %s,%lu\n",str2,strlen(str2));
  printf("STR3[0] %s,%lu\n",str3[0],strlen(str3[0]));
  printf("STR3[1] %s,%lu\n",str3[1],strlen(str3[1]));
  return 0;
}
```

```
STR1 = 0x7FFE326EC8E5   <== stack
STR2 = 0x5585446ED004   <== global
STR3 = 0x7FFE326EC8D0   <== stack
STR1    ab,2
STR2    ab,2
STR3[0] ab,2
STR3[1] 12,2
```

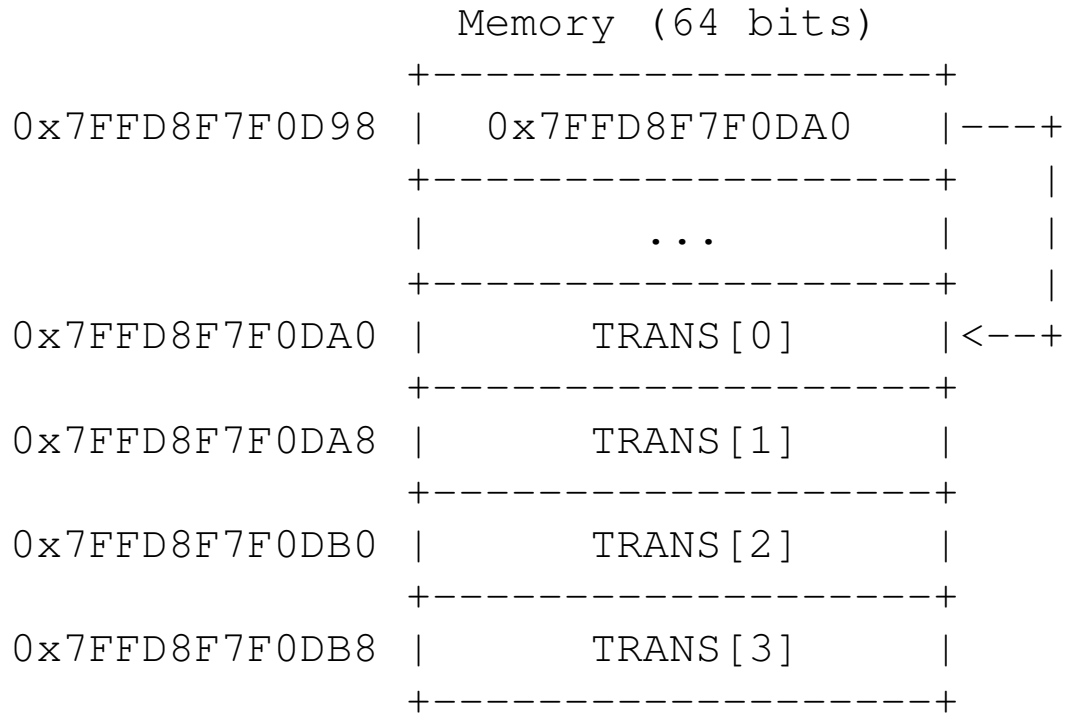
Aritmética de ponteiros

Regra básica: ao incrementar/decrementar um ponteiro, o valor adicionado/subtraído é igual ao tamanho do tipo de dado para o qual ele aponta.

```
#include <stdio.h>
#include  <inttypes.h>
typedef struct trans_s
{
  uint32_t ID;
  float value;
} trans_t;
int main(void)
{
  trans_t trans[4] = { 0 } ;
  trans_t *ptrans = trans;
  printf("SIZE    = %lu\n", sizeof(trans_t));
  printf("TRANS   = 0x%08lX\n", (uintptr_t)trans);
  printf("PTRANS  = 0x%08lX\n", (uintptr_t)ptrans);
  printf("&PTRANS = 0x%08lX\n", (uintptr_t)&ptrans);
  printf("TRAN[0] = 0x%08lX\n", (uintptr_t)&trans[0]);
  printf("TRAN[1] = 0x%08lX\n", (uintptr_t)(ptrans + 1));
  return 0;
}
```

Aritmética de ponteiros

```
SIZE    = 8
TRANS   = 0x7FFD8F7F0DA0
PTRANS  = 0x7FFD8F7F0DA0
&PTRANS = 0x7FFD8F7F0D98
TRAN[0] = 0x7FFD8F7F0DA0
TRAN[1] = 0x7FFD8F7F0DA8
```



Ponteiros para função

- O nome da função é um sempre um ponteiro !
- O grande problema é a notação, que é confusa !

```c
#include <stdio.h>
#include <inttypes.h>
int64_t sum(int32_t a, int32_t b)
{
    return a + b;
}
int main (void)
{
  printf("%lu\n",sum(10,20)); // 30
  int64_t (*pfun)(int32_t a, int32_t b) = sum;
  printf("%lu\n",pfun(30,40)); // 70
  uintptr_t pf = (uintptr_t) sum;
  int64_t r = ((int64_t (*)(int32_t a, int32_t b))pf)(50,60);
  printf("%lu\n",r); // 100
  return 0;
}
```

Ponteiros para função

- Mas não precisa ser assim !
- Acompanhe como criar um ponteiro pra função e como o programa se transforma depois disso.

```
int64_t sum(int32_t a, int32_t b);
typedef int64_t sum(int32_t a, int32_t b);
typedef int64_t (sum)(int32_t a, int32_t b);
typedef int64_t (sum_t)(int32_t a, int32_t b);
typedef int64_t (*sum_t)(int32_t a, int32_t b);
```

```
typedef <retorno> (*nome_do_tipo)(lista,de,parâmetros);
```

Ponteiros para função

- Bem mais legível:

```c
#include <stdio.h>
#include <inttypes.h>

typedef int64_t (*sum_t)(int32_t a, int32_t b);

int64_t sum(int32_t a, int32_t b)
{
    return a + b;
}
int main (void)
{
  printf("%lu\n",sum(10,20)); // 30
  sum_t pfun = sum;
  printf("%lu\n",pfun(30,40)); // 70
  uintptr_t pf = (uintptr_t) sum;
  int64_t r = ((sum_t)pf)(50,60);
  printf("%lu\n",r); // 100
  return 0;
}
```

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)

```c
#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)

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

```c
// 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)

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

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

```c
// 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.

Qualificadores

`const volatile`

const: indica que a variável é constante, ou seja, que seu valor não muda durante a execução. O compilador pode usar essa informação para colocar essa variável em flash e encontrar erros em tempo de compilação.

volatile: indica que o valor da variável pode ser alterado por outros elementos além do fluxo de programa principal em execução. Exemplos:

- Uma variável global compartilhada entre duas tarefas ou entre o programa principal e uma interrupção associadas a uma opção de compilação com maior nível de otimização.
- Uma variável que aponta para um registro do processador, ou seja, que pode ter o valor modificado pelo próprio hardware.

Qualificadores e ponteiros

Usar adequadamente o qualificador `const` com ponteiros é, frequentemente, uma boa prática:

```
void lcd_write(const uint8_t *data)
{
  // o dado apontado deve ser constante,
  // a linha abaixo gera um erro de compilacao
  data[0] = 1; // error: assignment of read-only location
}

void lcd_write(uint8_t *const data)
{
  // o ponteiro dever ser constante,
  // a linha abaixo gera um erro de compilacao
  data++; // error: increment of read-only parameter
}

void lcd_write(const uint8_t *const data)
{
  // o ponteiro dever ser constante,
  // assim como o dado apontado por ele !
}
```

Arquivos de código fonte (modelo)

```c
// 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 (modelo)

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

#include "demo.h"

static bool started = false;

static void accel_drv_init(void)
{
}

void accel_read(accel_data_t *data)
{
}

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

Estruturas

- Ajudam a organizar os dados, evitando variáveis espalhadas.
- Melhoram a visualização e entendimento do código (legibilidade e manutenção).
- Reduzem complexidade, se bem usadas.
- `typedef` pode ajudar !

```
struct coord_s
{
  uint32_t x;
  uint32_t y;
};

struct rect_s
{
  struct coord_s c1;
  struct coord_s c2;
};

struct rect_s rect = { 0 };
void func(struct rect_s *rect);
```

```
typedef struct coord_s
{
  uint32_t x;
  uint32_t y;
} coord_t;

typedef struct rect_s
{
  coord_t c1;
  coord_t c2;
} rect_t;

rect_t rect = { 0 };
void func(rect_t *rect);
```

Enumerações

- Permitem organizar definições relacionadas.
- Pode ajudar na detecção de erros (valores inválidos).
- Reduzem complexidade, melhoram a legibilidade.
- Obedecem regras de escopo, algo que não é possível com defines.

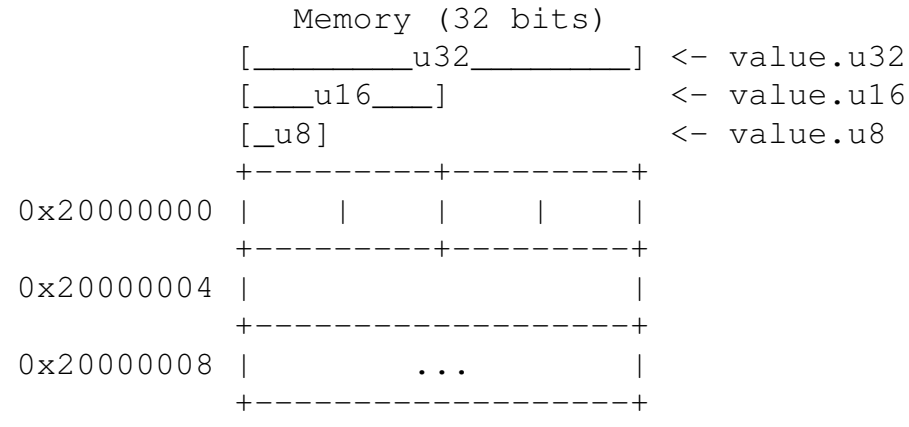
```c
#define RTC_WEEKDAY_SUN 0
#define RTC_WEEKDAY_MON 1
#define RTC_WEEKDAY_TUE 2
#define RTC_WEEKDAY_WED 3
#define RTC_WEEKDAY_THU 4
#define RTC_WEEKDAY_FRI 5
#define RTC_WEEKDAY_SAT 6

void rtc_wday_set(uint8_t wday)
{
}
```

```c
typedef enum rtc_wday_e
{
  RTC_WEEKDAY_SUN = 0,
  RTC_WEEKDAY_MON = 1,
  RTC_WEEKDAY_TUE = 2,
  RTC_WEEKDAY_WED = 3,
  RTC_WEEKDAY_THU = 4,
  RTC_WEEKDAY_FRI = 5,
  RTC_WEEKDAY_SAT = 6,
} rtc_wday_t;

void rtc_wday_set(rtc_wday_t wday)
{
}
```

Uniões

- Enquanto as estruturas dispõem sequencialmente os seus membros na memória, as uniões os armazenam na mesma posição.
- Isso permite "visualizar" a mesma área de memória com diferentes lentes.
- Em geral, muito útil em generalizações, evitando duplicações de memória.

```c
typedef union kved_value_u
{
  uint8_t u8;
  uint16_t u16;
  uint32_t u32;
} kved_value_t;

typedef struct kved_data_s
{
  kved_value_t value;
  bool updated;
} kved_data_t;
```



Exemplo completo (1/2)

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

typedef enum obj_type_e
{
    OBJ_TYPE_RECT = 0,
    OBJ_TYPE_CIRCLE,
} obj_type_t;

typedef struct coord_s
{
  uint32_t x;
  uint32_t y;
} coord_t;

typedef struct rect_s
{
    coord_t c1;
    coord_t c2;
} rect_t;
```

```c
typedef struct circle_s
{
    coord_t c;
    uint32_t radius;
} circle_t;

typedef union objs_u
{
    circle_t circle;
    rect_t rect;
} objs_t;

typedef struct obj_s
{
    obj_type_t type;
    objs_t elem;
} obj_t;
```

Exemplo completo (2/2)

```c
void obj_print(obj_t *obj)
{
  if(obj->type == OBJ_TYPE_RECT)
  {
    printf("RECT: %u,%u,%u,%u\n",
      obj->elem.rect.c1.x,
      obj->elem.rect.c1.y,
      obj->elem.rect.c2.x,
      obj->elem.rect.c2.y);
  }
  else if(obj->type == OBJ_TYPE_CIRCLE)
  {
    printf("CIRCLE: %u,%u,%u\n",
      obj->elem.circle.c.x,
      obj->elem.circle.c.y,
      obj->elem.circle.radius);
  }
}
```

```c
int main(void)
{
  obj_t obj1 =
    { .type = OBJ_TYPE_RECT,
      .elem.rect.c1 = { 0, 0},
      .elem.rect.c2 = {10,10}};

  obj_t obj2 =
    { .type = OBJ_TYPE_CIRCLE,
      .elem.circle.c = { 0,0 },
      .elem.circle.radius = 10};

  obj_print(&obj1);
  obj_print(&obj2);

  return 0;
}
```