MC202 - Estruturas de Dados

Guilherme P. Telles

IC

29 de Agosto de 2019

Avisos

- Estes slides contêm erros.
- Estes slides são incompletos.
- Estes slides são escritos usando português anterior à reforma ortográfica de 2009.

Estruturas de dados

- Uma estrutura de dados é uma forma de organizar dados na memória para tornar a manipulação desses dados eficiente.

Estruturas de dados

- Uma estrutura de dados é uma forma de organizar dados na memória para tornar a manipulação desses dados eficiente.
  - organizar dados na memória: manipulação de memória.

Estruturas de dados

- Uma estrutura de dados é uma forma de organizar dados na memória para tornar a manipulação desses dados eficiente.
  - organizar dados na memória: manipulação de memória.
  - dados: a composição dos dados e a forma como mudam.

Estruturas de dados

- Uma estrutura de dados é uma forma de organizar dados na memória para tornar a manipulação desses dados eficiente.
  - organizar dados na memória: manipulação de memória.
  - dados: a composição dos dados e a forma como mudam.
  - manipulação: as operações realizadas sobre os dados e a freqüência delas.

Estruturas de dados

- Uma estrutura de dados é uma forma de organizar dados na memória para tornar a manipulação desses dados eficiente.
  - organizar dados na memória: manipulação de memória.
  - dados: a composição dos dados e a forma como mudam.
  - manipulação: as operações realizadas sobre os dados e a freqüência delas.
  - eficiência: tempo e memória.

Manipulação de memória

- Linguagem C.

Manipulação de memória

- Linguagem C.
- Permite um controle preciso da quantidade de memória que é usada e de como ela é organizada.

Manipulação de memória

- Linguagem C.
- Permite um controle preciso da quantidade de memória que é usada e de como ela é organizada.
  - Alocação dinâmica de memória e apontadores.

Manipulação de memória

- Linguagem C.
- Permite um controle preciso da quantidade de memória que é usada e de como ela é organizada.
  - Alocação dinâmica de memória e apontadores.
- C++ também é boa para isso.

Parte I

Conceitos de MC102 e novos

Programa etc.

- Um programa-fonte é um texto com instruções para um computador.
- Para executar um programa-fonte ele precisa ser traduzido em uma forma que o computador entenda.
  - Para algumas linguagens, o programa é traduzido linha-a-linha sempre que ele é executado (interpretação).
  - Para outras, o programa é traduzido inteiramente antes de ser executado (compilação).
- Uma sentença é uma seqüência de palavras, números e operadores que realiza uma ação.
- Sentenças são formadas por uma ou mais expressões.
- Uma expressão é ou uma variável ou uma chamada de função ou um valor ou um conjunto de variáveis, chamadas de funções e valores combinados por operadores.

Variável

- Uma ou mais posições de memória associadas a um nome e a um tipo e que tem um valor. O valor é a informação que está armazenada nessas posições de memória.
- O valor da variável pode ser modificado, mas o nome não pode.

Tipo

- Define o significado dos bits que compõem uma variável e quais operadores podem ser aplicados a ela.

Apontador

- Um apontador é uma variável cujo valor é um endereço da memória.
- Dizemos que uma variável que é um apontador "aponta" para uma variável ou para uma região alocada dinamicamente na memória.
- O tipo de uma variável que é um apontador é "apontador para [o tipo de dado cujo endereço é armazenado]".

Array

- Um array (ou vetor ou arranjo) é uma coleção indexada por um inteiro formada elementos homogêneos.

Função

- Um bloco delimitado de sentenças de uma linguagem de programação que tem um nome.
- Pode haver definição de variáveis dentro do bloco.
- Uma função recebe parâmetros. Dentro da função, um parâmetro se comporta como uma variável.
- O nome da função é usado para chamar a função.
- A chamada da função causa a execução dos comandos que a compõem.
- O ponto de entrada da função é único.
- Outros nomes são sub-rotina, sub-programa, procedimento.

Definições e declarações

- Uma definição de variáveis ou constantes associa o tipo ao nome de variável ou constante e causa a alocação de memória para a variável.
- Uma definição de função associa o tipo e ordem dos parâmetros ao nome de função e define o bloco de sentenças da função.

- Uma declaração associa o tipo ao nome de variável ou constante, mas não causa a alocação de memória.
- Uma declaração associa o tipo e ordem dos parâmetros ao nome de função, mas sem definir seu bloco de sentenças.

Parte II

C

Linguagem C

- Linguagem estruturada.
- Criada em 1972 por Dennis Ritchie (Bell Labs).
- Padronizada pelo ANSI pela primeira vez em 1988.
- Várias versões padronizadas, todas com compatibilidade pregressa.
- Há extensões não padronizadas.

- Em assembly (linguagem de baixo nível), os detalhes da máquina que está sendo programada estão implícitos todo o tempo (número de registradores, instruções de máquina, etc). O modelo fornecido pela linguagem é muito distante das aplicações práticas.
- Em uma linguagem de alto nivel, p.ex. Java, os detalhes da máquina estão escondidos. Há um custo de memória e tempo de processamento nesse modelo.
- C oferece um modelo está próximo da máquina e permite construir código eficiente, sem estar atrelado a uma máquina em particular.
- Portável.
- Robusta e estável.

- C é bastante adequada para o desenvolvimento de software básico e para manipular grandes volumes de dados.
- Assume que o programador está sempre certo: permite uma gama ampla de construções, inclusive as estranhas. Essa liberdade também permite cometer erros espetaculares.

Conteúdo

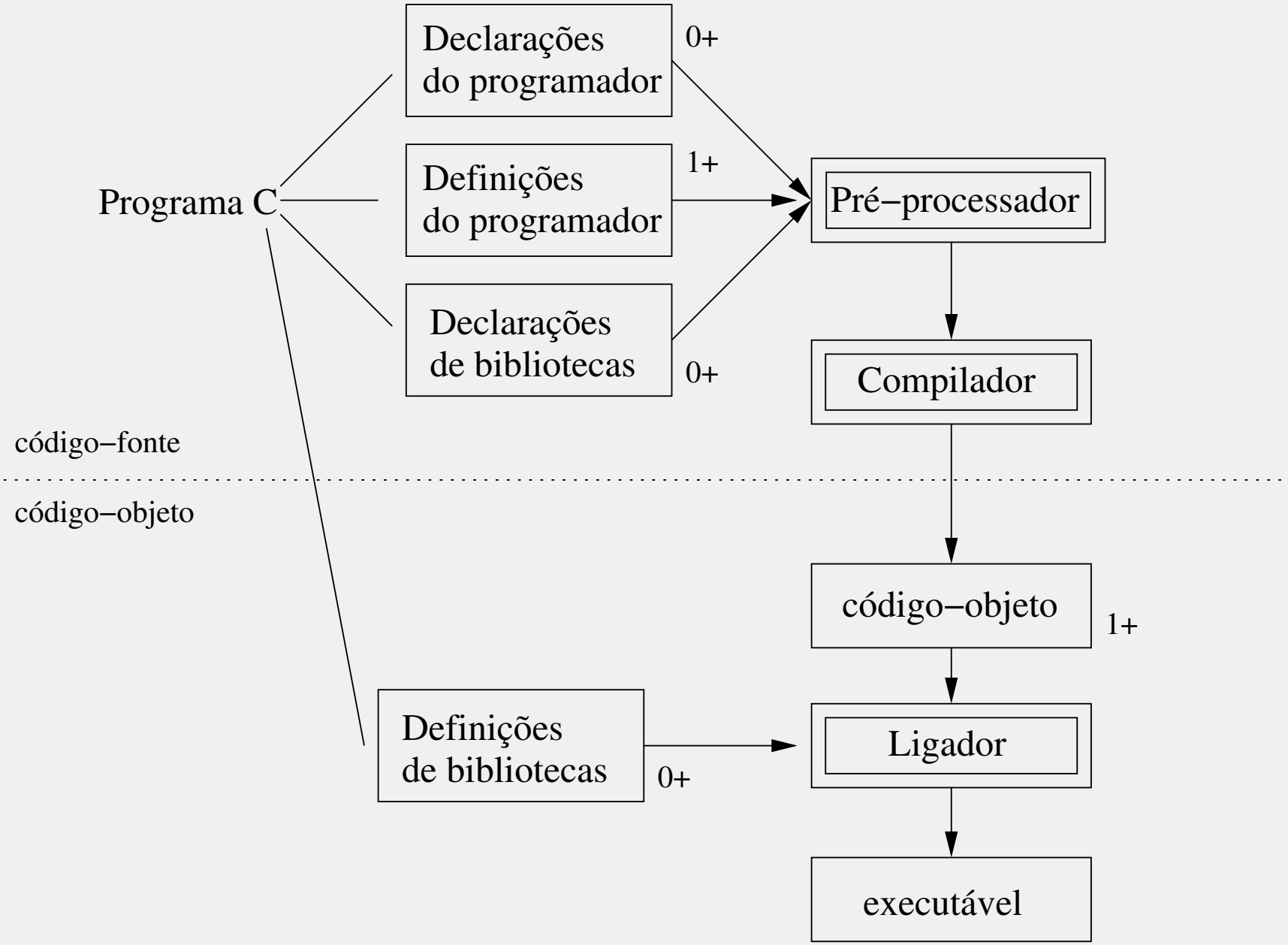
Material

- Quase qualquer livro, apostila ou tutorial sobre C vai servir.
  - Kernigham e Ritchie (1988): referência tradicional, bom mas antigo.
  - A book on C (A. Kelley e I. Pohl, 1998), muito bom, um pouco antigo, tem muitos exercícios.
  - Programming in C (S. Kochan, 2014), bem recomendado.
  - C: how to program (P. Deitel e H. Deitel, 2016), bem recomendado.
- Com críticas negativas:
  C The Complete Reference (H. Schildt), Let Us C (Y. Kanetkar), Learn C The Hard Way (Z. Shaw).

Programa C

- Um programa C consiste de um ou mais arquivos.
- Cada arquivo tem zero ou mais definições ou declarações de tipos, variáveis e funções, sendo que uma delas é a função `main()`.

Compilação


Declarações
do programador
0+
Definições
do programador
1+
Declarações
de bibliotecas
0+
Programa C
Pré−processador
Compilador
código−fonte
código−objeto
código−objeto
1+
Definições
de bibliotecas
0+
Ligador
executável

Pré-processador

- Reconhece linhas que começam com # (diretivas).
- Faz a junção dos arquivos com código-fonte que compõem um programa C.
- Faz substituição de constantes simbólicas e macros.
- Permite compilação condicional de trechos do programa.

Compilador

- Gera código-objeto a partir do código-fonte.
- O código gerado pelo compilador pode ter referência a variáveis e funções definidos em outros arquivos contendo código-objeto.

Ligador

- Resolve as referências entre códigos-objeto e gera um único programa executável.
- Os arquivos contendo código-objeto podem ter sido gerados pelo programador ou podem fazer parte das bibliotecas da linguagem ou de bibliotecas externas.

Ambientes

- editor + cc: emacs + GCC, vi + GCC
- IDEs: DEV-C++, Eclipse
- Debugging: gdb, ddd, IDEs, valgrind

Compilação na linha-de-comandos

- `exm-max.c`
- Redirecionamento de entrada e saída.
- `exm-mean.c`
- Para projetos que não são minúsculos: `make`.
- Quando fica ainda maior: `automake`.

Debugging

Debugging

- Como o nome indica, debugging é caçar e remover bugs.
  Em português, depuração.
- Duas técnicas principais:
  - debugger
  - printf

debugger

- Um debugger é um software que permite inspecionar o estado de um programa durante a execução (valores das variáveis, pilha, chamadas de funções).
- Permite inspecionar um programa mesmo sem o código-fonte (em baixo nível).
- Permite alterar valores de variáveis a quente.
- Pode ser uma ferramenta independente ou integrada a uma IDE.

gdb

- Compilamos com `-g`.
- Executamos o gdb fornecendo o programa como entrada e digitamos comandos para o gdb. Os principais são:

```
ENTER
break (b), clear
run, next (n), step (s), continue (c)
list, list -, list n,  where, backtrack
print (p), whatis, set variable
help, help commando
show commands, show commands n
quit
```

ddd

- Interface gráfica para o gdb.

valgrind

- Verifica acessos inválidos e memória perdida.
- Compilamos com `-g` e executamos o valgrind com a mesma linha-de-comando que usaríamos normalmente.

printf

- É uma estratégia simples: adicionamos comandos para imprimir valores das variáveis ao longo da execução do programa.
- Permite ter uma visão que cobre toda a duração da execução do programa.
- Boa para encontrar erros que são difíceis de reproduzir e que são infreqüentes.
- Normalmente é usada em combinação com `ifdef` e `assert`.

Compilação condicional

- Diretivas que fazem com que o pré-processador não envie trechos do programa para o compilador.

```
#if expressao-integral-constante
#elif expressao-integral-constante
#else
#endif

#ifdef identificador
#endif

#ifndef identificador
#endif
```

- `defined identificador` ou `defined(identificador)` podem ser usados com `#if` para testar se um nome está definido. Retorna 0 ou 1.

```
#ifdef UNIX
const int nice = 10;
#elif defined(MSDOS)
const int nice = 0;
#else
const int nice = 5;
#endif
```

```
#ifdef DEBUG
printf(...)
#endif
```

```c
#include <stdio.h>
#define DEBUG 1

int somar(int V[], int n) {

  int soma = 0;

  for (n-=1; n>=0; n--) {
    soma += V[n];
    #ifdef DEBUG
    printf("%d\n",soma);
    #endif
  }

  return soma;
}


int main(void) {

  int A[10], i;

  for (i=0; i<10; i++)
    A[i] = i;

  printf("Soma: %d\n",somar(A,10));

  return 0;
}
```

exm-debug-1.c

- A constante `DEBUG` (ou qualquer outra) pode ser definida no arquivo ou na linha-de-comandos para o GCC.

```
#define DEBUG 1

#ifdef DEBUG
printf(...)
#endif
```

```
gcc -DDEBUG=1 prog.c -o prog
gcc -DDEBUG prog.c -o prog
```

assert.h

- Define a macro `assert` que testa uma condição lógica e termina o programa se ela for falsa.

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


int main() {

  double p,q;

  scanf("%lf %lf", &p, &q);

  assert(q != 0);

  printf("ratio: %lf\n",p/q);

  return 0;
}
```

exm-assert-1.c

- Pode ser desligada definindo-se `NDEBUG`.

```c
#include <stdio.h>
#define NDEBUG
#include <assert.h>

int main() {

  double p,q;

  scanf("%lf %lf", &p, &q);

  assert(q != 0);

  printf("ratio: %lf\n",p/q);

  return 0;
}
```

```
gcc -DNDEBUG prog.c -o prog
```

exm-assert-2.c

Estrutura sintática

Sentença

- Cada função de um programa C é composta de zero ou mais sentenças.
- Toda sentença em C é terminada por `;`.
- Uma sentença pode ser nula, representada como `;` apenas. É usada quando uma sentença é exigida mas uma ação não é necessária.

Tokens (átomos)

- Um token é uma cadeia de caracteres que é tratada como uma unidade na linguagem.
- Palavras-reservadas, nomes, constantes, operadores, pontuação são tokens.

Palavras-reservadas

- Eram 32:

```
char int float double
long short signed unsigned
typedef sizeof
auto register extern
const volatile static
enum struct union
if else switch case default
for while do
break continue goto
return void
```

- Há algumas novas para usos mais avançados.

Nomes ou identificadores

- São palavras que dão nomes a variáveis, tipos, funções etc.
- Regra: `[a-zA-Z_]([a-zA-Z0-9_])*`
- Apenas os primeiros 31 caracteres são considerados.[1]
- Maiúsculas e minúsculas são caracteres distintos.

[1] Não é tão simples assim, mas essa regra simplificada se aplica quase sempre.

Exemplos e contra-exemplos

- `a`
  `A`
  `_`
  `_nome`
  `data_de_nascimento`
  `Taxa1`
  `calculaValorMedio`
- `123`
  `1ou2`
  `nome-da-mae`
  `média`

Constantes literais

- Constantes inteiras

```
0
-401
1230
1e3
-20e2
0x2ab (hexadecimal)
0231  (octal)
```

Constantes literais

- Constantes racionais

```
0.0
-41.23
1e6
7e-3
-5e-4
```

Constantes literais

- Constantes caractere

```
'a'
'Z'
'@'
'\''
'\n'
'\t'
'\064'
'\x3a'
```

Constantes literais

- Constantes string

```
""
"a"
"uma linguagem"
"uma pequena frase.\n"
"uma frase que se
  quebra"
```

Constantes simbólicas

- Constantes definidas com `#define` são substituídas pelo pré-processador.

```
#define pi 3.1416
#define mess "Uma mensagem."
```

Operadores

- Alguns símbolos ou pares de símbolos são usados como operadores algébricos, lógicos, relacionais, de atribuição e outros.

```
- ++ -- ! * / % + -
& | && || ?: < <= > >= == !=
= += -= *= /= %= &= |=
() * &
```

etc.

Pontuação e comentários

- Os símbolos de pontuação são:

  `{ } ( ) ; ,`

- Brancos (espaço, tabulação, fim-de-linha) podem aparecer em qualquer lugar em qualquer quantidade.
- Os símbolos de comentários são:
  - `/* */` para um bloco de comentários.
  - `//` para comentário até o fim-de-linha.

Variáveis e tal

Declaração de variáveis

- A declaração de variáveis é da forma

```
qualificador* modificador* tipo nome {, nome}*;
```

```c
int main(void) {

  int i;
  float x, y, z;
  char c1, c2, c3;

  i = 10 * 2;
  x = y = z = 0.0;
  c1 = 'a';
  c2 = 'b';

  return 0;
}
```

exm-var-1.c

- Não há uma região exclusiva do programa para declarações, mas as variáveis devem ser declaradas antes de serem usadas.

```c
int main() {

  int i;
  i = 10 * 2;

  float x, y, z;
  x = y = z = 0.0;

  char c1, c2, c3;
  c1 = 'a';
  c2 = 'b';

  return 0;
}
```

exm-var-2.c

Atribuição

- O operador básico de atribuição é =.
- O lado esquerdo de uma atribuição deve ser um objeto na memória que pode receber um valor, como uma variável, elemento de vetor e campo de registro.

Expressões têm valor

- Expressões sempre têm um valor, embora ele não tenha que ser usado obrigatoriamente.
- Na sentença

```
x = 2;
```

  o valor da variável `x` se torna 2 e o valor da expressão `x = 2` é igual a 2.
- Na sentença

```
x = x + 1;
```

  o valor da variável `x` se torna 3 e o valor da expressão `x = x + 1` é igual a 3.

Inicialização de variáveis

- Variáveis podem ser inicializadas no momento da declaração.

```
int main() {

  int i = 0;
  double x = 1.0, y, z = 0.0;
  char c = 'a';

  return 0;
}
```

exm-var-3.c

Tipos Básicos

- São `char`, `int`, `float`, `double`.

char

- Representa caracteres e é usado para guardar números inteiros pequenos.
- Tem CHAR-BITS bit, pelo menos 8 bits.
- O valor armazenado em um char é um inteiro.
- Pode ser operado e impresso como caractere ou como inteiro.

- Em geral, a codificação dos caracteres é ASCII.
- O valor de um caractere não é igual ao seu símbolo, mas ao seu código ASCII.
- Normalmente um caractere é do tipo `char`, mas em algumas ocasiões pode ser preciso usar um `int` (e.g. `getchar` e `putchar`).
- Constantes caractere são do tipo `int`.

Caracteres barrados

- Caracteres não imprimíveis ou com significado especial são barrados, p.ex.:

`\a`	alerta sonoro	7	`\n`	fim de linha	10
`\b`	backspace	8	`\\`	barra	92
`\r`	carriage return	13	`\'`	aspas simples	39
`\f`	form feed	12	`\"`	aspas dupla	34
`\t`	tabulação	9	`\0`	caractere nulo	0

- A barra também representa caracteres com valor em octal (`'\007' '\07' '\7'`) ou hexadecimal (`'\0x1a'`).

```c
#include <stdio.h>

int main() {
  char carac;

  carac = 'a';

  carac = carac+2;

  printf("Como caracter: %c\n",carac);
  printf("Como inteiro: %d\n",carac);

  return 0;
}
```

exm-char-1.c

`int`

- Tipo integral com pelo menos 16 bits.
- Números são representados em complemento de 2.

`float` e `double`

- Tipos fracionários em ponto flutuante.
- O compilador pode reservar mais espaço para um `double` que para um `float`, mas isso não é obrigatório.
- Normalmente o `float` é de 32 bits com mantissa de aproximadamente 6 dígitos decimais e expoente entre -38 e $+38$.
- Normalmente o `double` é de 64 bits com mantissa de aproximadamente 15 dígitos decimais e expoente entre -308 e $+308$.

Blocos

- Um nome tem visibilidade restrita a um bloco.
  - Um bloco contém declarações e sentenças delimitado por `{` e `}`.
  - Blocos podem ser aninhados.
- Um arquivo pode ser visto como um bloco.
- A união de todos os arquivos que compõem o programa pode ser vista como o bloco mais externo possível.

Escopo de nomes

- Um nome definido em um bloco é visível no bloco e em todos os blocos internos.
- Um bloco interno pode definir um nome igual a outro já definido em um bloco externo. Tal definição eclipsa a definição mais externa.

Vida de nomes

- Um nome existe a partir do momento da declaração.
- Um nome continua existindo enquanto o fluxo de execução estiver dentro do bloco em que ele foi definido, em algum bloco interno ou em alguma função chamada dentro do bloco.[2]

[2] Exceção são variáveis `static`, a ver.

Operador `sizeof`

- Retorna o número de bytes necessários para armazenar na memória:
  - um tipo ou
  - uma expressão ou
  - uma variável escalar, vetor ou registro.
- Permite tomar decisões com base no tamanho da representação dos números, melhorando a portabilidade do código.

```c
#include <stdio.h>

int main() {

  printf("char %ld\n",sizeof(char));
  printf("int %ld\n",sizeof(int));
  printf("float %ld\n",sizeof(float));
  printf("double %ld\n",sizeof(double));

  printf("5+9 %ld\n",sizeof(5+9));
  printf("3.1*2.0 %ld\n",sizeof(3.1*2.0));

  float f;
  printf("f %ld\n",sizeof(f));

  return 0;
}
```

exm-sizeof-1.c

Apontadores

- Apontadores armazenam endereços de memória.
- Apontadores são declarados com o símbolo `*` preposto ao nome.
- Endereços são obtidos com o operador `&`.
- O endereço nulo tem valor `0`, com sinônimo `NULL`.

```c
#include <stdlib.h>

int main() {
  int i, *pi, **ppi;

  pi = 0;
  pi = NULL;

  pi = &i;
  ppi = &pi;

  pi = (int*) 15024;

  return 0;
}
```

exm-apont-1.c

Apontadores

- O operador `*` é o operador de indireção.

```
int j = *pi;
```

- `*` e `&` podem ser consideradas operações inversas.

```c
#include <stdio.h>

int main() {
  int i, *pi, **ppi;

  i = 10;
  pi = &i;
  ppi = &pi;

  printf("Valor de i: %d\n", i);
  printf("Endereco de i: %p\n", &i);
  printf("\n");

  printf("Valor de i via pi: %d\n", *pi);
  printf("Valor de pi: %p\n", pi);
  printf("Endereco de pi: %p\n", &pi);
  printf("\n");

  printf("Valor de i via ppi: %d\n", **ppi);
  printf("Valor de ppi: %p\n", ppi);
  printf("Endereco de ppi: %p\n", &ppi);

  return 0;
}
```

exm-apont-2.c

Apontadores Inválidos

- Para constantes.

  `&1`

- Para expressões.

  `&(x+2)`

- Para variáveis `register`.

  ```
  register r;

  &r
  ```

Controle

if

```
if (expressão)
  uma úinca sentença ou um bloco

if (expressão)
  uma úinca sentença ou um bloco
else
  uma úinca sentença ou um bloco
```

- O `if` é tomado se o valor da expressão é diferente de zero.
- Quando há mais de um `if`, o `else` se liga ao `if` mais próximo.

```c
int main() {

  int var = 1;

  if (var == 1) {
    puts("if var == 0\n");
  }

  if (var) {
    puts("if var\n");
  }

  if (1) {
    puts("if 0\n");
  }

  return 0;
}
```

exm-controle-0.c

while

```
while (expressão)
  uma úinca sentença ou um bloco
```

- O `while` é repetido enquanto o valor da expressão é diferente de zero.

for

```
for (expr-1; expr-2; expr-3)
  uma úinca sentença ou um bloco
```

- É equivalente a

```
expr-1;
while (expr-2) {
  sentenças
  expr-3;
}
```

- Qualquer expressão pode ser omitida. Se `expr-2` for omitida, seu valor é considerado igual a 1.

for

- O controle do laço pode ser feito em função de duas variáveis, usando o operador `,`.

```
for (i=0, j=0; i+n>=j; i++, j+=2)
```

do

```
do {
   sentenças
} while (expressão);
```

- O `do` é repetido enquanto o valor da expressão é diferente de zero.

```c
int main() {

  var = 0;
  while (var <= 10) {
    printf("while %d\n", var);
    var++;
  }

  for (var = 0; var <= 10; var++)
    printf("for %d\n", var);

  var = 0;
  do {
    printf("do %d\n", var);
    var++;
  } while (var <= 10);

  return 0;
}
```

exm-controle-1.c

switch

```
switch (expressão-integral) {
  case constante-integral: sentenças
                           break;
  ...
  case constante-integral: sentenças
                           break;
  default: sentenças
}
```

- Deve haver pelo menos um `case`.
- O `break` pode ser omitido em qualquer `case`.

```c
int main() {

  int i;
  scanf("%d", &i);

  switch (i) {
  case 1:
  case 2:
    printf("Igual a 1 ou 2\n");

  case 3:
  case 4:
    printf("Menor que 5\n");
    break;

  case 5:
    printf("Igual a 5\n");
    break;

  default:
    printf("Menor que zero ou maior que 5\n");
  }

  return 0;
}
```

exm-controle-2.c

Condicional

```
expressão-1 ? expressão-2 : expressão-3
```

- O valor do condicional é igual ao valor da `expressão-2` se a `expressão-1` for diferente de 0. Caso contrário é igual ao valor da `expressão-3`.

```c
int main() {

  int i, par;

  scanf("%d", &i);

  // par = 1 se i for par e 0 c.c.
  par = i%2 ? 0 : 1 ;

  printf("%d\n", par);

  return 0;
}
```

exm-controle-3.c

Desvios incondicionais

- `break`, `continue`, `goto`
- São considerados prática de programação ruim porque quebram o fluxo convencional dos outros comandos e pioram a legibilidade.

`break`

- Causa o término de uma repetição ou `switch`.
- A execução do programa continua na sentença que vem imediatamente após a repetição ou `switch`.

`continue`

- Causa a interrupção de uma iteração em uma repetição.
- A execução do programa continua no início da próxima iteração.
- Pode ser usado em `for`, `while` e `do`.
- No caso do `for`, a `expressão-3` é executada antes da primeira sentença da próxima iteração.

goto

- Causa um salto incondicional para uma sentença rotulada dentro da função corrente.
- A sintaxe do rótulo é

  ```
  identificador : sentença
  ```

- A sintaxe do comando é

  ```
  goto identificador;
  ```

- O escopo do rótulo é o corpo da função.

Operadores

Operadores

- Uma *expressão* é combinação de símbolos bem formada que representa um valor.
- Operadores têm regras de precedência e associatividade em expressões.
  - A precedência determina qual a ordem de aplicação dos operadores.
  - A associatividade determina em qual ordem operadores de mesma precedência são aplicados.

Operadores

Operador	Associatividade
`() [] . ->`	esq-dir
`++ -- + - ! ~ (tipo) * & sizeof`	dir-esq
`* / %`	esq-dir
`+ -`	esq-dir
`<< >>`	esq-dir
`< <= > >=`	esq-dir
`== !=`	esq-dir
`&`	esq-dir
`^`	esq-dir
`|`	esq-dir
`&&`	esq-dir
`||`	esq-dir
`?:`	dir-esq
`= += -= *= /= %= &= ^= |= <<= >>=`	dir-esq
`,`	esq-dir

Operadores relacionais

```
>  >=  <  <=
```

- Retornam valores inteiros 0 ou 1.
- Em C, falso é 0 e verdadeiro é $\neq 0$.
- Se a diferença entre números for muito grande, algumas expressões podem resultar em 0, embora matematicamente não sejam iguais a 0.

```
x < x+y
(x-(x+y)) < 0.0
```

Operadores de igualdade

`==` `!=`

- Retornam valores `int` 0 ou 1.

Operadores lógicos

`&&` `||` `!`

- Implementam as tabelas de **e**, **ou** e **não** lógicos.
- Expressões com valor diferente de 0 são operadas como se fossem iguais a 1.
- Expressões com os operadores `&&` e `||` deixam de ser calculadas assim que o resultado puder ser decidido, mesmo que uma parte da expressão não tenha sido executada.

Operadores de atribuição

```
= += -= *= /= %= &= ^= |= <<= >>=
```

- `var op= expressão`
  equivale a
  `var = var op expressão`

Operadores de incremento

++ --

- São unários e podem ser pré-postos ou pós-postos.
- Quando pré-posto, o valor da variável é incrementado antes de dar valor à expressão.
- Quando pós-posto, o valor da variável é incrementado depois de dar valor à expressão.

```c
int main() {

  int i;

  i = 5;
  printf("i++ %d\n", i++);

  i = 5;
  printf("++i %d\n", ++i);

  i = 5;
  printf("i-- %d\n", i--);

  i = 5;
  printf("--i %d\n", --i);

  return 0;
}
```

exm-inc-1.c

Operadores bit a bit

```
&  |  ~  ^  <<  >>
```

- Se aplicam a tipos integrais.

Operador ,

```
expr1, expr2
```

- Operador binário que permite agrupar expressões.
- O valor de `expr1` é calculado primeiro.
- O valor da expressão como um todo é o valor de `expr2`.

Modificadores e tal

Modificadores de representação

- São `short`, `long`, `signed` e `unsigned`.
- Podem aparecer combinados.

`short`

- `short` se aplica a `int`.
- O compilador pode reservar menos espaço para um `short int` que para um `int`, mas isso não é obrigatório.
- Pelo menos 16 bits.

`long`

- `long` se aplica a `int` e `double`.
- O compilador pode reservar mais espaço para um `long` que para um `int` ou que para um `double`, mas isso não é obrigatório.
- `long int` tem pelo menos 32 bits.
- `long double` pode ter precisão quádrupla, com 128 bits.

`long long int`

- O compilador pode reservar mais espaço para um `long long int` que para um `long int`, mas isso não é obrigatório.
- Pelo menos 64 bits.

`signed` e `unsigned`

- `signed` é inteiro com sinal.
  - Se aplica a
    `char`, `int`, `short int`, `long int`, `long long int`.
- `unsigned` é inteiro sem sinal.
  - Se aplica a
    `char`, `int`, `short int`, `long int`, `long long int`.
- O `unsigned` usa o bit de sinal como parte do número.

Sinônimos

- `int = signed int`
  `short = short int`
  `long = long int`
  `long long = long long int`
  `unsigned = unsigned int`
  `signed = signed int = int`
- `char` é `signed char` ou `unsigned char`, depende do compilador.

Constantes `long` e `unsigned`

- Podemos acrescentar sufixos a constantes para especificar o tipo delas.

constante	modificador	exemplo
u ou U	`unsigned`	51u
l ou L	`long`	51L
ul ou UL	`unsigned long`	51ul

- Se não for acrescentado algum sufixo, o compilador escolhe a menor dentre `int`, `long` ou `unsigned long` para representar a constante.

Constantes `float` e `double`

- Constantes fracionárias devem incluir o ponto decimal.
- Podemos acrescentar sufixos a constantes para especificar o tipo delas.

constante	modificador	exemplo
f ou F	`float`	3.14f
l ou L	`long double`	3.14L

- Se não for acrescentado algum sufixo, a constante é `double`.

Conversões implícitas

- O valor de uma expressão é convertido implicitamente para o menor tipo capaz de comportá-lo.
- Se envolve signed e unsigned, é convertido para unsigned.

```
int i;
long l;
double d;
unsigned u;

d+i
l+i
u+i
l+d
```

Conversões implícitas

- `char` e `short` são convertidos para `int`.
- O efeito de conversões que reduzem o tamanho da representação do valor de uma sentença depende do compilador.

Conversões explícitas (casting)

- Casting determina que uma expressão de um tipo deve ser tratada como se fosse de outro tipo.
- O operador unário `()` é o operador de casting.

```
(long) ('H'+5.0)
x = (double) ((int) y+1)
(float) (x=77)
```

- Podem fazer com que a representação de uma expressão seja modificada como em
  `int i; (float) i`
  ou apenas reinterpretada sem mudança no valor como em
  `float* p; (int) p`

typedef

- Permite associar um tipo a um identificador, criando sinônimos.

```
typedef int bool;
typedef int natural;
typedef long long int longao;
typedef long double doublao;
```

Entrada e saída

Entrada e saída

- Em C, operações de entrada e saída são implementadas como macros e funções de bibliotecas.
- A biblioteca `stdio` possui as funções e macros de entrada e saída.

stdio

- As funções escrevem e lêem de *streams*.
  - Uma stream representa uma sequência de objetos (bytes, registros etc.) que podem ser acessados sequencialmente.
- Uma stream é uma variável associada com um arquivo e do tipo `FILE*`.
- Quando um programa é iniciado, três streams são definidas e abertas automaticamente: `stdin`, `stdout` e `stderr`.
- Estas streams não estão associadas com arquivos, mas com o teclado e com o monitor.

Saída formatada

- `int printf(const char *fmt, ...);`

  Imprime em `stdout` a *lista de parâmetros* ... usando a *cadeia de formato* fmt para determinar a forma como cada um será impresso.

  Retorna o número de caracters impressos ou um número negativo se houver erro.

```c
#include <stdio.h>

int main() {

  int i = 10;
  int j = -3;
  float f = 3.1416;

  printf("Uma pequena frase.\n");

  printf("Um inteiro com valor %d.\n", i);

  printf("Dois inteiros com valores %d e %d.\n", i, j);

  printf("Dois inteiros com valores %d e %d.\n", j, i);

  printf("Um real com valor %f.\n", f);

  return 0;
}
```

exm-printf-1.c

Cadeia de formato

- A cadeia de formato é composta de
  - zero ou mais caracters comuns que serão copiados na saída e
  - *especificações de conversão* que se aplicam a zero ou mais parâmetros na lista.

Especificação de conversão

- Cada especificação de conversão tem a forma

```
%[parâmetro][flags][largura][.precisão][tamanho]conversão
```

`%[parâmetro][flags][largura][.precisão][tamanho]conversão`

Espec.	Parâmetro	Conversão
d,i	int	inteiro decimal com sinal
u	unsigned	decimal sem sinal
o	unsigned	octal sem sinal
x,X	unsigned	hexadecimal sem sinal
f,F	double	arredondamento e notação [-]ddd.ddd
e,E	double	arredondamento e notação científica
g,G	double	e se o expoente é menor que -4 ou maior que a precisão, senão f
a,A	double	formato hexadecimal
c	char	unsigned char
s	const char*	nenhuma
p	void*	hexadecimal
n	int*	o número de caracteres impressos até aquele ponto é armazenado na variável

- Os indicadores de tamanho especificam modificações da representação.

char	%c, %hhi
unsigned char	%c, %hhu
short	%hi
unsigned short	%hu
int	%i, %d
unsigned	%u
long	%li
unsigned long	%lu
long long	%lli
unsigned long long	%llu
float	%f, %F, %g, %G, %e, %E, %a, %A
double	%lf, %lF, %lg, %lG, %le, %lE, %la, %lA
long double	%Lf, %LF, %Lg, %LG, %Le, %LE, %La, %LA

`%[parâmetro][flags][largura][.precisão][tamanho]conversão`

- Especifica o número mínimo de dígitos que deve ser mostrado nas conversões `d i o u x X`.
- Especifica o número mínimo de dígitos à direita da vírgula nas conversões `a A e E f F`.
- Especifica o número máximo de dígitos significativos nas conversões `g G`.
- Especifica o número máximo de caracteres impressos nas conversões `s S`.

`%[parâmetro][flags][largura][.precisão][tamanho]conversão`

- Um número decimal que especifica largura mínima do campo onde o valor vai ser impresso.
- O valor impresso será alinhado à direita ou à esquerda e o campo será preenchido com espaços ou zeros.
- Se o campo for menor que o valor a ser impresso, a largura é ignorada.
- A largura do campo pode ser dada pelo próximo parâmetro para printf, usando `*`, ou pelo parâmetro `m`, usando `*m$`.

```c
#include <stdio.h>

int main() {

  char caractere = 'z';
  int inteiro = 8192;
  float flutuante = 3.1416;
  char string[] = "string";

  printf("%1c \n", caractere);
  printf("%3c \n", caractere);
  printf("%6c \n", caractere);
  printf("%9c \n", caractere);

  printf("%1d \n", inteiro);
  printf("%3d \n", inteiro);
  printf("%6d \n", inteiro);
  printf("%9d \n", inteiro);

  printf("%9.1d \n", inteiro);
  printf("%9.2d \n", inteiro);
  printf("%9.3d \n", inteiro);
  printf("%9.4d \n", inteiro);

  printf("%1f \n", flutuante);
  printf("%3f \n", flutuante);
  printf("%9f \n", flutuante);
  printf("%12f \n", flutuante);

  printf("%12.2f \n", flutuante);
  printf("%12.4f \n", flutuante);
  printf("%12.6f \n", flutuante);
  printf("%12.8f \n", flutuante);

  printf("%1s \n", string);
  printf("%3s \n", string);
  printf("%6s \n", string);
  printf("%9s \n", string);
  printf("%12s \n", string);

  printf("%12.1s \n", string);
  printf("%12.2s \n", string);
  printf("%12.3s \n", string);
  printf("%12.4s \n", string);
  printf("%12.5s \n", string);

  return 0;
}
```

exm-printf-3.c

```c
#include <stdio.h>

int main() {

  int i = 100;
  float f = 3.1416;

  printf("%5d\n", i);
  printf("%5d.2\n", i);

  int largura = 7, precisao = 3;

  printf("%*d\n", largura, i);
  printf("%*.*f\n", largura, precisao, f);

  return 0;
}
```

exm-printf-2.c

`%[parâmetro][flags][largura][.precisão][tamanho]conversão`

- Alteram o formato do campo.

0	Usa zeros ao invés de espaços para alinhar à direita
-	Alinha à esquerda
+	Sempre acrescenta um sinal + ou -
espaço	Acrescenta um espaço antes de número positivo
#	Forma alternativa para g,G,f,F,e,E,o,x,X

- Mais detalhes no manual.

```c
#include <stdio.h>


int main() {

  int i = 51;

  printf("0: %010d \n",i);
  printf("-: %-10d \n",i);
  printf("+: %+10d \n",i);

  return 0;
}
```

exm-printf-4.c

`%[parâmetro][flags][largura][.precisão][tamanho]conversão`

- Tem a forma `n$` indicando que o n-ésimo parâmetro deve ser usado na próxima conversão.
- Permite o reuso de um parâmetro múltiplas vezes e em ordem arbitrária.
- Se `$` for usado, ele deve ser usado em todas as conversões.

```
printf("%2$d %2$c, %1$d %1$c \n", 73, 101);
```

Lista de parâmetros

- Os parâmetros são usados na ordem em que aparecem na cadeia de formato.
- Cada `*` e cada especificador de conversão pegam o próximo parâmetro da lista
- Opcionalmente, podemos especificar o índice do parâmetro na lista ou o próximo parâmetro, escrevendo `%m$` ao invés de `%` e `*m$` ao invés de `*`.

Entrada formatada

- `int scanf(const char *fmt, ...);`
  Lê valores de `stdin` usando a *cadeia de formato* fmt para determinar a conversão e armazena em variáveis através dos apontadores na lista ...
- Retorna o número de variáveis convertidas, ou EOF se houver uma condição que impediu qualquer conversão, como por exemplo, o fim da entrada. Seta a variável `errno` em caso de erro.

Cadeia de formato

- A cadeia de formato é composta de
  - *Especificações de conversão* que se aplicam a zero ou mais parâmetros na lista. Opcionalmente podem ter flags.
  - Zero ou mais caracters comuns que são casados com a entrada.
  - Espaço, tabulação e fim-de-linha na cadeia de formato casa com zero ou mais espaços, tabulações e fins-de-linha na entrada.

Flags

*	O campo deve ser lido mas não deve ser armazenado em uma variável.
número	Limita o número de dígitos lidos com i ou f, ou o número de símbolos com s.
'	Com números, especifica que há separadores de milhar.
m	Com cadeias, faz com que memória suficiente para a cadeia seja alocada.
`[caracteres]`	Lê enquanto houver caracters da lista. Apenas para s.
`[^caracteres]`	Lê enquanto houver caracters fora da lista, para s.

- `[]` permite faixa com hífen, p.ex., 2-7, a-j.
- `scanf` também admite conversões com `%n$`.
- Mais detalhes no manual.

Especificadores de conversão

	Casa com	Parâmetro
d	inteiro decimal com sinal	int*
i	inteiro decimal, hexa ou octal com sinal	int*
u	decimal sem sinal	unsigned*
o	octal sem sinal	unsigned*
x, X	hexadecimal sem sinal	unsigned*
e,E,f,g	número em ponto flutuante com sinal	float*
s	uma seqüência sem espaços, tabs ou fls	char*
c	uma seqüência de tamanho w	char*
p	void*	hexadecimal
[	uma cadeia formada pelos caracteres especificados	char*

Retorno

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

int main() {

  char p[30];
  errno = 0;
  int n = scanf("%m[a-z]", p);

  if (n == 1) {
    printf("read: %s\n", p);
  }
  else if (errno != 0) {
    perror("scanf");
  }
  else {
    fprintf(stderr, "No matching characters\n");
  }

  return 0;
}
```

exm-scanf-1.c

Saída não-formatada

- `int putchar(int c);`

  Imprime o inteiro `c` convertido para `unsigned char`. Retorna `c` convertido para `unsigned char` ou EOF se houver erro.

- `int putc(int c, FILE* stream);`

  Imprime o inteiro `c` convertido para `unsigned char` na stream indicada. Retorna `c` convertido para `unsigned char` ou EOF se houver erro.

- `int puts(const char* s);`

  Escreve a cadeia `*s` no monitor e retorna um número não-negativo ou EOF se houver erro.

```c
#include <stdio.h>

int main() {

  puts("Tres caracteres: ");
  putchar('Z');
  putchar('\n');
  putchar(48);

  puts("Uma pequena frase.\n");

  return 0;
}
```

exm-put-1.c

Entrada não-formatada

- `int getchar(void);`

  Equivalente a `getc(stdin)`.

- `int getc(FILE *stream);`

  Retorna um caractere lido do teclado convertido para `int` ou EOF em caso de erro.

Arrays

Vetores

- A definição de um vetor tem a forma

  ```
  tipo nome[n]
  ```

- O tamanho `n` deve ser integral e positivo.
- Os índices dos elementos do vetor vão de 0 a `n`-1.
- Os elementos do vetor são armazenados em posições consecutivas na memória.
- O tamanho de um array não muda.

Acesso

- O operador `[]` é usado para acesso a elementos do vetor.
- Os limites do vetor não são verificados.
- Um acesso fora da faixa válida tem efeito difícil de prever e depende do ambiente.
  - Pode retornar ou modificar outra variável do programa.
  - Pode tentar retornar ou modificar uma região fora do programa.

```c
#include <stdio.h>

#define n 8

int main() {

  int A[n], i;

  for (i=0; i<n; i++)
    A[i] = i;

  for (i=0; i<n; i++)
    printf("%d ",A[i]);

  return 0;
}
```

exm-vetores-1.c

```c
#include <stdio.h>


int main() {

  int n, i;

  scanf("%d",&n);

  int A[n];

  for (i=0; i<n; i++)
    A[i] = i;

  for (i=0; i<n; i++)
    printf("%d ",A[i]);

  return 0;
}
```

exm-vetores-1b.c

Inicialização

- A inicialização de um vetor tem a forma

  `tipo nome[c] = {`s_1`,...,`s_c`}`
  `tipo nome[] = {`s_1`,...,`s_c`}`

- Se a lista tem tamanho menor que `c`, os últimos elementos são inicializados com zero.

  `tipo nome[c] = {0}`

- Se a lista é maior que `c`, os últimos elementos são ignorados.
- Vetores declarados com tamanho definido por variável não podem ser inicializados.

Vetores e apontadores

- O nome de um vetor unidimensional é um apontador constante para o endereço do elemento na posição 0.
- Vamos supor que

```
int V[] = {1,2,3,4,5};
```

- Então

`V` é equivalente a `&V[0]`.

Aritmética de apontadores

- Se `p` é um apontador para algum elemento de um vetor então a sentença `p+1` resulta no endereço para recuperar ou armazenar o elemento seguinte.
- As expressões `p++`, `++p`, `p--` e outras têm sentido similar.
- Se `p` e `q` são apontadores para elementos de um vetor, `p-q` resulta no número de elementos do vetor entre `p` e `q`.

- Vamos supor que

```
int V[n];
int *p;
```

- Então

`p = V` é equivalente a `p = &V[0]`

`p = V+1` é equivalente a `p = &V[1]`

- De forma mais geral,

`V[i]` é equivalente a `*(V+i)`.

```c
#include <stdio.h>

int main() {

  int V[10] = {0,1,2,3,4,5,6,7,8,9};

  printf("V[0]: %d\n", V[0]);
  printf("*V: %d\n", *V);

  printf("V[1]: %d\n", V[1]);
  printf("*(V+1): %d\n", *(V+1));

  printf("V[5]: %d\n", V[5]);
  printf("*(V+5): %i\n",*(V+5));

  printf("(V+9)-(V+5): %li\n",(V+9)-(V+5));
  printf("(V+5)-(V+9): %li\n",(V+5)-(V+8));

  return 0;
}
```

exm-vetores-2.c

Matrizes, cubos etc.

- Pares de colchetes adicionais acrescentam novas dimensões a um array.

```
int M[5][3], C[10][10][10];
```

- O acesso a membros é feito pela composição de operadores `[]`.

```
M[3][2] += M[1][1] + 1;
C[0][1][1] = 0;
```

Inicialização

- A inicialização de um array multidimensional é por linhas.
- Cada dimensão pode ser separada por `{ }`.
- Valores omitidos são inicializados como zero.
- O tamanho da primeira dimensão pode ser omitido e será igual ao número de pares de chaves correspondentes.

```
int M[2][3] = {11,12,13,21,22,23};
int M[2][3] = {{11,12,13},{21,22,23}};
int M[][3] = {{11,12,13},{21,22,23}};
```

```
int C[2][2][3] = {111,112,113,121,122,123,
                  211,212,213,221,222,223};
int C[][2][3] = {{111,112,113},{121,122,123},
                 {211,212,213},{221,222,223}};
```

Arrays multi-dimensionais e apontadores

- Supondo `int M[3][5]`,
  - `M[i]` pode ser pensado como a linha `i`.
  - `M[i][j]` pode ser pensado como o elemento na linha i, coluna j.
  - O nome `M` é equivalente a `&M[0]`, um vetor com 5 elementos.
  - A partir de `&M[0][0]` está o espaço consecutivo para 15 ints.
  - Se os elementos forem acessados na ordem em que estão alocados, índices mais a direita variam mais depressa. Ou seja, os elementos estão armazenados linha-a-linha.

```c
#include <stdio.h>

int main() {

  int i,j;
  int M[3][5] = { {1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15} };

  for (i=0; i<3; i++) {
    for (j=0; j<5; j++) {
      printf("%2d ", M[i][j]);
    }
    printf("\n");
  }

  i = 2;
  j = 3;

  printf("\n%d\n", M[i][j] );
  printf("%d\n", *(M[i] + j) );
  printf("%d\n", (*(M + i))[j] );
  printf("%d\n", *((*(M + i)) + j) );
  printf("%d\n", *(&M[0][0] + 5*i + j) );

  return 0;
}
```

exm-vetores-6a.c

```c
#include <stdio.h>

int main() {

  int M[3][4] = {{11,12,13,14},
                 {21,22,23,24},
                 {31,32,33,34}};

  printf("**M %i\n",**M);

  printf("*M %i\n", *(M[0]));
  printf("*(*M+2) %i\n",*(*M+2));

  printf("*(&M[0][0]+2) %i\n",*(&M[0][0]+2));
  printf("*(M[2]+1) %i\n",*(M[2]+1));
}
```

exm-vetores-6b.c

- Para qualquer vetor, o mapeamento entre valores de apontadores e índices do vetor é dado por uma função de mapeamento de armazenamento.

```
int M[r][s];
M[i][j] = *(&M[0][0] + i*s + j)

int C[r][s][t];
C[i][j][k] = *(&C[0][0][0] + i*s*t + j*t + k)
```

sizeof

- `sizeof` é uma exceção em que o nome do array não é tratado como um apontador. Ele retorna o número de bytes ocupados pelo array.

```c
#include <stdio.h>

int main() {

  int A[10];
  int B[10][20];
  int C[10][20][30];

  printf("A %ld\n",sizeof(A));
  printf("B %ld\n",sizeof(B));
  printf("C %ld\n",sizeof(C));

  return 0;
}
```

exm-sizeof-2.c

Cadeias de caracteres

Cadeias de caracteres (strings)

- Uma string em C é um vetor de `char` que tem o caractere 0 (`'\0'`) marcando o fim da cadeia.

```
char s[12] = "palavra";
```

p	a	l	a	v	r	a	\0				

- A inicialização

  ```
  char s[] = "abc";
  ```

  é equivalente a

  ```
  char s[] = {'a', 'b', 'c', '\0'};
  ```

- O `\0` é verificado por funções de saída, que imprimem até encontrá-lo, e por outras funções que processam strings.
- As funções de entrada normalmente acrescentam o `\0`.
- É responsabilidade do programador definir o vetor grande o bastante para armazenar o `\0`.

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


int main () {

  char s1[] = "abcdef";
  int i;

  for (i=0; i<strlen(s1); i++)
    putchar(s1[i]);

  putchar('\n');

  return 0;
}
```

exm-strings-1.c

```c
#include <stdio.h>

int main () {

  char s1[] = "abcdef";
  char* s2 =  "ghijkl";
  char* pc;

  for (pc = s1; *pc != '\0'; pc++)
    putchar(*pc);

  for (pc = s2; !*pc; pc++)
    putchar(*pc);

  putchar('\n');

  return 0;
}
```

exm-strings-2.c

Entrada

- A função `scanf` possui alguns especificadores de conversão para strings:
  - `s`: lê caracteres até encontrar um branco.
  - `nc`: lê n caracteres, não acrescenta `\0`.
  - `[]`: lê caracteres enquanto fizerem parte do conjunto especificado.
  - `[^]`: lê caracteres enquanto não fizerem parte do conjunto especificado.

- Entrada: `abacate abacaxi\n`
- Resultado:

Conversão	string
"s"	abacate\0
"10c"	abacate ab
"[abc]"	abaca\0
"[^e]"	abacat\0
"[^\n]"	abacate abacaxi\0

Várias strings

- Várias strings podem ser manipuladas como matrizes de caracteres ou como vetores de apontadores para caractere.

```
char m[5][10] = {"Carlos","Dora","Lia","Lea",""};

char* as[] = {"Carlos","Dora","Lia","Lea",""};


char S[k][51];

for (i=0; i<k; i++)
    scanf("%s ", S[i]);
```

string.h

- Declara várias funções para manipular strings.
- Declara o tipo `size_t`, que é grande o bastante para armazenar o tamanho do maior vetor que pode ser construído na linguagem.

- `size_t strlen(const char *s);`

  Retorna o número de caracteres em `s`, sem contar o `\0`.

- `char* strcat(char* dest, const char* src);`

  Concatena `src` em dest, sobrescrevendo o `\0` em `dest` e acrescentando um `\0`. Retorna um apontador para `dest`. A cadeia `dest` deve ser grande o suficiente.

- `char *strncat(char *dest, const char *src, size_t n);`

  Similar a `strcat`, porém apenas os `n` primeiros caracteres de `src` são concatenados.

- `int strcmp(const char *s1, const char *s2);`

  Compara `s1` e `s2` e retorna um inteiro menor, igual ou maior que zero se o `s1` for respectivamente, lexicograficamente menor, igual ou maior que `s2`.

- `int strncmp(const char *s1, const char *s2, size_t n);`

  Similar a `strcmp`, porém no máximo os `n` primeiros caracteres de `s1` e `s2` são comparados.

- `char *strcpy(char *dest, const char *src);`

  Copia `src` em `dest`, incluindo o `\0`. Retorna um apontador para `dest`. A cadeia `dest` deve ser grande o suficiente.

- `char *strncpy(char *dest, const char *src, size_t n);`

  Similar a `strcpy`, no máximo os primeiros `n` caracteres de `src` serão copiados. Se `\0` não estiver entre os n caracteres, `dest` não será terminada por `\0`.

Funções de busca

`strchr`	retorna um apontador para a primeira ocorrência de um caractere
`strrchr`	retorna um apontador para a última ocorrência de um caractere
`strpbrk`	retorna um apontador para a primeira ocorrência de um caractere de um conjunto
`strspn`	retorna o maior prefixo de uma cadeia que tenha apenas caracteres em um conjunto
`strstr`	retorna um apontador para o início da primeira ocorrência de uma subcadeia

Funções de conversão em `stdlib` e `stdio`

`atoi`	string para int
`atol/strtol`	string para long
`atoll/strtoll`	string para long long
`strtof`	string para float
`strtod`	string para double
`strtoul`	string para unsigned long
`strtoull`	string para unsigned long long
`sprintf`	conversões para string

Funções

Função

- Em C uma função pode ser vista como um tipo especial de bloco.
- As regras de escopo de nomes se aplicam a funções da mesma forma que a blocos.

Definição

- A definição de uma função tem a forma

```
[tipo|void] id(lista-de-declarações) {
  declarações e sentenças
}
```

- O tipo informa ao compilador como o valor retornado pela função vai ser convertido se for preciso.
- A palavra `void` é usada para dizer que a função não retorna nenhum valor.
- Se nada for especificado o retorno é `int`.
- A lista de declarações pode ser vazia, representada por `f(void)`.
- (`f()` é uma função com um número variável de parâmetros declarada na forma tradicional.)

Chamada

- Uma chamada de função tem a forma

  `id(lista-de-expressões)`

- A lista de expressões correspondem aos parâmetros na mesma ordem da declaração da função.

Retorno

- O retorno de função tem sintaxe

```
return;
return sentença;
return (sentença);
```

- Pode haver zero ou mais `return` em uma função.
- A cláusula `return` faz com que o controle retorne para o ponto em que a função foi chamada.
- Se não houver `return`, o controle retorna quando o fim do corpo da função é alcançado.

Retorno

- O valor retornado, se houver, é convertido para o tipo da função se for necessário.
- O valor retornado não precisa ser usado.
- Uma função que retorna valor pode não ter uma cláusula `return`.
- Funções não podem retornar vetores ou funções.

Parâmetros

- Todos os parâmetros para uma função são passados por valor.
- Parâmetros não podem ser inicializados.
- O mecanismo de passagem por referência é obtido com o uso de apontadores como parâmetros e endereços de variáveis nas chamadas.

```c
#include <stdio.h>

void swap_bad(int a, int b) {

  int x = a;
  a = b;
  b = x;
}


void swap_good(int* p, int* q) {

  int x = *q;
  *q = *p;
  *p = x;
}
```

```c
int main() {

  int x = 51, y = 42;

  printf("x=%d y=%d\n",x,y);

  swap_bad(x,y);
  printf("bad  x=%d y=%d\n",x,y);

  swap_good(&x,&y);
  printf("good x=%d y=%d\n",x,y);

  return 0;
}
```

exm-par-1.c

- No exemplo, `p` e `q` são dois apontadores passados por valor.
- Se modificarmos o valor deles (isto é, o endereço armazenado neles) nada acontece com `x` e `y`.
- A troca acontece porque modificamos o *valor apontado* por eles.

Declaração

- Funções devem ser declaradas antes de ser chamadas.
- Declarações têm a forma

  ```
  [tipo|void] id(lista-de-declarações);
  ```

- As declarações são chamadas de *protótipos*.
- Os nomes de parâmetros podem ser omitidos na declaração.

Declaração

- Se a declaração de uma função `X` for omitida, quando o compilador encontra uma chamada da função `X` com qualquer número de parâmetros ele assume que existe uma declaração da forma

  ```
  int X()
  ```

- Nada é assumido a respeito dos parâmetros.

Vetores como parâmetros de função

- Um parâmetro declarado como um vetor é um apontador.

  ```
  int f(int V[]){}
  ```

  é equivalente a

  ```
  int f(int* V){}
  ```

- O endereço base do vetor é passado por valor.
- O tamanho do vetor passado para uma função não pode ser recuperado por `sizeof`.

```c
#include <stdio.h>

#define n 51


void func(int array[n]) {
  int j;

  for (j=0; j<n; j++)
    array[j] = j;

  printf("array dentro %lu\n",sizeof(array));
  printf("*array dentro %lu\n",sizeof(*array));
}


int main() {

  int x[n];

  printf("x fora %lu\n",sizeof(x));

  func(x);
}
```

exm-par-2a.c

- Precisamos passar o tamanho do vetor também.

```c
void func(int array[13])

void func(int n, int array[])

void func(int n, int array[n])
```

Arrays multi-dimensionais como parâmetros

- Um parâmetro declarado como um array multi-dimensional também é um apontador.
- Precisamos passar todas as dimensões na declaração do array como parâmetro, mas podemos omitir a primeira.

```c
void func(M[13][23]);
void func(M[][23]);

void func(int rows, int cols, int M[rows][cols]);
void func(int rows, int cols, int M[][cols]);

void func(int slice, int rows, int cols, int C[slice][rows][cols]);
void func(int slice, int rows, int cols, int C[][rows][cols]);
```

Estilo tradicional

- A forma tradicional de definição é

```
tipo id (lista-de-identificadores)
  definições-dos-parâmetros
{...}
```

- Na forma tradicional o tipo dos parâmetros deve ser assegurado pelo programador, não há conversão.

```
int (a,b,c)
  int a,b;
  double c;
{
  return a+b*c;
}
```

enum, struct, union

Enumeração

- Uma enumeração é um tipo de dados que pode assumir um conjunto restrito de valores.
- Em C, são valores inteiros que têm um nome.

- A definição típica de uma enumeração tem a forma

```
enum id {lista-de-identificadores};
```

- Por exemplo

```
enum dias {dom,seg,ter,qua,qui,sex,sab};
```

- Esta definição cria o tipo `enum dias`.
- Os enumeradores são os identificadores `dom, ..., sab`.
- Os enumeradores são constantes `int` com valores 0, 1, ...

```c
int main() {

  enum dias {dom,seg,ter,qua,qui,sex,sab};
  enum dias d1, d2;

  d1 = dom;
  d2 = seg;

  if (d1 == d2)
    ;

  while (d1 != qua)
    ;

  d2 = 8;  // Estranho, mas funciona

  int i = 2 * d1;  // idem

  return 0;
}
```

Outras formas de definição

```
enum cor {azul,verde} c1,c2;
```

Outras formas de definição

```
enum cor {azul,verde} c1,c2;
```

```
enum cor {azul=4,verde,roxo=3,rosa};
```

o que implica verde=5, rosa=4 e azul==rosa.

Outras formas de definição

```
enum cor {azul,verde} c1,c2;
```

```
enum cor {azul=4,verde,roxo=3,rosa};
```

o que implica verde=5, rosa=4 e azul==rosa.

```
enum {azul,verde} c1,c2;
```

Outras formas de definição

```
enum cor {azul,verde} c1,c2;
```

```
enum cor {azul=4,verde,roxo=3,rosa};
```

o que implica verde=5, rosa=4 e azul==rosa.

```
enum {azul,verde} c1,c2;
```

```
typedef enum cor {azul,verde} cor;
```

Outras formas de definição

```
enum cor {azul,verde} c1,c2;
```

```
enum cor {azul=4,verde,roxo=3,rosa};
```

o que implica verde=5, rosa=4 e azul==rosa.

```
enum {azul,verde} c1,c2;
```

```
typedef enum cor {azul,verde} cor;
```

```
typedef enum {azul,verde} cor;
```

Escopo

- Os enumeradores estão no mesmo espaço de nomes das variáveis.

```
enum cor {azul,verde};
float azul; /* nao funciona */
```

- Os nomes de enumerações estão em um espaço de nomes próprio.

```
enum cor {azul,verde};
float cor; /* funciona */
```

Enumerações e funções

- Uma enumeração pode ser passada como parâmetro e retornada por uma função.

```
typedef enum {boi,camelo,pato} E;

E funcao(int i, E e);
```

Registro

- Um registro é um conjunto de variáveis. Elas não precisam ser do mesmo tipo.

- A definição típica de registros tem a forma

```
struct identificador {
  defs-de-campos;
  ...
  defs-de-campos;
};
```

- Os campos são definidos da mesma forma que variáveis.
- Por exemplo:

```
struct item {
  char tipo;
  int valor, situacao;
};
```

- Esta definição cria o tipo `struct item`.

Operadores

- O operador de acesso aos campos de um registro é o ponto `.`
- O operador de acesso aos campos de um registro apontado é a seta `->`.
- O operador de atribuição = pode ser usado com registros e causa a atribuição de valores campo a campo.
- Outros operadores devem ser aplicados campo a campo.

```c
struct tarefa {
  char descricao[30];
  int duracao;
};

struct pessoa {
  char nome[50];
  int idade;
  struct tarefa tarefas[5];
};

struct pessoa p1;

printf("%s %d %s\n",
       p1.nome, p1.idade,
       p1.tarefas[1].descricao);

struct pessoa* ap = &p1;

printf("%s %d %s\n",
       ap->nome, ap->idade,
       ap->tarefas[1].descricao);

return 0;
```

exm-struct-1.c

Outras formas de definição

```
struct carta {
  char naipe;
  int valor;
} c1, c2, baralho[52];

typedef struct carta carta;

carta c3, c4;
```

Outras formas de definição

```
struct carta {
  char naipe;
  int valor;
} c1, c2, baralho[52];

typedef struct carta carta;

carta c3, c4;
```

```
struct {
  char naipe;
  int valor;
} c1, c2;
```

Outras formas de definição

```
struct carta {
  char naipe;
  int valor;
} c1, c2, baralho[52];

typedef struct carta carta;

carta c3, c4;
```

```
struct {
  char naipe;
  int valor;
} c1, c2;
```

```
typedef struct {
  char naipe;
  int valor;
} carta;
```

Composição

- A registro pode agrupar um número arbitrário de dados de tipos diferentes.
- Vetores, registros e apontadores também podem ser membros de registros.
- É possível definir um apontador dentro de um registro que é do seu próprio tipo:

```
struct qualquer {
  ...
  struct qualquer *proximo;
}
```

Escopo

- Os nomes dos membros de uma registro devem ser distintos.
- Registros distintos podem ter membros com nomes iguais.
- Os nomes de registros estão em um espaço de nomes próprio.

```
struct aux {int i;};
float aux; /* funciona */
```

Inicialização de Registros

- Registros podem ser inicializados de forma similar aos vetores:
- ```
  struct ponto{
    int x;
    int y;
  };

  struct ponto p1 = {220,110};
  struct ponto p2 = {110};   /* y = 0 */
  ```
  ```

Registros e funções

- Uma registro é sempre passado por valor e todos os membros, incluindo vetores e registros são copiados.
- Registros podem ser retornados por funções. Eles são retornados por valor.

Uniões

- Uma união define um conjunto de campos que serão armazenados numa porção compartilhada da memória, isto é, apenas um campo será armazenado de cada vez.
- É responsabilidade do programador interpretar corretamente o dado armazenado em uma união.
- O espaço alocado é suficiente para armazenar o maior dos seus campos.
- A sintaxe é similar à de `struct`.

```c
#include <stdio.h>


int main(void) {

  union int_or_float {
    int i;
    float f;
  } n;


  n.i = 4444;
  printf("%10d %16.10f\n", n.i, n.f);

  n.f = 4444.0;
  printf("%10d %16.10f\n", n.i, n.f);

  return 0;
}
```

exm-union-1.c

Representação numérica

Representação de números

- Em um computador típicos, temos apenas bits.
- Há formas padronizadas para organizar cadeias de bits para representar números.

Overflow, underflow

- Overflow: acontece quando um cálculo produz um valor de magnitude maior que a capacidade de representação do armazenamento.
- Underflow: acontece quando um cálculo produz um valor de magnitude menor que a capacidade de representação do armazenamento.

Representação de inteiros com sinal

- Pode ser feita de várias formas, dentre elas:
  - Sinal e magnitude.
  - Complemento de um.
  - Complemento de dois.
  - Excess-k.

Sinal e magnitude

- Um bit (normalmente o mais significativo) representa o sinal.
- Os bits restantes representam o número.
- O bit de sinal é 0 para positivos e 1 para negativos.
- O zero é representado de duas maneiras.

$$
\begin{aligned}
-127 &= 1111\ 1111 \\
-126 &= 1111\ 1110 \\
-125 &= 1111\ 1101 \\
&\dots \\
-2 &= 1000\ 0010 \\
-1 &= 1000\ 0001 \\
0 &= 1000\ 0000 \\
0 &= 0000\ 0000 \\
1 &= 0000\ 0001 \\
2 &= 0000\ 0010 \\
&\dots \\
125 &= 0111\ 1101 \\
126 &= 0111\ 1110 \\
127 &= 0111\ 1111
\end{aligned}
$$

- Exige que os circuitos de adição examinem os sinais e o valor absoluto dos números para determinar se a operação que deve ser aplicada é de soma ou subtração.
- Usada nos primeiros computadores binários como o IBM 7090 (fim dos anos 1950 e anos 1960).

Complemento de um

- Um número negativo é obtido a partir da complementação bit a bit dos dígitos no número positivo. E vice-versa.
- O zero é representado de duas maneiras.

$$
\begin{aligned}
-127 &= 1000\ 0000 \\
-126 &= 1000\ 0001 \\
-125 &= 1000\ 0010 \\
&\dots \\
-2 &= 1111\ 1101 \\
-1 &= 1111\ 1110 \\
0 &= 1111\ 1111 \\
0 &= 0000\ 0000 \\
1 &= 0000\ 0001 \\
2 &= 0000\ 0010 \\
&\dots \\
125 &= 0111\ 1101 \\
126 &= 0111\ 1110 \\
127 &= 0111\ 1111
\end{aligned}
$$

- Para somar dois números faz-se uma soma binária convencional, mas é necessário adicionar o último vai-um, se houver.

  Por exemplo, -1 + 2:

```
 1 1111 110
   1111 1110
  +0000 0010
---------------
   0000 0000
   0000 0001
---------------
   0000 0001
```

- Exige que os circuitos de adição examinem o último vai-um e façam mais uma soma.
- Usada em computadores como o PDP-1 e o UNIVAC 1100/2200 (anos 1960).

Complemento de dois

- Um número negativo é obtido complementando os bits do número positivo e somando 1. E vice-versa.
- O zero é representado apenas de uma maneira.
- Os números positivos e o zero têm o primeiro bit igual a 0

$$
\begin{aligned}
-128 &= 1000\ 0000 \\
-127 &= 1000\ 0001 \\
-126 &= 1000\ 0010 \\
&\dots \\
-3 &= 1111\ 1101 \\
-2 &= 1111\ 1110 \\
-1 &= 1111\ 1111 \\
0 &= 0000\ 0000 \\
1 &= 0000\ 0001 \\
2 &= 0000\ 0010 \\
3 &= 0000\ 0011 \\
&\dots \\
126 &= 0111\ 1110 \\
127 &= 0111\ 1111
\end{aligned}
$$

- Para n bits, o complemento um número x em complemento de dois é $\overline{x}$ tal que $x + \overline{x} = 2^n$.
- Para n bits, a faixa de números em complemento de dois é $[-2^{n-1}, +2^{n-1} - 1]$.

- Usado na maioria dos computadores binários, porque evita a necessidade de circuitos para determinar se a operação que deve ser aplicada é de soma ou subtração e de soma de vai-um.

Soma

- Somam-se os números e descarta-se qualquer vai-um além do último bit.

3+(-8) = -5

```
  0 0000 000
    0000 0011
  +1111 1000
-------------
    1111 1011
```

-1+2 = 1

```
  1 1111 110
    1111 1111
  +0000 0010
-------------
    0000 0001
```

-1+(-5) = -6

```
  1 1111 110
    1111 1110
  +1111 1011
-------------
    1111 1001
```

Overflow na soma

- Se os números são ambos positivos ou ambos negativos, um overflow ocorre quando o resultado tem sinal oposto:

$$(+A) + (+B) = -C \text{ ou } (-A) + (-B) = +C$$

- Essa situação é detectada comparando os dois últimos vai-um. Se eles forem diferentes, aconteceu um overflow.

-7+(-6)	6+5	6+1	-2+(-2)
1 000	0 100	0 000	1 110
1010	0110	0110	1110
+1001	+0101	+0001	+1110
0011	1011	0111	1100

Subtração

- A subtração é realizada fazendo-se a negação do subtraendo e somando: $A - B = A + (-B)$.
- Haverá overflow na subtração se houver overflow na soma.

Multiplicação

- A multiplicação de dois números de n bits pode resultar em um número de até $2n$ bits. Se os operandos forem transformados em números de $2n$ bits então a multiplicação pode ser feita por somas sucessivas.
- Esse método resulta em muitas operações com $2n$ bits e há metodos mais eficientes que estão fora do nosso escopo.

Divisão

- A divisão de dois números de n bits é feita por subtrações sucessivas usando $2n$ bits.

Excess-k

- Escolhe-se um valor positivo fixo k e representam-se os números no intervalo $[-k, 2^n - k - 1]$.
- Cada valor é k unidades maior que o valor representado.
  - $-k$ é representado por todos os bits 0.
  - O zero é representado por k.

Excess-127

$$
\begin{aligned}
-127 &= 0000\ 0000 \\
-126 &= 0000\ 0001 \\
-125 &= 0000\ 0010 \\
&\dots \\
-2 &= 0111\ 1101 \\
-1 &= 0111\ 1110 \\
0 &= 0111\ 1111 \\
1 &= 1000\ 0000 \\
2 &= 1000\ 0001 \\
3 &= 1000\ 0010 \\
&\dots \\
126 &= 1111\ 1101 \\
127 &= 1111\ 1110 \\
128 &= 1111\ 1111
\end{aligned}
$$

Excess-k

- É vantajosa em aplicações onde é necessário comparar dois números eficientemente.

Representação de fracionários

- Há duas formas principais:
  - Ponto-fixo.
  - Ponto-flutuante.

Ponto-fixo

- Representa números com o ponto decimal em uma posição fixa.
- Por exemplo, um número em ponto-fixo de 4 dígitos depois do ponto decimal poderia armazenar números como

  12.3456
  57.9331
  00.0001
- Números como 12.345678 seriam arredondados para 12.3456 e 0.0000456 para 0.0000.
- Não é boa para armazenar números pequenos e grandes na mesma aplicação.

- As operações são rápidas, pois são feitas como números inteiros.
- Aplicações comuns são a manipulação de valores monetários e outras em que a precisão numérica necessária é conhecida antecipadamente, como som e gráficos.

Números binários em ponto-fixo

- Usando $k + \ell$ bits, a representação poderá armazenar números no intervalo $[0, 2^k - 2^{-\ell}]$ em incrementos de $2^{-\ell}$.
- Números negativos podem ser representados em qualquer dos métodos disponíveis.

Ponto-flutuante

- Um número em ponto flutuante a pode ser representado por dois números m e e tais que

$$a = m \times b^e.$$

- Nessa representação escolhe-se a base b e a mantissa m é um número com p dígitos (precisão), na forma $\pm d.ddd...ddd$. Cada dígito é um inteiro entre 0 e $b-1$. O expoente é inteiro.

- Permite armazenar números de várias magnitudes diferentes, o que não é possível com ponto fixo.
- Não permite representar números ao mesmo tempo grandes e precisos.
- Por exemplo, um número em ponto flutuante com 4 dígitos decimais ($b = 10, p = 4$) e um expoente no intervalo $[-4, 4]$ pode ser usado para representar 1234, 1.234, ou 0.0001234, mas não teria precisão suficiente para representar 123.456 e 12345.67, que teriam que ser arredondados para 123.5 e 123500.

Binários em ponto-flutuante

- Um número binário em ponto flutuante no padrão ANSI/IEEE tem três componentes: sinal $+/-$, mantissa m e expoente e, representando juntos o valor $\pm m2^e$.
- A mantissa é um número em ponto fixo no formato normalizado:
  - No intervalo $[1, 2)$.
  - Zeros à esquerda são removidos da representação.
  - Como a representação binária da mantissa sempre começa com 1, esse 1 fixo é omitido e somente a parte fracionária é representada explicitamente.
- O expoente é um inteiro com sinal representado no formato excess-k.

- Float e double

	float	long
Largura em bits	32	64
Bits na mantissa	23+1	52+1
Intervalo da mantissa	$[1, 2 - 2^{-23}]$	$[1, 2 - 2^{-52}]$
Bits de expoente	8	11
Excess-k	127	1023

$01000001110001000000000000000000$

Sinal: 0, positivo

Expoente $= 10000011 = 131$
$131 - 127 = 4$

Mantissa $= 1.10001000000000000000000 = 1 + 1/2 + 1/32 = 1.53125$

$01000001110001000000000000000000 = 1.53125 \times 2^4 = 24.5$

- O zero não tem representação apropriada, pois a mantissa é sempre um valor maior que zero.
- Inclui os valores especiais 0, $+\infty$, $-\infty$, NaN, desnormalizado
- Valores especiais são representados fazendo todos os bits do expoente iguais a 0 ou iguais a 1.
- Números desnormalizados aumentam a densidade entre o 0 e o 1, ao incluirem zeros à esquerda.

Arredondamento

- Nem todo fracionário não-periódico é representável.
  - 0.1 é arredondado como 0.0001100110011_2 em precisão simples, $0.099975585937 5_{10}$.
- Há várias formas de arredondamento previstas, incluindo arredondar para o mais próximo, arredondar para o par mais próximo, arredondar ao infinito e outras.

Densidade

- A densidade de números representáveis não é constante.
- Como a mantissa é fixa, é possível representar por exemplo mais números distintos entre 1 e 2 do que entre 1023 e 1024.

Operações em ponto-flutuante

- Soma e subtração envolvem deslocamento para igualar os expoentes, soma das mantissas e normalização.
- Multiplicação e divisão envolvem multiplicação das mantissas, soma dos expoentes e normalização.

Limitações

- Sempre há erros de precisão quando usamos ponto-flutuante que vêm principalmente da incapacidade de representar vários números, de arredondamentos e truncamentos, e do acúmulo de erros ao longo de várias operações.
- Há várias técnicas para lidar com os erros, incluindo representar os números de outras formas (como precisão quádrupla ou decimais), testar relações de igualdade usando intervalos etc. Podem exigir mais tempo de computação e memória.

Em C

Overflow inteiro

- Em C, quando há overflow ou underflow, o número "dá a volta" na representação.

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

int main(int argc, char** argv) {

  int x = INT_MAX;
  int y = 1;
  printf("%d + %d = %d\n",x,y,x+y);

  y = INT_MAX;
  printf("%d + %d = %d\n",x,y,x+y);

  x = INT_MIN;
  y = -1;
  printf("%d + %d = %d\n",x,y,x+y);

  y = INT_MIN;
  printf("%d + %d = %d\n",x,y,x+y);

  return 0;
}
```

exm-over-1.c

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

int main(int argc, char** argv) {

  int x = INT_MAX;
  int y = 1;
  int sum;

  if ((y > 0 && x > INT_MAX - y) || (y < 0 && x < INT_MIN - y)) {
      printf("Overflow.\n");
  }
  else {
    sum = x + y;
  }

  return 0;
}
```

exm-over-2.c

Overflow em ponto-flutuante

- A biblioteca `fenv.h` define constantes e funções para definir e testar o estado de exceções de operações em ponto-flutuante.
- As constantes são:

```
FE_INEXACT
FE_DIVBYZERO
FE_UNDERFLOW
FE_OVERFLOW
FE_INVALID
```

- As funções são:

```
int feclearexcept (int excepts)
int feraiseexcept (int excepts)
int fesetexcept (int excepts)
int fetestexcept (int excepts)
```

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


double compute() {
  double a, b, x;
  scanf("%lf %lf", &a, &b);
  x = a/b;

  return x;
}


int main(int argc, char** argv) {

  feclearexcept(FE_ALL_EXCEPT);

  double f = compute();

  int raised = fetestexcept(FE_OVERFLOW | FE_INVALID | FE_DIVBYZERO);

  if (raised & FE_OVERFLOW)
    printf("Overflow.\n");

  if (raised & FE_INVALID)
    printf("Invalid.\n");

  if (raised & FE_DIVBYZERO)
    printf("Div by 0.\n");

  printf("f: %lf\n",f);

  return 0;
}
```

exm-fenv-1.c

- De man `pow`

```
On success, these functions return the value of x to the power of y.

If x is a finite value less than 0, and y is a finite noninteger,
a domain error occurs, and a NaN is returned.

If the result overflows, a range error occurs, and the functions
return HUGE_VAL, HUGE_VALF, or HUGE_VALL, respectively, with the
mathematically correct sign.

If result underflows, and is not representable, a range error
occurs, and 0.0 is returned.

Except as specified below, if x or y is a NaN, the result is a NaN.
```