Linguagens de Programação

Sistemas de Linguagens

Andrei Rimsa Álvares
andrei@cefetmg.br

Sumário

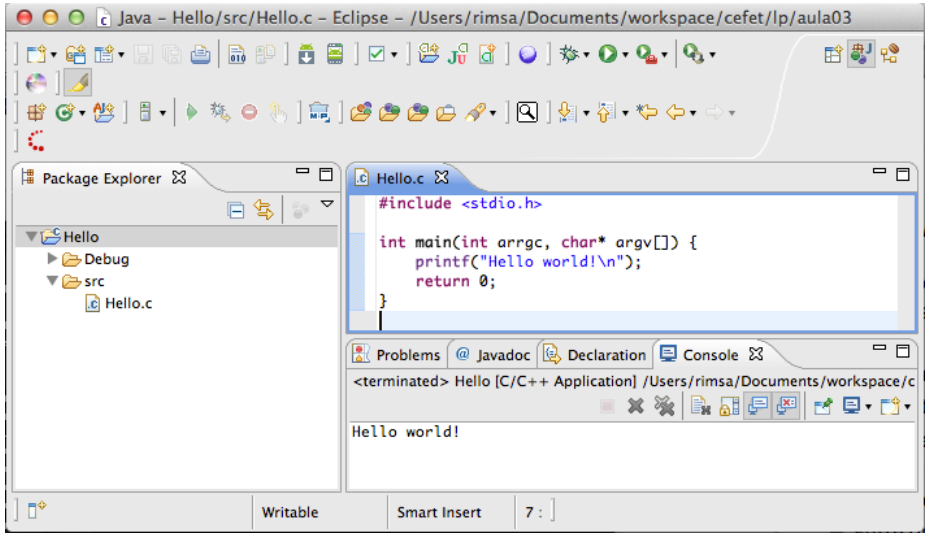
- Introdução
- A sequência clássica
- Sobre otimizações
- Variações da sequência clássica
- Depuradores (*debuggers*)
- Suporte em tempo de execução
- Amarrações

INTRODUÇÃO

Linguagens de Programação

Ambiente de Desenvolvimento Integrado

- Ambiente de desenvolvimento integrado (*IDE: Integrated Development Environment*) é um **programa de computador** que reúne características e ferramentas de apoio ao desenvolvimento de *software* com o objetivo de agilizar esse processo



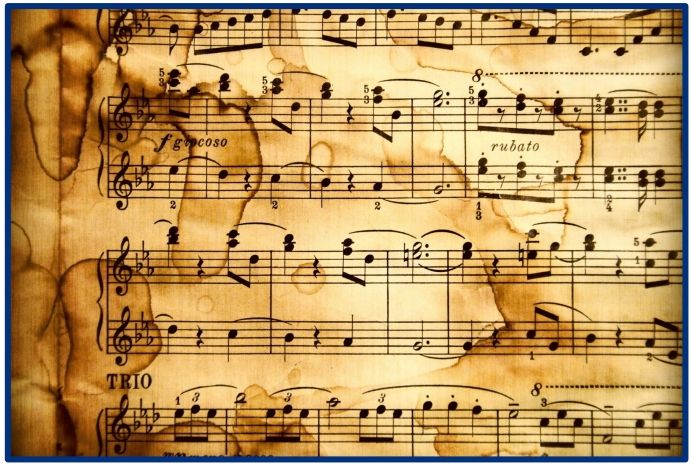
Eclipse IDE

Consiste normalmente de

- Um editor de código-fonte
- Ferramentas para construção automática
- Depurador
- Dentre outros...

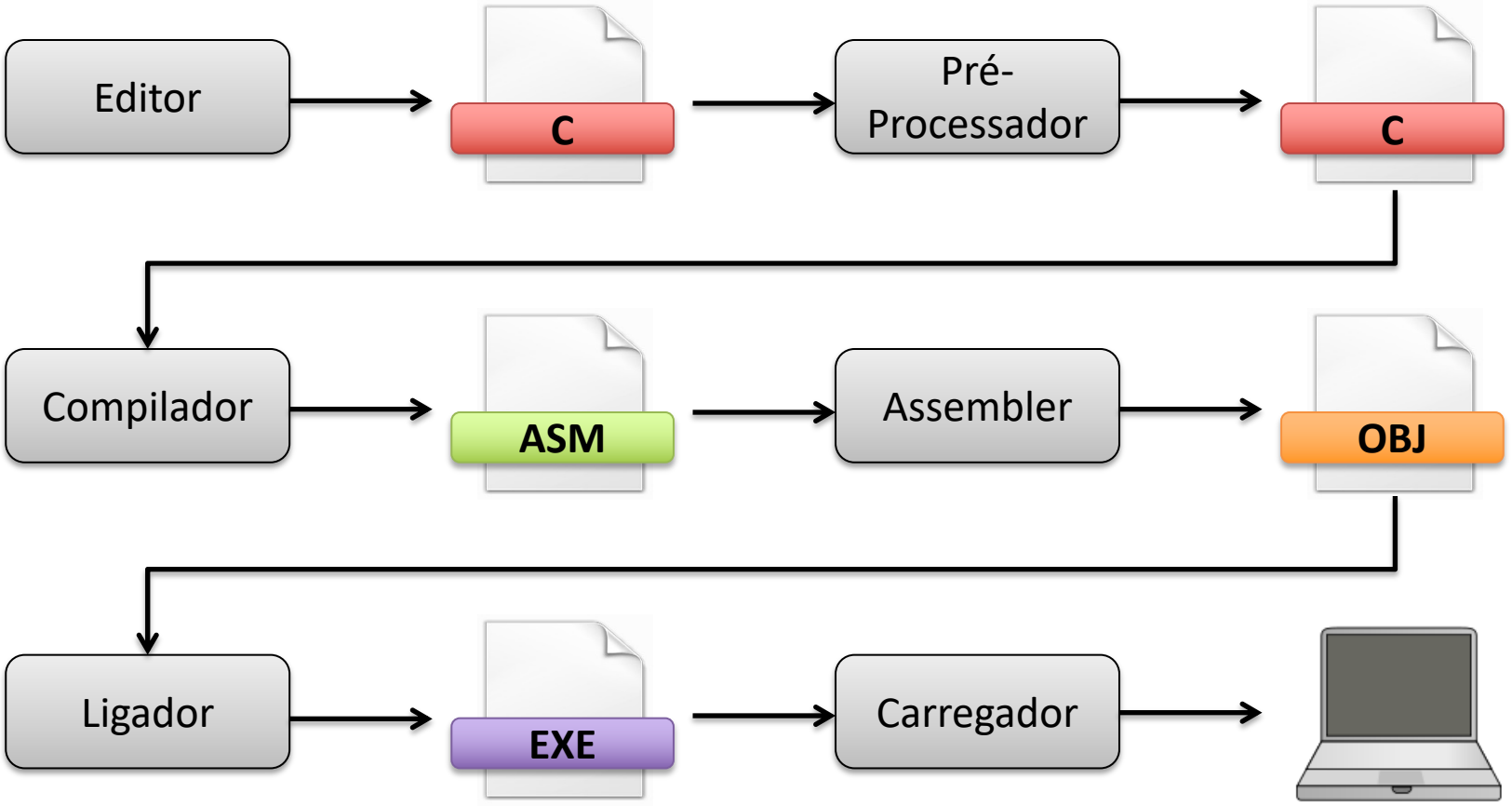
Quais são os passos envolvidos quando se pressiona o botão de executar?

A SEQUÊNCIA CLÁSSICA



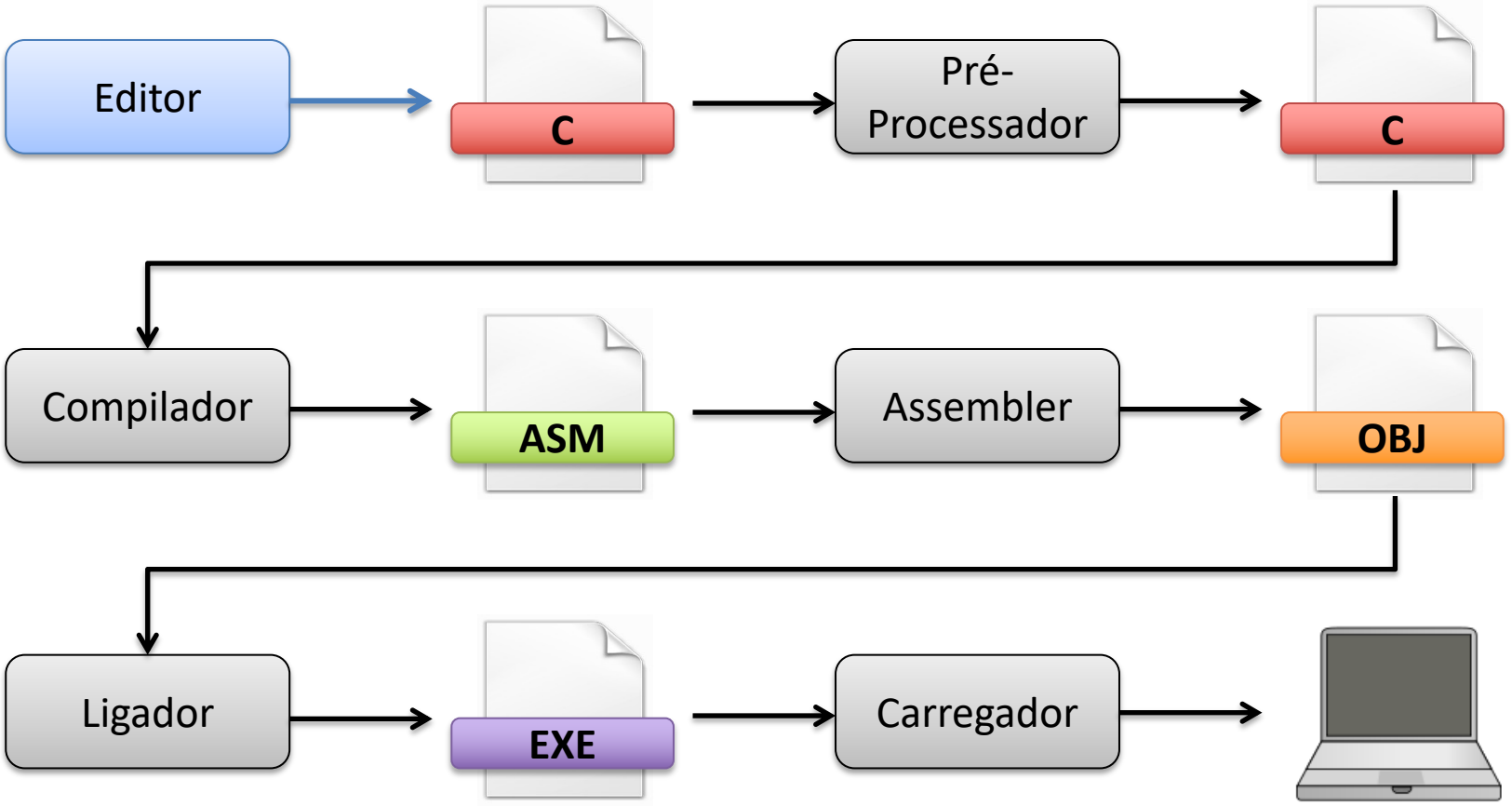
A Sequência Clássica

- O diagrama a seguir mostra a sequência clássica de passos envolvendo a execução de um programa (detalhes podem variar)



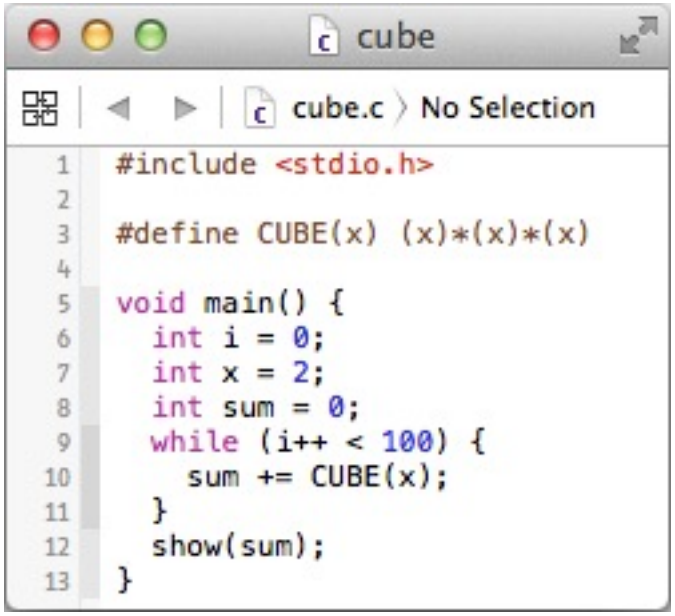
Editor

- O diagrama a seguir mostra a sequência clássica de passos envolvendo a execução de um programa (detalhes podem variar)



Editor

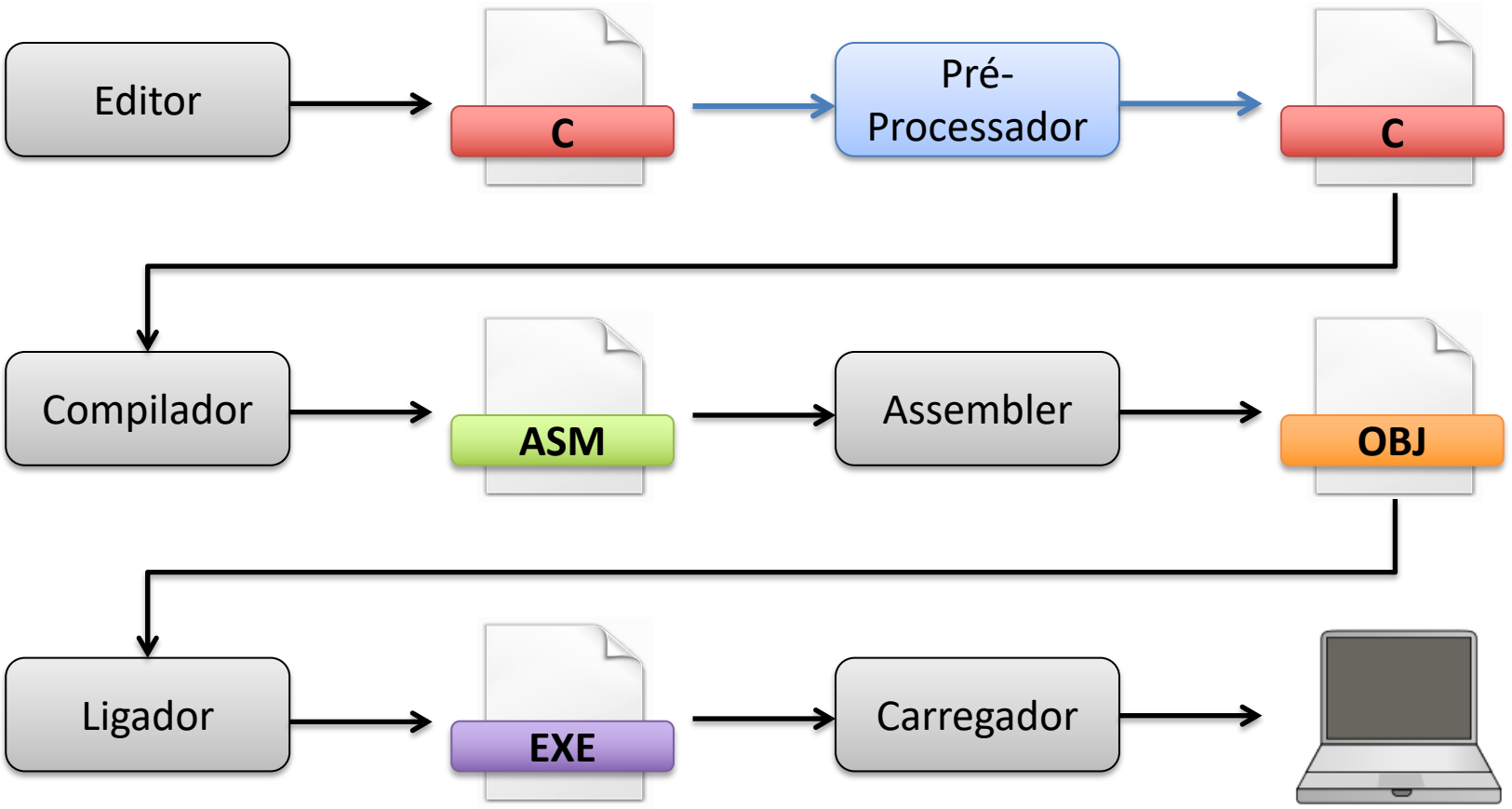
- O programador usa um **editor de texto** para criar um arquivo textual que contém o programa
- O programa é escrito em uma **linguagem de alto nível** independente de máquina
- Exemplo: programa que calcula `100` vezes o cubo de `x`



`cube.c`

Pré-Processador

- O diagrama a seguir mostra a sequência clássica de passos envolvendo a execução de um programa (detalhes podem variar)



Pré-Processador

- O pré-processamento envolve a **transformação** de um programa de entrada em um outro programa de saída semelhante
- O tipo de pré-processamento depende de sua natureza
  - Alguns são capazes apenas de substituições simples de texto e expansão de macros; enquanto outros são tão poderosos quanto LPs
- Exemplo

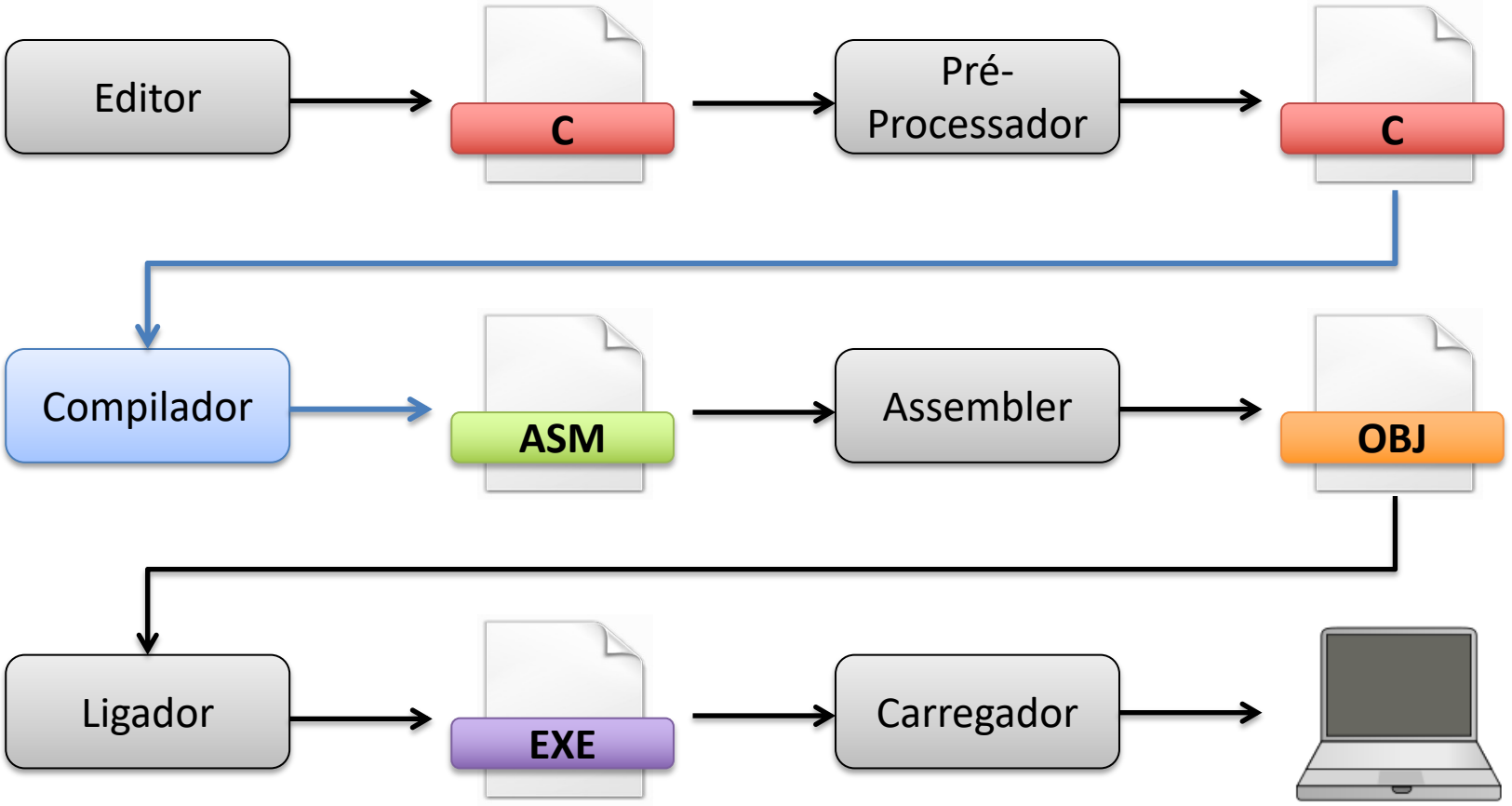
```
gcc -E \
    -o cube_pre.c \
    cube.c
```

cube_pre.c

```
405 extern int __vsnprintf_chk (char * restrict, size_t, int, size_t,
406         const char * restrict, va_list);
407 # 491 "/usr/include/stdio.h" 2 3 4
408 # 2 "cube.c" 2
409
410
411
412 void main() {
413   int i = 0;
414   int x = 2;
415   int sum = 0;
416   while (i++ < 100) {
417     sum += x * x * x;
418   }
419   show(sum);
420 }
```

Compilador

- O diagrama a seguir mostra a sequência clássica de passos envolvendo a execução de um programa (detalhes podem variar)



Compilador

- O compilador transforma o arquivo-fonte para linguagem *assembly*
- O código gerado é dependente de máquina (x86, ARM, MIPS, ...)
- Cada linha representa um pedaço de dados, ou uma única instrução em nível de máquina
- Antes de Fortran (1957), programas eram escritos em *assembly*; atualmente, somente em situações específicas (eficiência, ...)

Compilador

- Exemplo

```
gcc –S –o cube.S \
  cube_pre.c
```

Este exemplo utiliza a sintaxe **AT&T** para x86

Como seria o *assembly* para um outro tipo de processador, como ARM por exemplo?

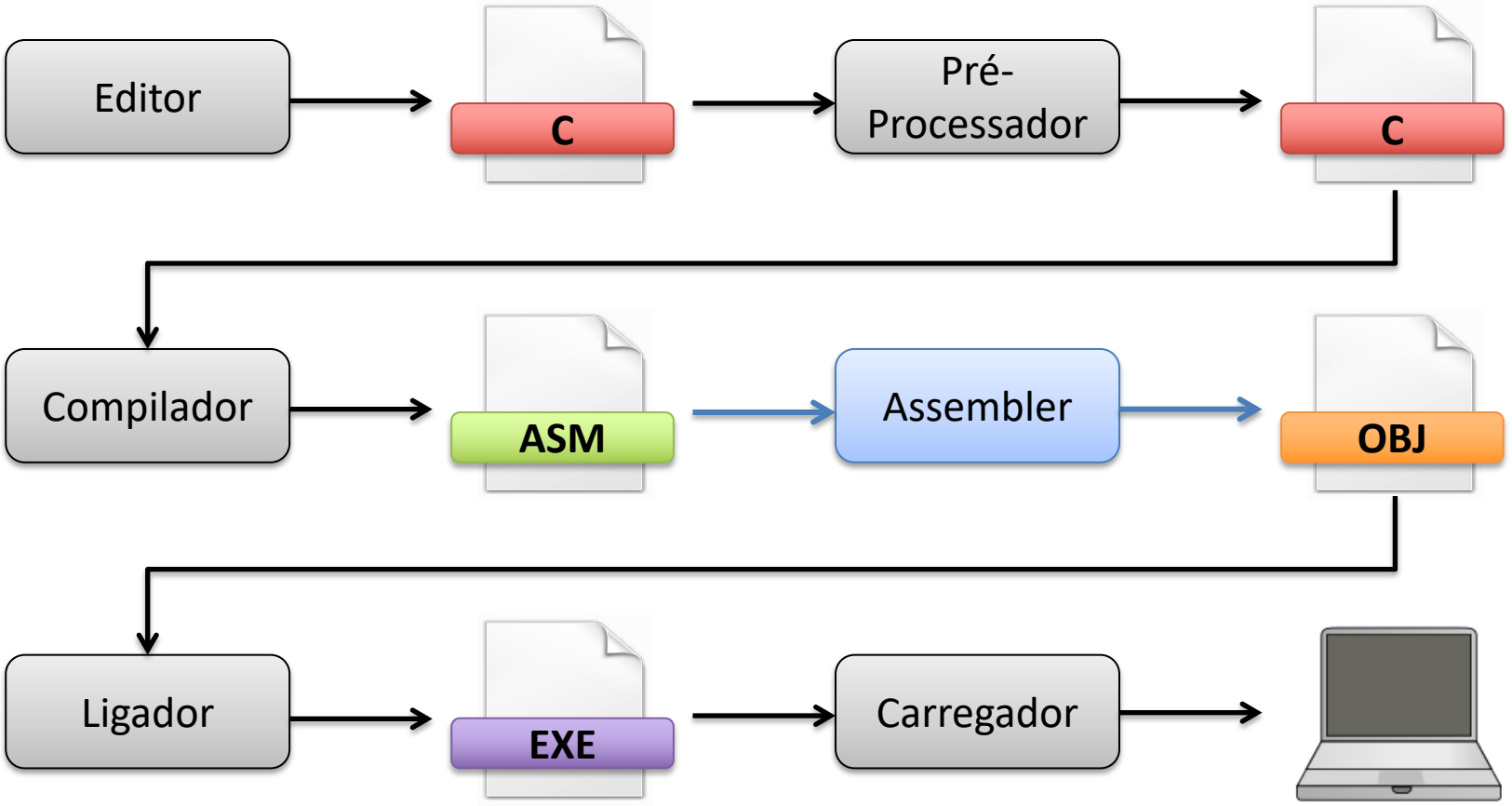
cube.S

```
        .section        __TEXT,__text,regular,pure_instructions
        .globl  _main
        .align  4, 0x90
_main:
        .cfi_startproc
## BB#0:
        pushq   %rbp
Ltmp2:
        .cfi_def_cfa_offset 16
Ltmp3:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Ltmp4:
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        movl    $0, -4(%rbp)
        movl    $0, -8(%rbp)
        movl    $2, -12(%rbp)
        movl    $0, -16(%rbp)
LBB0_1:
        movl    -8(%rbp), %eax
        movl    %eax, %ecx
        addl    $1, %ecx
        movl    %ecx, -8(%rbp)
        cmpl    $100, %eax
        jge     LBB0_3
## BB#2:
        movl    -12(%rbp), %eax
        imull   -12(%rbp), %eax
        imull   -12(%rbp), %eax
        movl    -16(%rbp), %ecx
        addl    %eax, %ecx
        movl    %ecx, -16(%rbp)
        jmp     LBB0_1
LBB0_3:
        movl    -16(%rbp), %edi
        callq   _show
        movl    $0, %eax
        addq    $16, %rsp
        popq    %rbp
        retq
        .cfi_endproc


.subsections_via_symbols
```

Assembler

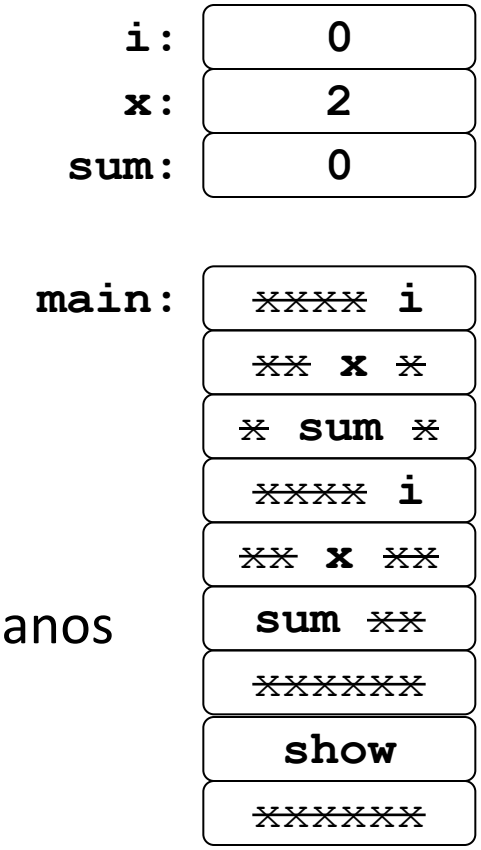
- O diagrama a seguir mostra a sequência clássica de passos envolvendo a execução de um programa (detalhes podem variar)



Assembler

- Linguagem *assembly* ainda não é executável
  - Em formato de texto, legível por humanos
  - Possui nomes, não endereços de memória
- O *assembler* converte cada instrução em linguagem *assembly* para o formato binário da máquina - **linguagem de máquina**
- O arquivo **objeto** resultante não é legível por humanos
- Exemplo

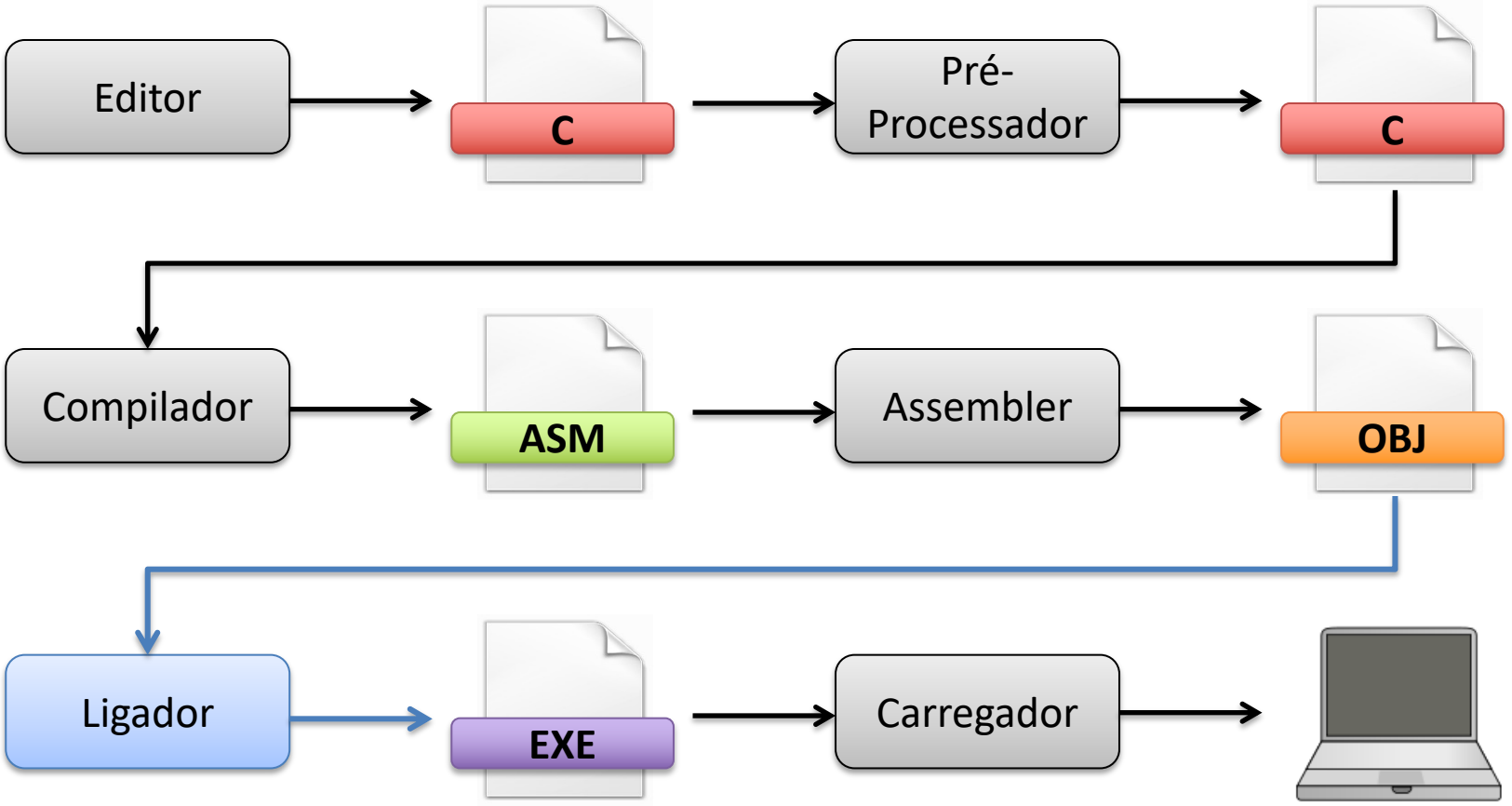
```
as -o cube.o cube.S
gcc -c -o cube.o cube.S
```



Onde está o código do `show`?

Ligador

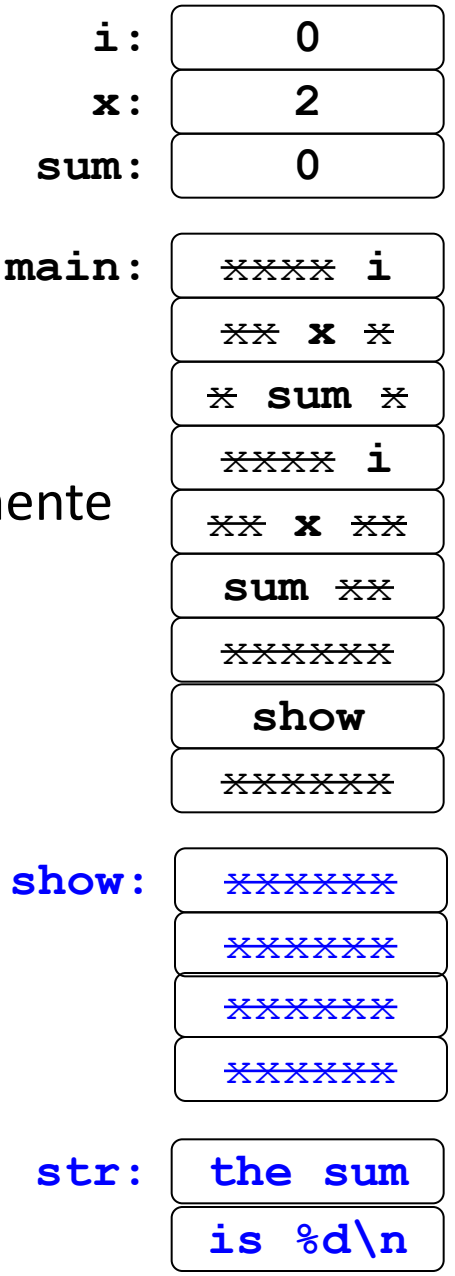
- O diagrama a seguir mostra a sequência clássica de passos envolvendo a execução de um programa (detalhes podem variar)



Ligador

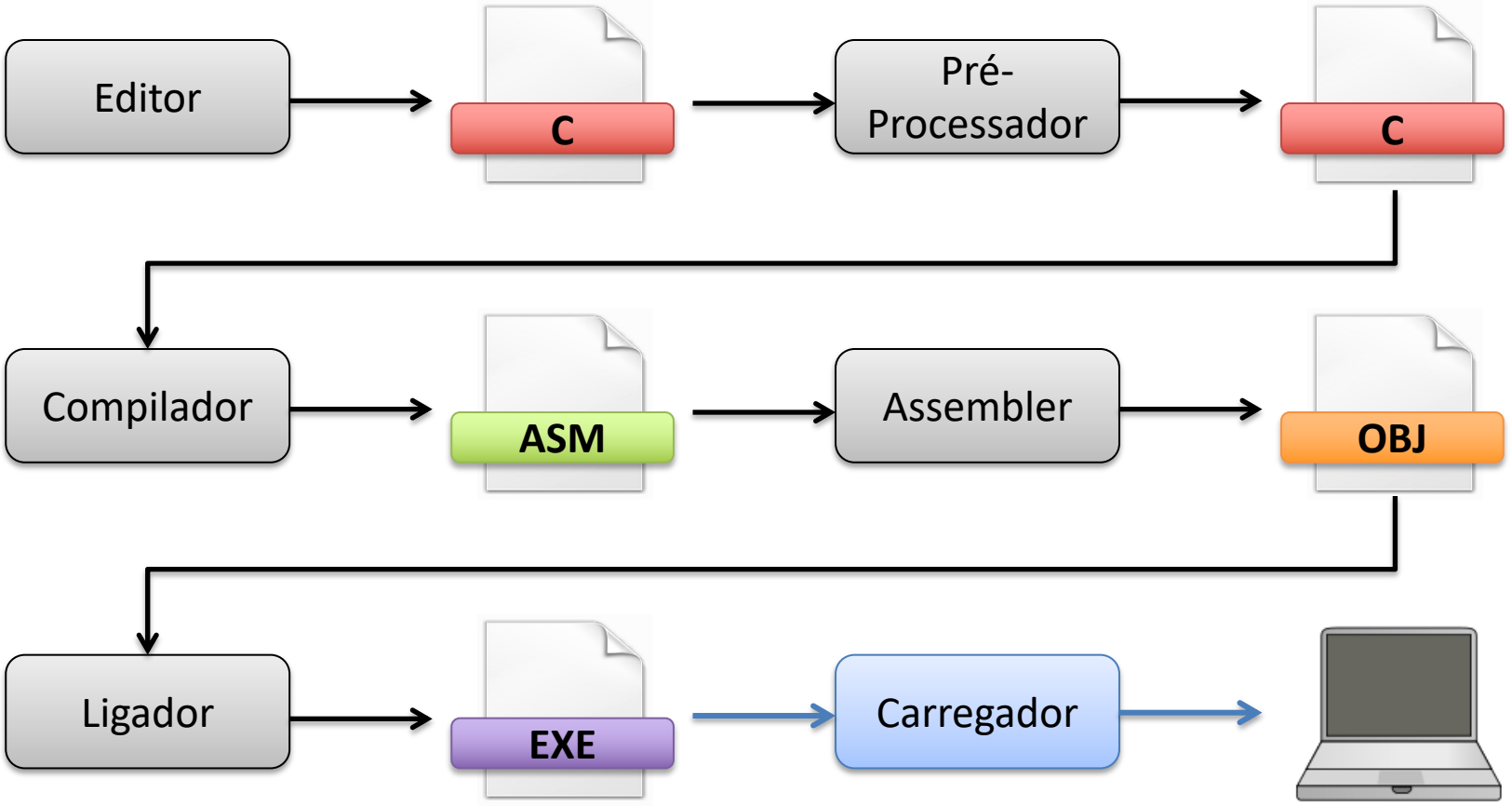
- Arquivos objetos não são executáveis
  - Ainda faltam códigos
  - Ainda têm alguns nomes
  - Maioria código de máquina, mas não completamente
- **Ligador** coleta e combina todas as partes diferentes
  - Por exemplo, o código do **show** foi compilado em outro lugar e pode ter sido até feito em outra linguagem de programação
- Resultado é o arquivo executável
- Exemplo

```
ld –o cube.exe cube.o show.o ...
gcc –o cube.exe cube.o show.o
```



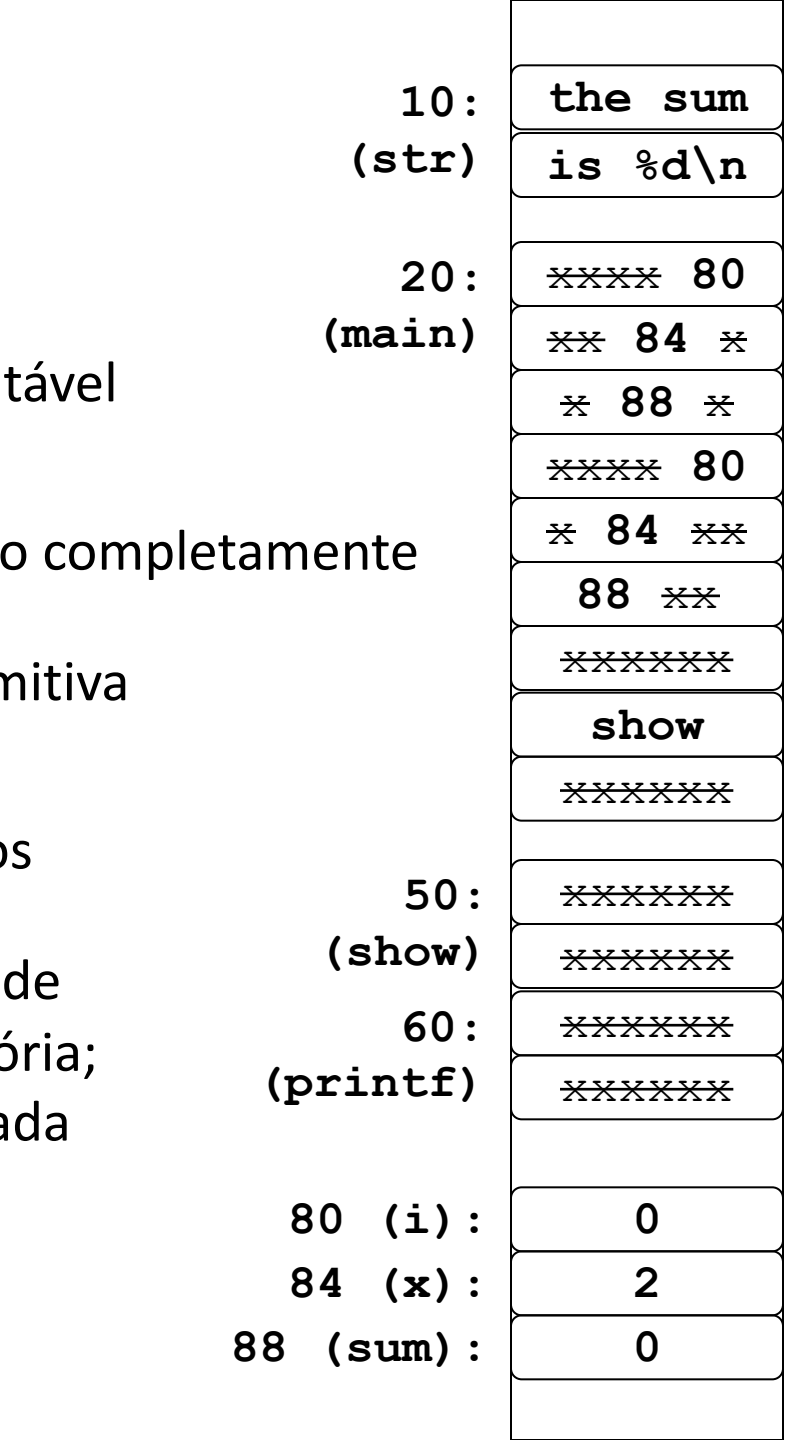
Carregador

- O diagrama a seguir mostra a sequência clássica de passos envolvendo a execução de um programa (detalhes podem variar)



Carregador

- Arquivo "executável" ainda não é executável
  - Ainda tem alguns nomes
  - Maioria código de máquina, mas não completamente
- Exemplo: assumindo uma memória primitiva
  - Organizada como *array* de bytes
  - Índice de cada byte são os endereços
- Antes de ser executado, não se sabe onde o programa será carregado nessa memória; o **carregador** acha um endereço para cada pedaço e substitui por endereços



Executando

- Após carregado, o programa é inteiramente linguagem de máquina
  - Todos os nomes foram substituídos por endereços de memória
- Processador começa a execução de suas instruções fazendo com que o programa rode

SOBRE OTIMIZAÇÕES

Sobre Otimizações

- Código gerado por um compilador é geralmente otimizado para fazê-lo mais rápido, menor ou ambos
- Outras otimizações podem ser feitas pelo **assembler**, **ligador** e/ou **carregador**
- **Equívoco comum:** o código resultante é melhor, mas não é garantido que será ótimo

Exemplo

- Código original vs código otimizado

```
int i = 0;
while (i < 100) {
   a[i++] = x*x*x;
}
```


```
int i = 0;
int temp = x*x*x;
while (i < 100) {
   a[i++] = temp;
}
```

Você sabe o nome desta otimização?

Exemplo

- Remoção de **invariante de laço** é lidado pela maioria dos compiladores modernos
- Ou seja, compiladores geram o mesmo código eficiente para os dois casos anteriores
- Portanto, é perda de tempo para o programador fazer essa otimização manualmente

Outras Otimizações

- Algumas otimizações que transformam o código em uma representação intermediária podem adicionar novas variáveis
- Outras otimizações removem variáveis, removem códigos, adicionam códigos, movem códigos de um lado para o outro
- Uma pergunta simples, como a descrita a seguir, pode ter respostas complicadas

> “Qual código em linguagem *assembly* foi gerado para esse comando?”

VARIAÇÕES DA SEQUÊNCIA CLÁSSICA

Linguagens de Programação

Variações da Sequência Clássica

- Algumas variações da sequência clássica

1. Escondendo os passos
2. Ambientes de desenvolvimento integrado
3. Interpretadores
4. Máquinas virtuais
5. Ligação tardia
6. Instrumentação de código (perfilamento)
7. Compilação dinâmica (JIT)

1. Escondendo os Passos

- Muitos sistemas de computação permitem fazer a parte de compilar-*assemblar*-ligar com apenas um comando
  - Exemplo: compilador gcc em um sistema UNIX

```
$ gcc cube.c
```

Compilar, *assemblar* e ligar

```
$ gcc cube.c –S
$ as cube.S –o cube.o
$ ld –o cube.exe cube.o ...
```

Compilar, então *assemblar* e depois ligar

- A maioria dos compiladores modernos incorporam todas as funcionalidades de um *assembler*, gerando código-alvo diretamente

2. Ambientes de Desenvolvimento Integrado

- Uma interface única para editar, executar e depurar programas

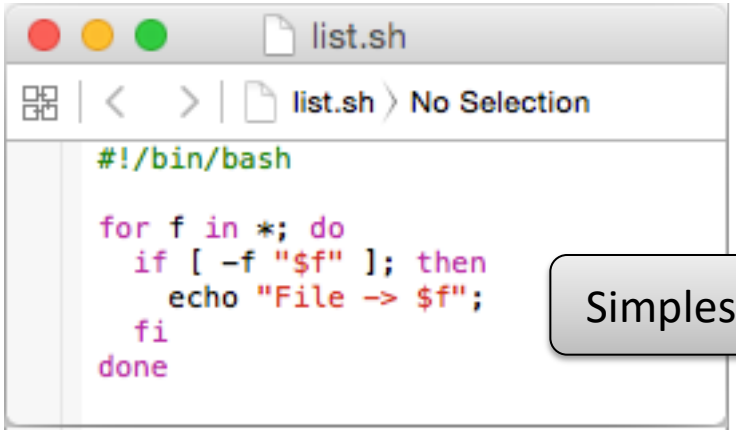
- Essa integração pode adicionar poder em cada um dos passos:
  - Editores conhecem a sintaxe da linguagem
  - Sistemas podem manter um banco de dados de código-fonte (não arquivos texto individuais) e códigos objetos
  - Sistemas podem manter diferentes versões, coordenar colaboração com outras pessoas no projeto
  - Reconstrução após mudanças incrementais podem ser coordenadas
  - Depuração é facilitada

3. Interpretadores

- **Interpretar** um programa é executar os passos especificados, sem primeiramente transformar em uma linguagem de baixo nível

- Interpretadores são usualmente muito mais **lentos**
  - Compilação demora mais tempo no começo, mas depois programas rodam na velocidade do hardware
  - Interpretação começa imediatamente, mas cada passo deve ser processado em software

- Exemplo: listar todos os arquivos do diretório atual



Simples distinção?

4. Máquinas Virtuais

- Um sistema de computação pode produzir código para uma linguagem de máquina para qual não existe hardware; em **código intermediário**

- Máquinas virtuais devem ser simuladas em software – ou seja, interpretadas em software

- Sistemas de computação podem fazer toda a sequência clássica, mas depois devem interpretar o programa em código intermediário

Qual a vantagem de se fazer isso?

4. Máquinas Virtuais

- Por que usar máquinas virtuais?

  - Portabilidade (Execução multi-plataforma)
    - Máquinas virtuais podem ser implementadas em software para diferentes plataformas
    - Simular máquinas físicas é mais difícil

  - Segurança reforçada
    - O programa em execução nunca está diretamente com o controle do processador
    - Interpretadores podem intervir se o programa tentar fazer algo que não deveria

4. Máquinas Virtuais

- Exemplo de máquina virtual

Alguma outra máquina virtual?

  - O sistema de computação de **Java** usualmente compila código para a máquina virtual **JVM** (*Java Virtual Machine*)
  - O código intermediário é conhecido como **bytecode**
  - Praticamente todos os navegadores modernos possuem um interpretador de bytecode
    - Quando se visita uma página com Java *applet*, o navegador executa a *applet* interpretando seu bytecode

5. Ligação Tardia

- O passo de ligação pode ser feito tardiamente
- Código para funções de bibliotecas não é incluído no programa executável do programa chamado
- Vantagens de ligação tardia
  - Múltiplos programas compartilham uma cópia de funções de biblioteca: uma cópia em disco e outra em memória
  - Bibliotecas de funções podem ser atualizadas independentemente dos programas: todos os programas usam a nova biblioteca na próxima vez que forem executados
  - Podem evitar de carregar códigos que não são executados

5. Ligação Tardia

- Ligação tardia no **Windows**
  - Bibliotecas de funções para ligação tardia são mantidas em arquivos **.dll** (*dynamic link library*)
  - Existem em duas formas
    1) Ligação dinâmica em **tempo de carga**: carregador encontra os arquivos DLL (que já podem estar em memória), e liga o programa a funções necessárias antes de executar
    2) Ligação dinâmica em **tempo de execução**: programas em execução podem fazer chamadas de sistema explícitas para encontrar DLLs e carregar funções específicas

5. Ligação Tardia

- Ligação tardia no **Linux**
  - Bibliotecas de funções para ligação tardia são mantidas em arquivos **.so** (*shared object*) seguidos de um sufixo
  - Também em duas formas
    1) **Bibliotecas compartilhadas:** carregador liga o programa a funções necessárias antes de executar
    2) **Bibliotecas carregadas dinamicamente:** programas em execução podem fazer chamadas de sistema explícitas para encontrar bibliotecas e carregar funções específicas

5. Ligação Tardia

- Ligação tardia em **Java**
  - A máquina virtual Java (JVM) automaticamente carrega e liga classes quando um programa as usa
  - O carregador de classes (*Classloader*) tem muito serviço
    - Pode carregar uma classe em um local remoto – por exemplo pela internet
    - Verifica minuciosamente se o código carregado é compatível com as exigências da máquina virtual

6. Instrumentação de Código (*Perfilamento*)

- A instrumentação de código (*perfilamento*) permite monitorar a execução de um programa

- Instrumentação é usada para
  - Depurar o desempenho de programas
    - Ex.: encontrar trechos de código quentes (mais executados), ...
  - Avaliar o comportamento de programas
    - Ex.: verificar vazamentos de memória, ...
  - Entre outros

6. Instrumentação de Código (*Perfilamento*)

- A instrumentação de código pode ser
  - **Estática**: o programa é recompilado adicionando-se instruções para a instrumentação (o código-fonte é necessário)
    - Exemplo: gcc com gprof
  - **Dinâmica**: o programa é simulado em um ambiente de execução controlado, seu comportamento é observado enquanto é executado (somente o código compilado é necessário)
    - Exemplos: valgrind, PIN, DynamoRIO, qemu, ...

6. Instrumentação de Código (*Perfilamento*)

- Uma outra aplicação de instrumentação de código: otimizações orientadas a perfil (*PGO: profile guided optimizations*)

- A sequência clássica é executada duas vezes

  - **Na primeira vez:** o compilador adiciona código para gerar estatísticas sobre a execução do programa
    - Ex: `gcc –fprofile-generate ...`
  - **Na segunda vez:** as informações obtidas em tempo de execução são realimentadas no processo de compilação para auxiliar na geração de códigos melhores
    - Ex: `gcc –fprofile-use ...`

7. Compilação Dinâmica (JIT)

- Algumas compilações podem ser realizadas **depois** que o programa começa sua execução

- Muitas variações
  - Compila cada função apenas quando é chamada
  - Começa com interpretação e compila apenas aqueles pedaços que são chamados frequentemente
  - Compila grosseiramente inicialmente (para código intermediário, por exemplo); gasta mais tempo em pedaços de códigos executados frequentemente (compilar para código nativo e otimizar, por exemplo)

- Conhecida como compilação Just-in-time (JIT)

DEPURADORES (*DEBUGGERS*)

Linguagens de Programação

Depuradores

- Algumas capacidades de depuradores (*debuggers*)
  - Examinar um programa *on-the-fly*
    - Passo a passo, breakpoints, modificar variáveis
  - Examinar um *snapshot*, como um *core dump*
  - Modificar programas já em execução
    - Recompilar, religar, recarregar partes do programa enquanto o mesmo está em execução

Algumas funcionalidades de depuração exigem uma IDE

Informações de Depuração

- Algumas perguntas:
  - Em que ponto do programa está a execução?
  - Qual é a pilha de chamada que levou a esse ponto?
  - Quais são os valores das variáveis na memória?

- Informações do nível do código-fonte podem ser encontradas em código em nível de máquina (desde que explicitamente adicionados pelo compilador)
  - Nomes de variáveis e funções
  - Localização de códigos pela posição no código-fonte

Isso pode ser difícil de manter, por causa de otimizações

SUPORTE EM TEMPO DE EXECUÇÃO

Linguagens de Programação

Suporte em Tempo de Execução

- O ligador pode adicionar código adicional mesmo que o programa original não os referencie explicitamente
  - **Processamento de inicialização:** iniciar o estado da máquina
  - **Tratamento de exceções:** reagir a exceções causadas na execução
  - **Gerenciamento de memória:** alocação de memória, reutilização quando o programa terminou
  - **Interface com o S.O.:** comunicação entre o programa em execução e operações de entrada e saída do S.O., por exemplo
- É um importante atuador oculto no sistema de computação

AMARRAÇÕES

Linguagens de Programação

Conceituação

- **Amarração** (*binding*) é uma associação entre entidades de programação, como
  - Variável/Valor
  - Identificador/Tipo
  - Operador/Símbolo

- O tempo no qual ocorre a amarração é chamado de **tempo de amarração**

- Enfoque maior na amarração de identificadores a entidades

Tempos de Amarração

- As construções de uma linguagem podem ter os seguintes tempos de amarração

1) Tempo de projeto da LP
2) Tempo de implementação
3) Tempo de compilação
4) Tempo de ligação
5) Tempo de carga
6) Tempo de execução

Por que é importante saber estes tempos de amarração?

1) Tempo de Projeto da LP

- Definição dos símbolos da linguagem e seu comportamento
  - Ex: `*` (multiplicação)

- Definição das palavras reservadas e seu comportamento
  - Ex.: `true`, `if`, `while`

2) Tempo de Implementação

- Implementada durante a codificação do tradutor (em geral nos compiladores)

- Em C, a definição da faixa de valores do tipo é feita em tempo de implementação

Qual o tamanho do tipo `int` em C para uma arquitetura de 16 bits? E de 32 bits?

3) Tempo de Compilação

- Associação da variável com seu tipo

```
int i;
```

```
char c;
```

- Associação de expressões com seus operadores

```
3 * 2
```

```
(x >> 4) & 0xF
```

4) Tempo de Ligação

- Integração (ligação) de vários módulos compilados previamente

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

int main(int argc, char* argv[]) {
  int n = atoi(argv[1]);
  printf("%f\n", sqrt(n));
  return 0;
}
```

Onde está a implementação da função `sqrt`?

Na biblioteca de matemática (`libm.a`) que pode ser ligada usando a flag de compilação `-lm`

5) Tempo de Carga

- Durante o carregamento do programa
  - Memórias associadas a variáveis globais e constantes
  - Endereços de memória relativos substituídos por endereços de memória absolutos

6) Tempo de Execução

- Ocorridas em tempo de execução

- Exemplo
  - Associação de valor a variável

```
int x = 5;
```

  - Associação de áreas de memória a variáveis

```
int x = new int;
```

Exercício

- Considere o seguinte trecho de código em C para preencher a tabela

```
void f() {
   int cont;
   /* ... */
   cont = cont + 5;
   /* ... */
}
```

Exemplo	Tempo de Amarração
Possíveis tipos de *cont*	
Tipo de *cont*	
Possíveis valores de *cont*	
Valor de *cont*	
Possíveis significados do operador +	
Significado do operador + nesta atribuição	
Representação interna do literal 5	

Amarrações

- O momento no qual ocorre a ligação pode ser classificado como **cedo** (*early binding*) ou **tardio** (*late binding*)

- Quanto mais cedo ocorre a ligação, maior a eficiência de execução do programa, mas menor a flexibilidade das estruturas disponibilizadas

Amarração de Tipo

- O tipo deve ser amarrado à variável para que esta seja referenciada pelos programas

- Aspectos importantes
  - Como o tipo deve ser definido?
  - Quando a amarração deve ser feita?

- Amarrações de tipos

Amarração Estática	Amarração Dinâmica
• Antes da execução • Permanece inalterada na execução	• Durante a execução • Pode ser criada/alterada na execução

Amarração de Tipo Estática

- Pode ser obtida através de
  - **Declaração explícita**: instrução do programa que lista nomes de variáveis e especifica os tipos
    - Ex. em C: `int i, char c;`
  - **Declaração implícita**: convenções determinam seu tipo
    - Ex. em Fortran: se começam com I, J, K, L, M, ou N: `INTEGER`; outras: `REAL`
    - Ex. em Perl: se começam com `$`: escalar; `@`: arranjo (*array*); `%`: dicionário (*hashtable*)

Amarração de Tipo Dinâmica

- A variável é vinculada a um tipo quando lhe é atribuído um valor em uma instrução de atribuição
- A variável que está sendo atribuída é vinculada ao tipo do valor, variável ou expressão do lado direito da atribuição
- Vantagens
  - Flexibilidade e generalidade
- Desvantagens
  - Redução da capacidade de detecção de erro pelo compilador
  - Custo de implementação da vinculação dinâmica

Amarração de Tipo Dinâmica

- Exemplo de alteração de tipo dinamicamente em C#

```
dynamic variavel = new Int32();
variavel = 10;   // Int32
variavel = 2.5;  // Double
variavel = "25/02/2011"; //String
variavel = DateTime.Parse(variavel); // Data
```

- Exemplo de operadores que alteram o tipo
  - PHP

```
<?php
   $x = 20;
   var_dump($x);
   $x .= 'Texto...';
   var_dump($x);
?>
```

  - JavaScript

```
<script>
   var x = 20;
   alert(typeof(x));
   x += "Texto...";
   alert(typeof(x));
</script>
```

Amarrações de Armazenamento

- **Alocação** ⇨ Amarração feita em uma célula de memória disponível obtida de um pool de memória
- **Desalocação** ⇨ Devolver a célula de memória desvinculada à variável de volta ao pool de memória
- **Tempo de vida** ⇨ O tempo no qual aquela variável está vinculada à célula de memória alocada
  - Começa quando a variável é alocada e termina quando ela é desalocada

Amarrações de Armazenamento

- Classificação das variáveis quanto à amarração de armazenamento
  - Variáveis estáticas
  - Variáveis *stack*-dinâmicas (pilha)
  - Variáveis *heap*-dinâmicas
    - Explícitas
    - Implícitas

Variáveis Estáticas

- Vinculadas à memória antes da execução e permanecem até o programa terminar

- Vantagens
  - Variáveis em subprogramas sensíveis à história (retêm valores entre chamadas)
  - Eficiência: endereçamento direto; inexistência de *overhead* de alocação e desalocação

- Desvantagens
  - Flexibilidade reduzida (recursão)
  - Impossibilidade de compartilhamento da memória

Variáveis Estáticas

- Exemplo em C

```c
#include <stdio.h>

static char* var = "global";

void main() {
   printf("%s\n", var);
}
```

Em C esta alocação é feita em tempo de carga

- Exemplo em Java

```java
class Foo {
   public static String bar = "foobar";
}
```

Em Java esta alocação é feita em tempo de carregamento da classe

Variáveis *Stack*-Dinâmicas

- Vinculadas à células de memória no momento da declaração, mas o tipo é definido estaticamente

- Vantagens
  - Permite recursividade
  - Permite o compartilhamento de memória

- Desvantagens
  - Overhead de alocação e desalocação em tempo de execução
  - Não são sensíveis à história

Variáveis Stack-Dinâmicas

- Exemplo em C

```
void funct(int size) {
  char buffer[size];
  /* ... */
}
```

Variáveis *Heap*-Dinâmicas Explícitas

- Células (abstratas) de memória alocadas e desalocadas pelo programador durante a execução do programa
- A *heap* mantém uma coleção de células alocadas dinamicamente (altamente desorganizada)
- Células somente podem ser referenciadas através de ponteiros ou referências
- A vinculação de tipo é feita estaticamente, mas o armazenamento é feito dinamicamente

Variáveis Heap-Dinâmicas Explícitas

- Vantagens
  - Gestão dinâmica de memória

- Desvantagens
  - Custo associado à manutenção da *heap* (alocação/desalocação)
  - A utilização de ponteiros não é segura

Variáveis Heap-Dinâmicas Explícitas

- Exemplo em C

```c
int* node = malloc(sizeof(int));
/* ... */
free(node);
```

- Exemplo em C++

```cpp
int* node = new int;
/* ... */
delete node;
```

- Exemplo em Java

```java
Integer node = new Integer(10);
/* ... */
```

Onde é feita a liberação da memória em Java?

Variáveis Heap-Dinâmicas Implícitas

- Armazenamento (célula de memória) somente é associado quando um valor é atribuído

- Vantagens
  - Elevado grau de flexibilidade do tipo de variável, permitindo definir códigos genéricos

- Desvantagens
  - Ineficiente porque todos os atributos da variável são dinâmicos
  - Dificuldade de detecção de erros pelo compilador

Variáveis Heap-Dinâmicas Implícitas

- Exemplo em JavaScript

```
<script>
    var list = [ 5, "texto" ];
    alert(typeof(list));
</script>
```

- Exemplo em Perl

```
#!/usr/bin/perl

my @array = ( "1", "2", "3" );

for $e (@array) {
    print $e . "\n";
}
```

ISSO É TUDO, PESSOAL!