# Linguagem C

# Linguagem C

- Representação numérica;

- Tipos de dados:

  => variáveis

  => constantes

# Linguagem C

## HCS08QE128

| TIPO | FAIXA DE VALORES | TAMANHO EM BYTES |
|---|---|---|
| signed char | -128 a 127 | 1 |
| char | 0 a 255 | 1 |
| int | -32.768 a 32767 | 2 |
| unsigned int | 0 a 65.535 | 2 |
| unsigned long int | 0 a 4.294.967.295 | 4 |
| long int | -2147483647 a 2147483647 | 4 |
| float | 1.17549e-38 a 3.40282e38 | 4 |
| double | 2.23e-308 a 1.7e308 | 8 |
| long double | 2.23e-308 a 1.7e308 | 8 |

# Linguagem C

Quantos bits tem um int em C?

# Linguagem C

Quantos bits tem um int em C?

Depende DO COMPILADOR.

O compilador vai decidir isso. No nosso caso, o compilador do CodeWarrior fornece duas opções:

4-byte int

ou

2-byte int

# Linguagem C

Considerando para o QE128(HCS08) um int de 2 bytes...

**unsigned int** – temos 16 bits para representar um número **não-sinalizado**. Ou seja, seja qual for o padrão de bits na variável, o compilador o interpretará como um número sem sinal. Podemos representar valores de 0 até 2^16 -1.

**int/signed int** – temos 16 bits para representar um número **COM sinal**. Podemos representar valores de -32768 até 32767 (lembrando que utilizamos a representação em complemento de 2).

**DICA!** Ao representar **endereços** no HCS08 devemos utilizar o tipo **unsigned int** (ou **void ***), já que os endereços são de 16 bits e não faz sentido falar em endereço absoluto negativo.

# Linguagem C

**unsigned short** – idêntico ao unsigned int, porém com metade do número de bits.

**short/signed short** – indêntico ao int (signed int), porém com metade do número de bits.

**unsigned char** – temos 8 bits, em representação não-sinalizada.

**char/signed char** – 8 bits, representação sinalizada.

# Tipos construídos

**Typedef**

**C permite que sejam definidos** explicitamente **novos tipos de dados** usando a palavra reservada **typedef**.

**typedef** não cria realmente uma nova classe de dados, mas sim **define um novo nome para uma tipo já existente**.

# Tipos construídos

**Typedef**

**O uso do typedef torna os programas** em C mais legíveis e mais portáteis, pois bastará alterar a definição do **typedef** quando trocar de ambiente.

Ex.: typedef  unsigned char byte*;*

# Variáveis

- o nome deve começar com uma letra ou sublinhado (_)

- os caracteres subsequentes devem ser letras, números ou sublinhado (_).

- o nome de uma variável não pode ser igual a uma palavra reservada, nem igual ao nome de uma função declarada pelo programador, ou pelas bibliotecas do C.

- é "case sensitive“, portanto deve-se prestar atenção às maiúsculas e minúsculas.

# Dica de nomes de variáveis

- É uma prática tradicional do C, usar **letras minúsculas para nomes de variáveis** e **maiúsculas para nomes de constantes.**

- *Isto facilita na hora da leitura do código*;

## Declaração e Inicialização de Variáveis

As variáveis no C ***devem*** ser declaradas antes de serem usadas.

A forma geral da declaração de variáveis é:
*tipo_da_variável  lista_de_variáveis;*

## Declaração e Inicialização de Variáveis

- As variáveis de mesmo tipo e deverão ser separadas por vírgula.

- Como o tipo padrão do C é o **int**, quando vamos declarar variáveis **int** com algum dos modificadores de tipo, basta colocar o nome do modificador de tipo.

## Declaração e Inicialização de Variáveis

- Assim um **long** basta para declarar um **long int**.

- Exemplo:

```
char ch, letra;  // duas variáveis do tipo char
long count;      // uma variavel long int
float pi;        // uma variável float
```

# Declaração e Inicialização de Variáveis

Podemos inicializar variáveis no momento de sua declaração. Para fazer isto podemos usar a forma geral:

*tipo_da_variável nome_da_variável = constante;*

- Isto é importante pois quando o C cria uma variável ele *não* a inicializa.

- Isto significa que até que um primeiro valor seja atribuído à nova variável ela tem um valor *indefinido* e que não pode ser utilizado para nada.

# Declaração e Inicialização de Variáveis

Exemplos de inicialização são dados abaixo :

```
char ch='D';

int count=0;

float pi=3.141;
```

- *Nunca* presuma que uma variável declarada vale zero ou qualquer outro valor.

# Declaração e Inicialização de Variáveis

Verifique se o programa apresenta um erro:

```
int main()
{
    int i;
    int j;
    j = 10;
    int k = 20;
    return(0);
}
```

# Declaração e Inicialização de Variáveis

Verifique se o programa apresenta um erro:

```
int main()
{
    int i;
    int j;
    j = 10;
    int k = 20; /* Esta declaração de variável não é
    válida, pois não está sendo feita no início do bloco */
    return(0);
}
```

## Lugares para declarar as variáveis

1) O primeiro é **fora de todas as funções** do programa.

   Estas variáveis são chamadas **variáveis globais** e podem ser usadas a partir de qualquer lugar no programa. Todas as funções têm acesso a elas.

## Lugares para declarar as variáveis

2) *Início* de uma *função* ou *bloco de código*.

São chamadas de **variáveis locais** e só têm validade dentro do bloco ou função no qual são declaradas, isto é, só o bloco ou função à qual ela pertence sabe da existência desta variável.

# Lugares para declarar as variáveis

3) As chamadas **variáveis formais** de uma função são declaradas na **lista de parâmetros** de uma função. Apesar destas variáveis receberem valores externos, estas variáveis são conhecidas apenas pela função onde são declaradas (portanto, são também **variáveis locais**).

# Linguagem C (variáveis)

**Exemplo: Verificar se tem algum erro de código!!!**

```
1.  #include <stdio.h>
2.  int contador;    // variável GLOBAL
3.  int func1(int j);      //Protótipo de uma função

1.  int main()      {
2.  char condicao;      // variáveis lOCAIS da função
3.  int i;
4.  for (i=0; i<100; i=i+1)
5.   {          /* Bloco do for */
6.          float f2;     // variável LOCAL do bloco  de código
7.          /* etc ...                  */

1.          func1(i);
2.    }             /* etc  ...  */
•     return(0);
•   f2++;
1.   }
2.  int func1(int j) {    // variável FORMAL
3.   /* aqui viria o código da funcao  ...
•   */              }
```

# Linguagem C (variáveis)

**Exemplo: Verificar se tem algum erro de código!!!**

```
int contador;     // variável GLOBAL

int func1(int j) {     // variável FORMAL
 /* aqui viria o código da funcao                ...
*/                }

int main()        {
char condicao;       // variáveis lOCAIS da função
int i;
for (i=0; i<100; i=i+1)
 {            /* Bloco do for */
          float f2;      // variável LOCAL do bloco  de código
          /* etc ...                */

          func1(i);
 }                /* etc  ...  */
  return(0);
f2++;       //ERRO de COMPILAÇÃO, pois está fora do bloco!!
 }
```

# Declaração e Inicialização de Variáveis

Quando utilizar variável global e variável local ?

Qual escolher ?

# Declaração e Inicialização de Variáveis

Via de regra, utiliza-se **variáveis locais**.

Se alguma outra função precisar do conteúdo dessa variável passa-se como parâmetro.

Mas, por quê?

# Declaração e Inicialização de Variáveis

**VARIÁVEIS GLOBAIS** POSSUEM ESPAÇO DE MEMÓRIA RESERVADO **EXCLUSIVAMENTE.**

**VARIÁVEIS LOCAIS** UTILIZEM ESPAÇOS DE MEMÓRIAS **TEMPORÁRIOS**, COMO A **PILHA OU OS REGISTRADORES** DA CPU, OS QUAIS PODEM SER **REUTILIZADOS** POR OUTRAS VARIÁVEIS LOCAIS.

# Declaração e Inicialização de Variáveis

- Quando as **funções do seu programa são chamadas**, o conteúdo das variáveis (**locais**) vai sendo **empilhado na memória**, ou seja, na parte da pilha. Uma vez terminada a função, todos os dados referentes a ela na pilha são "**apagados**".

- Dados **globais** ficam na memória em uma posição exclusiva.

# Declaração e Inicialização de Variáveis

• Com isso, podemos entender o porquê da recomendação geral de utilização de variáveis locais. É tudo uma questão de **economia de memória**. Se todas as variáveis de nosso programa fossem globais, não haveria a possibilidade da liberação de memória.

• Há um outro motivo. Como qualquer função do programa pode utilizar uma variável global, é fácil que o programador se perca na sua utilização e que haja **erros na programação**.

# Declaração e Inicialização de Variáveis

Quando usar variáveis globais ?

# Declaração e Inicialização de Variáveis

- Em geral, quando o conteúdo de uma variável DEVE ser **acessado por várias funções** do programa.

- Isso acontece MUITO em programação de sistemas embarcados. A variável global pode representar o conteúdo de um registrador de estado, por exemplo. As funções do programa devem verificar o estado para decidir o que fazer.

# Declaração e Inicialização de Variáveis

• Veja um exemplo simples de dois programas que fazem a mesma coisa, porém, um deles utilizando variável global e o outro local.

• Nesse pequeno exemplo podemos ver que já eliminamos a necessidade de passagem por parâmetros da variável em questão.

• Num programa maior, no qual muitas funções chamam muitas outras funções, isso representa um ganho substancial em organização.

# Exemplo usando Variável Local

```
int main () {
        int registradorEstado;
        a(&registradorEstado);
        b(&registradorEstado);
}
void a (int * estado) {
        if (estado == 0) {
                /* faz algo */
                estado = 1;
        }
}
void b(int * estado) {
        if (estado == 1) {
                /*faz algo*/
                estado = 0;
        }
}
```

# Exemplo usando Variável Global

```
int registradorEstado;
int main () {
        a();
        b();
}
void a () {
        if (registradorEstado == 0) {
                /* faz algo */
                registradorEstado = 1;
        }
}
void b() {
        if (registradorEstado == 1) {
                /*faz algo*/
                registradorEstado = 0;
        }
}
```

## Tipos de dados avançados: Constantes

**Modificador de Acesso:** mudam a maneira com a qual a variável é acessada e modificada.

***Const***: Esta variável não pode ser modificada no programa. Como o nome já sugere é útil para se declarar constantes.

# Linguagem C (constantes)

**Modificador de Acesso**

Ex.:

```
const  float  PI = 3.141;  // pode ser
                           // inicializada.
```

Mas **PI** não pode ser alterado em qualquer outra parte do programa. Se o programador tentar modificar **PI** o compilador gerará um erro de compilação.