C

Guilherme P. Telles

IC

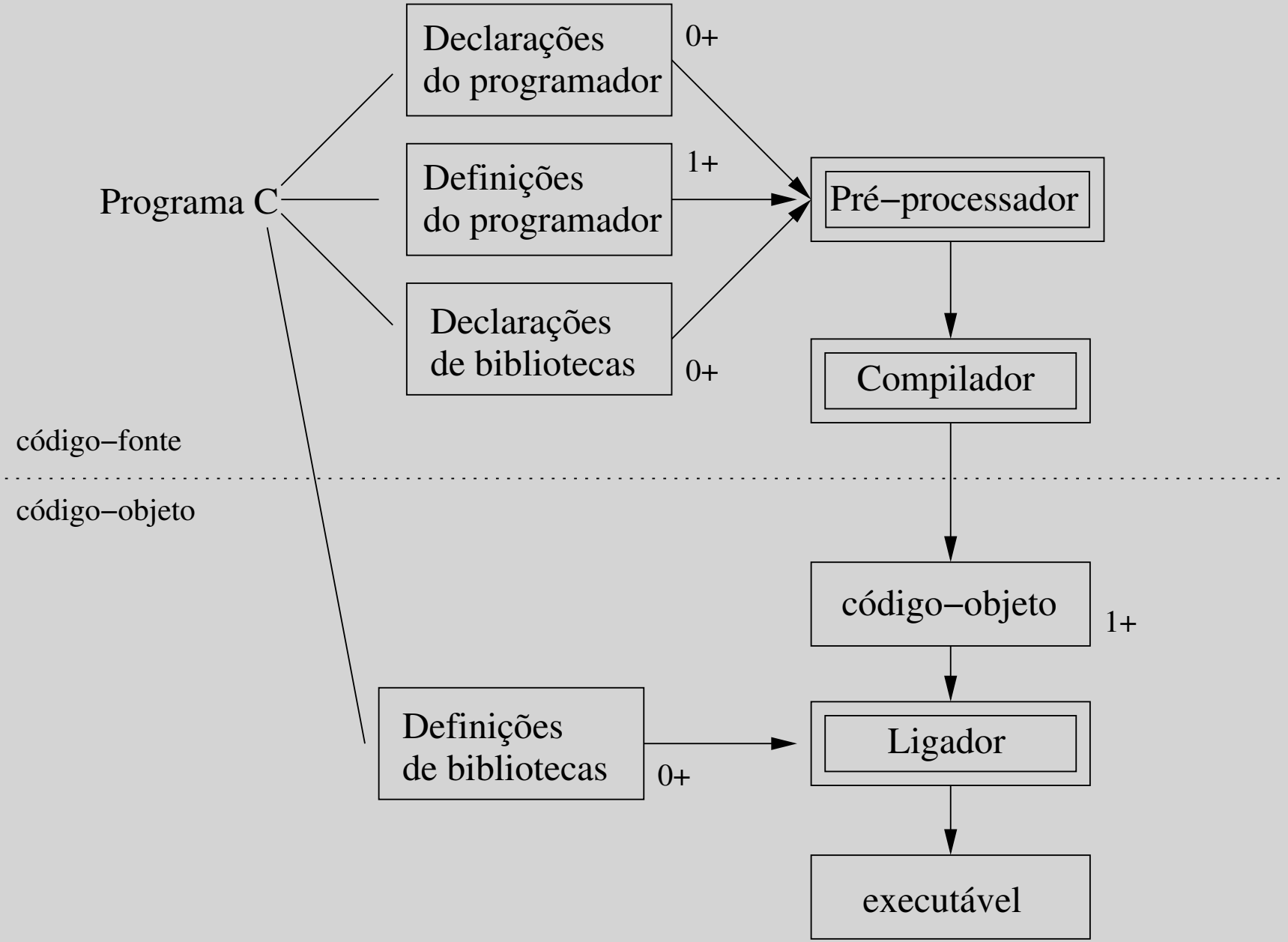
7 de março de 2023

Programa em C

Programa em C

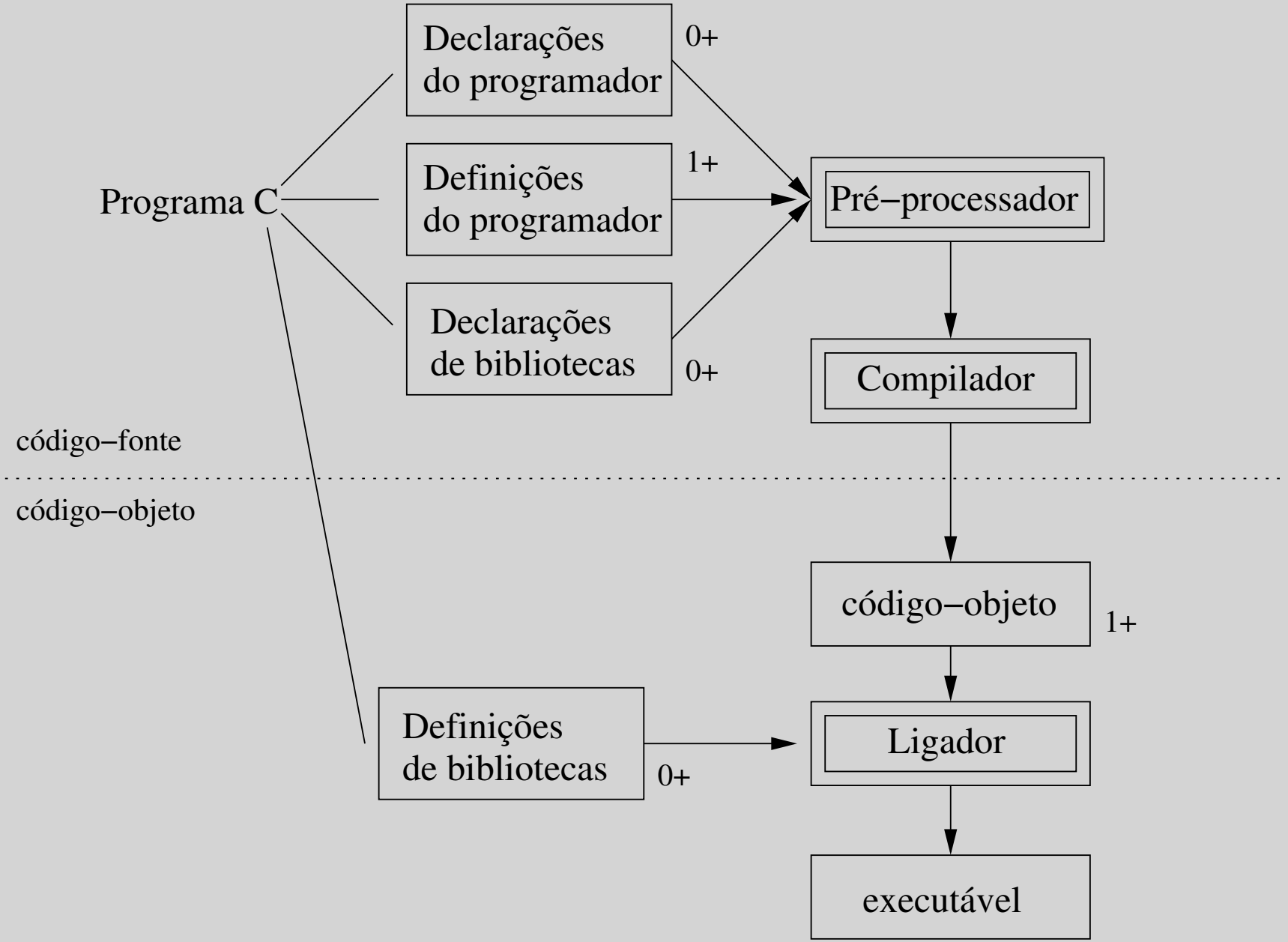
- Um programa em C consiste de um ou mais arquivos.
- Cada arquivo tem zero ou mais definições ou declarações de tipos, variáveis e funções, sendo que uma delas é a função `main()`.
- Os arquivos são processados em conjunto por componentes do sistema de compilação da linguagem para produzir um executável.

- Em C usamos a palavra “definição” para a criação de um nome de variável ou função **e** alocação de memória para a variável ou função.
- Usamos a palavra “declaração” para a criação de um nome de variável ou função associado com memória que já tinha sido alocada antes.


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

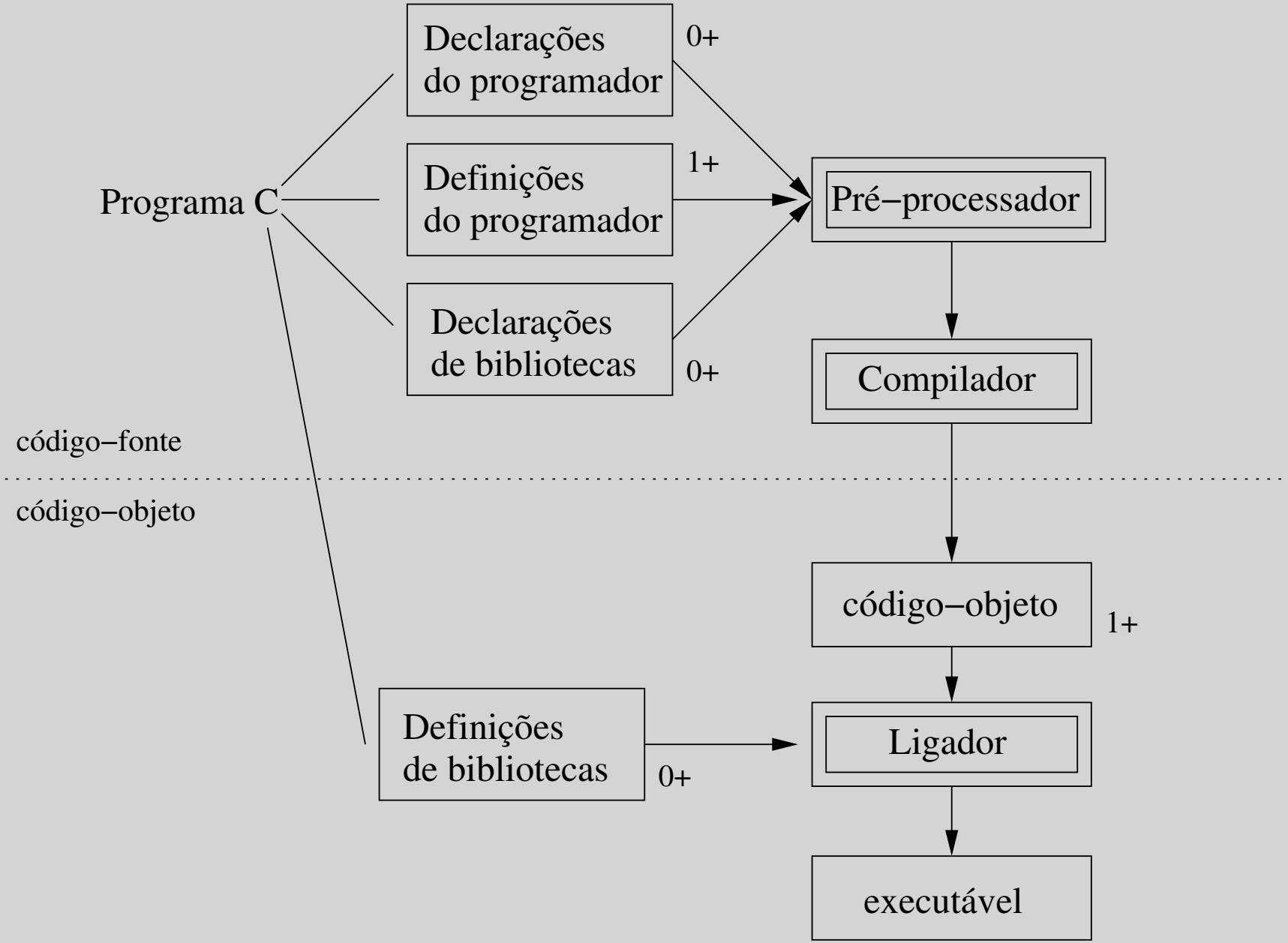
Pré-processador

- 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.
- Resolve trechos de compilação condicional do programa.


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

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.


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

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.

Programar em C

- Para programar em C podemos usar vários tipos de ambientes.
- editor + cc: emacs + GCC, vi + GCC
  editor + make
  editor + automake
  etc.
- IDEs: VS-code, DEV-C++, Eclipse etc.
- Debugging: gdb, ddd, IDEs, valgrind

Sentenças e tokens

Sentença

- Cada função de um programa C é composta de zero ou mais sentenças.
- Uma sentença é uma combinação válida de tokens.
- 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.

Token (átomo)

- 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

- São palavras que têm uso reservado.
- 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.

Nome (ou identificador)

- É uma palavra que dá nome 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 são:
  - `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 fracionárias

```
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."
```

Constantes simbólicas

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

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

```
#define pi 3.1415926

int main() {
  double x = 2 * pi;

  printf("2 x pi: %lf\n",x);
  return 0;
}
```

Operadores

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

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

etc.

Pontuação

- Os símbolos de pontuação são:

  `{ } ( ) ; ,`

- Brancos (espaço, tabulação, fim-de-linha) podem aparecer em qualquer lugar em qualquer quantidade.

Comentários

- `/* */` definem comentários marcados no início e no fim.
  - Podem incluir várias linhas.
  - Não podem ser aninhados.
- `//` define comentário até o fim-de-linha.

Bloco

- Um bloco contém sentenças delimitadas 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.

Variáveis

Definições vs. declarações

- Uma definição de uma variável ou constante associa o tipo ao nome e faz com que o compilador associe uma espaço de memória para ela.

Definições vs. declarações

- Uma definição de uma variável ou constante associa o tipo ao nome e faz com que o compilador associe uma espaço de memória para ela.
- Uma declaração de uma variável ou constante associa o tipo e o nome com um espaço de memória que foi definida em outro arquivo.[1]

[1] Depois de ser pré-processado, chamado de unidade de tradução.

Definição de variáveis

- A definição de variáveis é da forma

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

Definição de variáveis

- A definição de variáveis é da forma

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

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

- Se o qualificador `extern` for usado teremos uma declaração ao invés de uma definição. Mais sobre isso depois.

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.

- Não há uma região exclusiva do programa para definições de variáveis, mas elas devem ser definidas (ou declaradas) antes de serem usadas.

```
int i;
i = 10 * 2;

float x, y;
x = -1.0;
y = 3.14;

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

- Variáveis podem ser inicializadas no momento da definição.

```c
int i = 0;
```

```c
double x = 1.0, y, z = 0.0;
```

```c
char c = 'a';
```

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 se sobrepõe à definição mais externa.

exm-var-1.c

```c
#include <stdio.h>

int main() {

  int i = 1, x = 2;

  if (i == 1) {
    int x = 20;
    int y = 10;
    printf("x %d\n",x);
    printf("y %d\n",y);
  }

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

  return 0;
}
```

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.
  - Dessa forma, variáveis definidas dentro de uma função deixam de existir quando a função termina.[2]

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

Expressões têm valor

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

```
x = 2;
```

  o valor de `x` se torna 2 e o valor da expressão `x = 2` é 2.

- Na sentença

```
z = y = x+1;
```

  O valor da expressão `x+1` é 3.
  O valor de `y` se torna 3, o valor da expressão `y = x+1` é 3.
  O valor de `z` se torna 3, o valor da expressão `z = y = x+1` é 3.

void

- Expressões do tipo void não têm valor e não podem ser usadas.

```
void f() {}

int main(void) {
  foo();
  return 0;
}
```

Tipos Básicos

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

char

- Representa caracteres e é usado para guardar números inteiros pequenos.
- Tem `CHAR_BITS` bits, 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`.

exm-char-1.c

```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;
}
```

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'`).

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.
- Tipicamente o `float` é de 32 bits com mantissa de 6 dígitos decimais e expoente entre -38 e $+38$.
- Tipicamente o `double` é de 64 bits com mantissa de 15 dígitos decimais e expoente entre -308 e $+308$.

Apontador

- Um apontador é uma variável que armazena endereços de memória.

Apontador

- Um apontador é uma variável que armazena endereços de memória.
- Apontadores são declarados com o símbolo $*$ preposto ao nome.

Apontador

- Um apontador é uma variável que armazena endereços de memória.
- Apontadores são declarados com o símbolo $*$ preposto ao nome.
- O tipo de um apontador é "apontador para o tipo apontado por ele".

Apontador

- Um apontador é uma variável que armazena endereços de memória.
- Apontadores são declarados com o símbolo `*` preposto ao nome.
- O tipo de um apontador é "apontador para o tipo apontado por ele".
- Endereços são obtidos com o operador `&`.

exm-apont-1a.c

```c
#include <stdlib.h>

int main() {
  int i = 10;

  int* pi = &i;  // pi armazena o endereco de i na memoria.

  int** ppi = &pi;  // ppi armazena o endereco de pi na memoria.

  return 0;
}
```

- O endereço especial nulo tem valor `0`, com sinônimo `NULL`.

exm-apont-1b.c

```c
#include <stdlib.h>

int main() {
  int i = 10;

  int* pi = &i;  // pi armazena o endereco de i na memoria.

  int** ppi = &pi;  // ppi armazena o endereco de pi na memoria.

  pi = NULL;

  pi = (int*) 15024;  // pi armazena o endereco de memoria 15024.

  return 0;
}
```

Apontadores

- O operador $\star$ é o operador de indireção.

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

- $\star$ e & podem ser consideradas operações inversas.

exm-apont-2.c

```c
#include <stdio.h>

int main() {
  int i = 10;
  int pi = &i;
  int 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;
}
```

Apontadores inválidos

- Para constantes.

```
&1
```

- Para expressões.

```
&(x+2)
```

- Para variáveis `register`.

```
register int r;
&r
```

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.`

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 literais `long` e `unsigned`

- Para especificar o tipo de uma constante literal inteira precisamos acrescentar um sufixo a ela.

tipo	sufixo	exemplo
`unsigned`	`u` ou `U`	`51u`
`long`	`l` ou `L`	`51L`
`unsigned long`	`ul` ou `UL`	`51ul`

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

Constantes literais `float` e `double`

- Constantes fracionárias devem incluir o ponto decimal.
- Para especificar o tipo de uma constante literal fracionária precisamos acrescentar um sufixo a ela.

tipo	sufixo	exemplo
`float`	`f` ou `F`	`3.14f`
`double`	`d` ou `D`	`3.14d`
`long double`	`l` ou `L`	`3.14L`

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

Conversões implícitas de tipos

- Em expressões, `char` e `short` (signed ou unsigned) são sempre convertidos para `int` e se o valor não couber em `int`, para `unsigned int`.
- Por exemplo,

```
char c = 'Z';
printf("%c\n",c);
```

  no printf o tipo da expressão `c` é convertido para `int`.

- Essas conversões são chamadas de **promoções integrais**.

Conversões implícitas de tipos

- Quando há um operador binário e os operandos têm tipos diferentes,
  se um deles é long double então o outro é convertido para long double
  senão se um deles é double então o outro é convertido para double
  senão se um deles é float então o outro é convertido para float

  senão as promoções integrais são aplicadas e,
  se um deles é unsigned long então o outro é convertido para unsigned long

  senão se um deles é long e o outro é unsigned então
  se o long pode representar os valores do unsigned então
  o operando unsigned é convertido para long
  senão
  ambos operandos são convertidos para unsigned long

  senão se um deles é long então o outro é convertido para long
  senão se um deles é unsingned então o outro é convertido para unsigned

  senão ambos são int

- No exemplo abaixo, o valor de `l` é convertido para `float`, que só pode acomodar 6 dígitos decimais:

exm-conv-1.c

```c
#include <stdio.h>

int main(void) {

  long l = 1234567890;
  float f = 3.1416;

  printf("l+f %.15lf\n",l+f);

  printf("sizeof l+f %d\n",sizeof(l+f));

  return 0;
}
```

Conversão explícita (casting)

- Determina que uma expressão de um tipo deve ser convertida para outro tipo.
- O operador unário `()` é o operador de casting. A precedência é maior que a dos operadores aritméticos.

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

- No exemplo abaixo, o valor de `f` é convertido para `double`:

exm-conv-2.c

```c
#include <stdio.h>

int main(void) {

  long l = 1234567890;
  float f = 3.1416;

  printf("l+f %.15lf\n",l+(double)f);

  printf("sizeof l+f %d\n",sizeof(l+(double)f));

  return 0;
}
```

- O efeito de conversões que aumentam o tamanho da representação do valor da expressão (promoção) geralmente são bem comportados.
- O efeito de conversões que reduzem o tamanho da representação do valor da expressão (demoção) e ele "não cabe" na nova representação depende do sistema.

  P.ex., de um fracionário para um inteiro as casas decimais são perdidas, e se a parte inteira não cabe no tipo o corportamento depende do sistema.
- Casting permite a conversão do tipo de uma expressão, nunca de uma variável.

typedef

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

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