

# Alguns tópicos em programação em astronomia

Paulo Penteado

IAG / USP

[pp.penteado@gmail.com](mailto:pp.penteado@gmail.com)

[http://www.penteado.net/ast/aga0503\\_20110407.pdf](http://www.penteado.net/ast/aga0503_20110407.pdf)

# Sumário

## Linguagens de programação

- Características de linguagens
- Escolha de linguagens

## Organização de código

- Estruturação
- Documentação

## Variáveis: além de números

- Estruturas
- Ponteiros
- Objetos

Esta aula é uma versão abreviada de alguns dos tópicos de um curso de programação em astronomia. Mais detalhes sobre estes assuntos nas apresentações em <http://www.ppenteado.net/pea>

# Opções e escolha de linguagens

**Que linguagem usar? - Qual é a melhor?**

Linguagens de programação não são todas equivalentes.

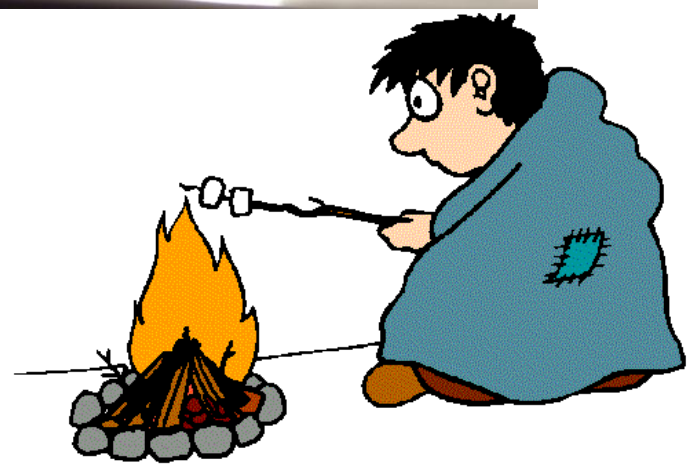
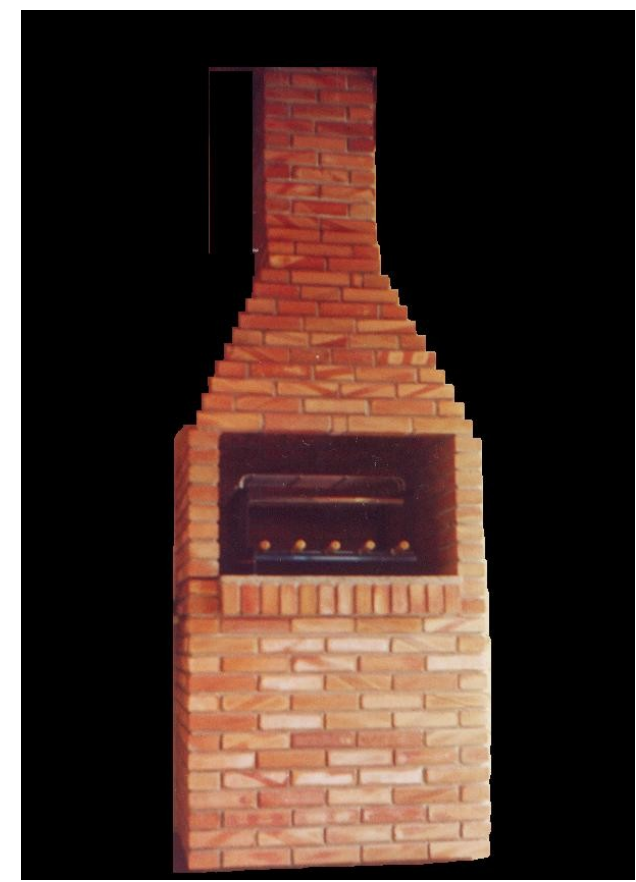
Assim como não são equivalentes

# Opções e escolha de linguagens

Que linguagem usar? - Qual é a melhor?

Linguagens de programação não são todas equivalentes.

Assim como não são equivalentes



Dá para cozinhar qualquer coisa tendo só uma churrasqueira? Dá para fazer arroz usando só um forno a gás? Talvez, mas com muito sofrimento desnecessário.

# Opções e escolha de linguagens

Infelizmente, **não existe “a melhor linguagem”**.

Não há uma que seja a melhor para qualquer uso geral (ou mesmo para qualquer uso comum em astronomia), porque **os problemas a resolver são variados, e as linguagens têm propósitos variados**.

“A melhor” depende **principalmente (não exclusivamente)** de que problema específico o código vai resolver.

Não há (em geral) uma “resposta certa” sobre escolha de linguagem / algoritmo / contêiner.

**Cada linguagem foi desenhada com um objetivo, e se adequa melhor a alguns problemas.**

Exemplos de linguagens mais voltadas para algumas áreas:

- Arrays em várias dimensões – **IDL, Python+NumPy**
- Portabilidade, em múltiplas plataformas – **Java, IDL**
- Strings e expressões regulares – **Perl, Python, IDL, Java, C++**
- Visualização – **IDL, Python, R**
- Algoritmos simples e computacionalmente pesados – **C++, Fortran**
- Estatística (não é só variância, desvio padrão, MQ e  $\chi^2$ ) – **R**

# Opções e escolha de linguagens

**Não é possível escolher uma única para aprender uma vez e nunca mais estudar:**

- Problemas diferentes vão pedir ferramentas diferentes.
- Novas ferramentas (incluindo novas versões das que já existem) vão aparecer.

Por outro lado,

**Em geral não é possível usar sempre a melhor ferramenta do mundo para cada tipo de problema:**

- É comum ter problemas de várias categorias diferentes, e misturar linguagens nem sempre é prático.
- A disponibilidade (para o autor e usuários), e o conhecimento prévio do autor podem fazer com que a linguagem mais ideal para uma classe de problema não seja conveniente para aquela situação.

Ex: Perl é a melhor para processar strings (texto), mas ninguém vai aprender Perl apenas para um único uso:

- É mais fácil usar a linguagem já bem conhecida, mesmo que seja um pouco mais desajeitada.

**Escolha de linguagem a aprender, ou a usar em um projeto, é em geral um compromisso.**

Se o código começar a ficar muito desajeito ou problemático, pode ser um sinal de que a escolha não está (não está mais) sendo a melhor.

Quem vai trabalhar com problemas computacionais tem grande chance de economizar tempo se gastar tempo para aprender formas melhores de resolver os problemas.

**O conhecimento do autor gera um viés para que seja mais fácil usar a linguagem que conhece melhor, mesmo que outra seja mais voltada para o problema.**

**Preferência pessoal (de linguagem, estilo, organização, algoritmos, bibliotecas) também é relevante para escolher a melhor solução:** com as escolhas mais confortáveis, pode ser mais rápido e confiável resolver o problema.

# Características de linguagens

Diferentes linguagens foram criadas para diferentes problemas, por diferentes escolhas.

**Entender a idéia e as escolhas feitas para uma linguagem ajuda a entender como funcionam e para quê.**

Critérios mais importantes (alguns discutidos individualmente adiante):

1. **“Tipo” (compilada X interpretada, estática X dinâmica)**
2. Popularidade
3. Disponibilidade (custo, plataformas disponíveis, portabilidade)
4. Estruturas de dados implementadas (formas de organizar dados em variáveis)
5. Arrays de mais de 1D (de importância freqüente em ciências computacionais)
6. Conteúdo das bibliotecas (padrão e comuns):
  - 6a. Visualização
  - 6b. Armazenamento de dados
  - 6c. Strings e expressões regulares (regex)
  - 6d. Rotinas científicas (matemática, estatística, processamento de imagens, etc.)
  - 6e. Tarefas gerais (acesso ao sistema, interfaces gráficas, rede, documentação, introspecção, *unit tests*, compatibilidade com outras linguagens, etc)

**As discussões e comparações adiante vão em geral omitir, mais notadamente: MATLAB, Mathematica, Ruby, Objective C, C#, Lisp, SQL, Visual Basic, e shells.**

# Características de linguagens - “tipos” de linguagens

As duas mais importantes classificações para linguagens:

- **Código compilado X código interpretado** (não mutuamente exclusivos)
- **Tipos estáticos X tipos dinâmicos**

São categorias independentes, embora freqüentemente correlacionadas

- Linguagens interpretadas **costumam** ter tipos dinâmicos

**Determinam o domínio da maior parte das possibilidades que uma linguagem terá.**

# Características de linguagens – compiladas X interpretadas

Todas as linguagens de uso comum são de relativamente alto nível:

- funcionam com abstrações próximas do usuário

- distantes do baixo nível (código de máquina) usado pelos processadores

# Características de linguagens – compiladas X interpretadas

**Todas as linguagens de uso comum são de relativamente alto nível:**

- funcionam com abstrações próximas do usuário
- distantes do baixo nível (código de máquina) usado pelos processadores

Ninguém hoje vai escrever código de máquina. Ex:

Adding the registers 1 and 2 and placing the result in register 6 is encoded:

[	op		rs		rt		rd	shamt	funct]		
	0		1		2		6		0	32	decimal
	<b>000000</b>		<b>00001</b>		<b>00010</b>		<b>00110</b>		<b>00000</b>	<b>100000</b>	<b>binary</b>

# Características de linguagens – compiladas X interpretadas

Todas as linguagens de uso comum são de relativamente alto nível:

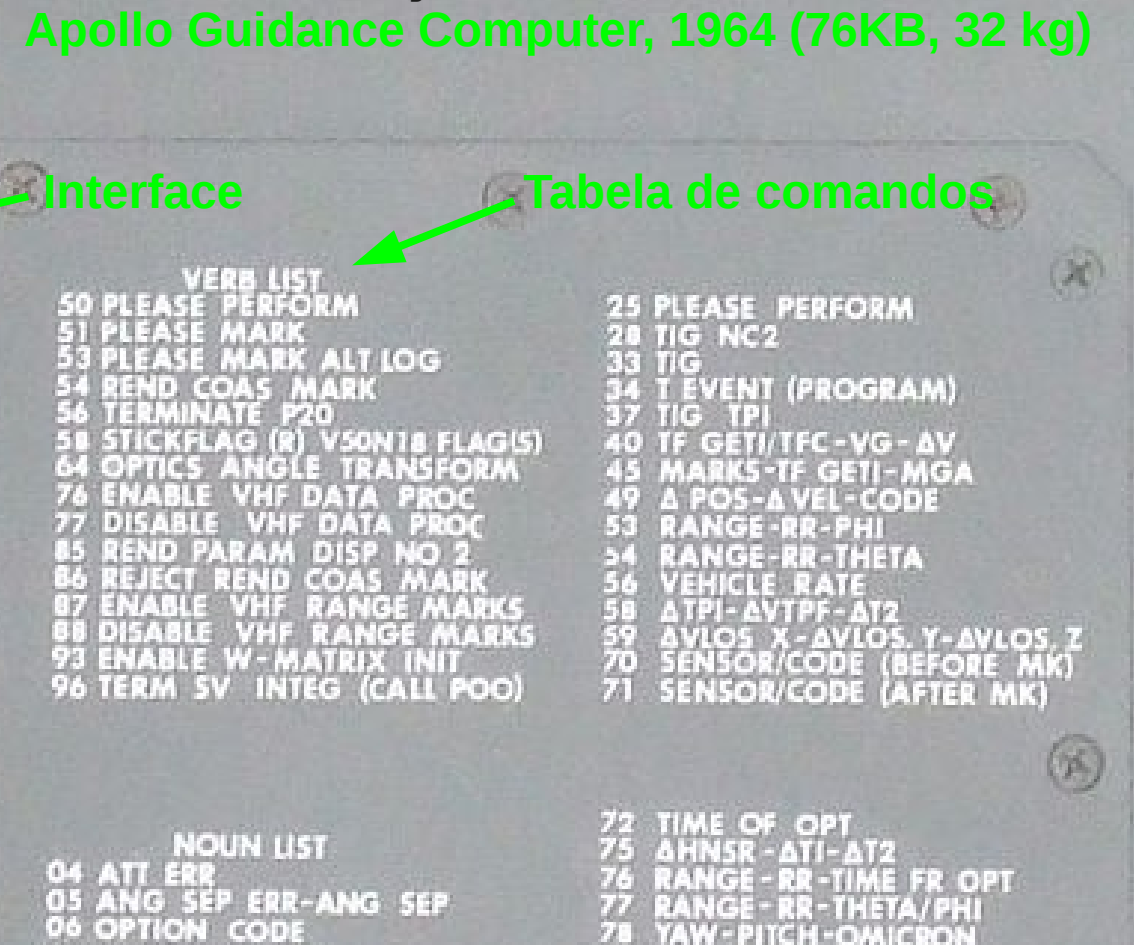
- funcionam com abstrações próximas do usuário
- distantes do baixo nível (código de máquina) usado pelos processadores

Ninguém hoje vai escrever código de máquina:

Adding the registers 1 and 2 and placing the result in register 6 is encoded:

[	op		rs		rt		rd		shamt		funct]
	0		1		2		6		0		32
	000000		00001		00010		00110		00000		100000
											decimal
											binary

Apollo Guidance Computer, 1964 (76KB, 32 kg)



Interface

Tabela de comandos

VERB LIST

50 PLEASE PERFORM  
 51 PLEASE MARK  
 53 PLEASE MARK ALT LOG  
 54 REND COAS MARK  
 56 TERMINATE P20  
 58 STICKFLAG (R) V50N18 FLAG(S)  
 64 OPTICS ANGLE TRANSFORM  
 76 ENABLE VHF DATA PROC  
 77 DISABLE VHF DATA PROC  
 85 REND PARAM DISP NO 2  
 86 REJECT REND COAS MARK  
 87 ENABLE VHF RANGE MARKS  
 88 DISABLE VHF RANGE MARKS  
 93 ENABLE W-MATRIX INIT  
 96 TERM SV INTEG (CALL POO)

25 PLEASE PERFORM  
 28 TIG NC2  
 33 TIG  
 34 T EVENT (PROGRAM)  
 37 TIG TPI  
 40 TF GETI/TFC-VG-ΔV  
 45 MARKS-TF GETI-MGA  
 49 Δ POS-Δ VEL-CODE  
 53 RANGE-RR-PHI  
 54 RANGE-RR-THETA  
 56 VEHICLE RATE  
 58 ΔTPI-ΔVTPF-ΔT2  
 59 ΔVLOS X-ΔVLOS Y-ΔVLOS Z  
 70 SENSOR/CODE (BEFORE MK)  
 71 SENSOR/CODE (AFTER MK)

NOUN LIST

04 ATT ERR  
 05 ANG SEP ERR-ANG SEP  
 06 OPTION CODE

72 TIME OF OPT  
 75 ΔHNSR-ΔTI-ΔT2  
 76 RANGE-RR-TIME FR OPT  
 77 RANGE-RR-THETA/PHI  
 78 YAW-PITCH-OMICRON

SLAVE

# Características de linguagens – compiladas X interpretadas

Compilador/interpretador traduz de uma linguagem de alto nível, com abstrações (código fonte) para a linguagem de máquina (binário executável).

## Diferença no momento da tradução:

- **Linguagens compiladas:** compiladores fazem a tradução previamente, quando o software está sendo desenvolvido. **Na hora do uso só o executável é usado.**
  - A forma mais tradicionalmente usada, principalmente em softwares pesados.
  - Usadas na maior parte do software, tanto comercial (você provavelmente não compilou o seu sistema operacional, navegador, editor de texto, etc.) como o específico para um problema (o tipo de código desenvolvidos neste curso).
- **Linguagens interpretadas:** Interpretadores fazem a tradução “ao vivo”, na hora do uso do programa.
  - Tem se tornado mais comuns em décadas recentes, por ser mais fácil usar.

# Características de linguagens – compiladas X interpretadas

**Em linguagens compiladas, o peso de uma (re)compilação é geralmente grande**

- Pode ser um processo lento e difícil, dependendo de muitos outros recursos externos (compilador, bibliotecas), que o usuário pode nem ter (e nunca chegar a precisar).

Linguagens exclusivamente/normalmente **compiladas**:

- C
- C++
- Fortran

Linguagens exclusivamente/normalmente **interpretadas**:

- R
- Perl
- Shells (bash, csh, etc.)

Linguagens **mistas/variáveis**:

- Java
- IDL
- Python

# Características de linguagens – compiladas X interpretadas

Em linguagens compiladas o trabalho de analisar o código é feito só uma vez, na hora de compilar.

Nas interpretadas, a análise é feita a cada execução: o computador realiza mais trabalho ao executar o programa, podendo demorar mais.

Mas esta **diferença pode ser irrelevante em muitas situações.**

**O tempo que um código consome não é só tempo de execução:** tempo de escrever, consertar, manter e testar o código também é relevante (muitas vezes demora muito mais que o uso do programa pronto). Exs:

- Problemas deste curso
- Millenium Simullation, de  $\sim 10^5$  partículas, levou só 28 dias para rodar.

Em muitas linguagens interpretadas (exs: IDL, Python, Java), muitas **partes mais computacionalmente pesadas são implementadas internamente de forma otimizada.**

# Características de linguagens – estáticas X dinâmicas\*

\*tipos estáticos X tipos dinâmicos

## O que é uma variável?

Antigamente, era um só nome para se referir ao conteúdo de um lugar da memória:

No lugar de dizer

*armazene o inteiro 2 na posição 47 (da memória)*  
*adicione o inteiro 1 ao conteúdo da posição 47*  
*imprima o conteúdo da posição 47*

Podia-se dizer (em pseudocódigo\*\*)

```
int number_of_days  
number_of_days=2  
number_of_days=number_of_days+1  
print number_of_days
```

É muito melhor se referir ao nome *number\_of\_days* do que aos lugares na memória:

- O nome indica o que aquele valor representa
- É mais legível e portátil

Muitos ainda pensam em variáveis apenas assim: só um nome a associar a um número ou string armazenado na memória (potencialmente não sendo um escalar).

\*\*Nenhuma linguagem específica

# Características de linguagens – estáticas X dinâmicas

## Uma variável é algo muito mais elaborado que só um lugar na memória:

- Uma variável é uma representação de um **tipo** de informação no programa.
- Um **tipo** é uma abstração para um conceito.
  - Exs: inteiros, reais, strings (texto), complexos, etc.
- **Internamente, não existem estes conceitos:** Não há um número real na memória. Há uma representação (binária) dele, através das regras definidas pelo tipo real.
- **Esta abstração inclui regras para seu uso.** Exs:
  - Somar 2 inteiros não é a mesma operação que somar 2 reais (*floats, doubles*). O processador usa regras diferentes para operar sobre valores inteiros ou sobre reais.
  - **3/2** é **1**, enquanto **3.0/2.0** é **1.5**
  - **acos(2.0)** não existe para reais, mas existe para complexos
  - Codificação de strings (**muitas** possibilidades diferentes)\*
  - Regras para ordenar strings (podem variar em critérios como ordem entre números, maiúsculas, minúsculas, etc.)

# Características de linguagens – estáticas X dinâmicas

**Como (e quando) se designa o tipo de uma variável?**

**Linguagens de tipos estáticos:** tipos designados no momento da compilação do programa

**Linguagens de tipos dinâmicos:** tipos designados no momento da execução da linha do programa

# Características de linguagens – estáticas

**Linguagens de tipos estáticos:** tipos designados no momento da compilação do programa

**Existem declarações de variáveis**

A **declaração** diz ao compilador para criar uma variável de algum tipo (e dimensões, se array), e associar a ela um nome (símbolo). Ex (Fortran a partir do 90):

```
integer number_of_days  
character*70 file_name  
real :: image(1024,768)  
real, allocatable:: temperatures()
```

**Na hora da compilação** o compilador cria as instruções para alocar estas variáveis, e uma lista de símbolos, que associa os nomes às variáveis na memória

**A lista de símbolos é *estática*:** estes tipos não podem mudar dentro do programa.

# Características de linguagens – estáticas

Em linguagens estáticas:

- **Variáveis precisam ser declaradas.**
- **Variáveis não podem mudar de tipo dentro do programa.**
- Os tipos (e em alguns casos, tamanhos\*) **precisam ser conhecidos na hora da compilação.**
- As interfaces de rotinas usam tipos (e em alguns casos, tamanhos\*) explicitamente especificados para os argumentos.

\*Dimensões de arrays (vetores, matrizes) e/ou comprimento de strings, dependendo do caso

# Características de linguagens – dinâmicas

**Linguagens de tipos dinâmicos: tipos designados no momento da execução da linha de código onde acontece uma atribuição (a um nome não qualificado).**

Exemplos: **IDL, Python, R, Perl.**

Em geral não se sabe na hora de escrever o código quais serão os tipos e dimensões das variáveis.

- Obter informações sobre variáveis (exs: dimensões, tipos) é uma das formas de ***introspecção***, algo essencial em linguagens dinâmicas, para o programa decidir o que fazer, de acordo com o que encontra.

# Características de linguagens – dinâmicas

Exemplo de mudanças em uma variável, e uso de introspecção para saber sobre ela (IDL):

```
IDL> help,some_variable
```

```
SOME_VARIABLE UNDEFINED = <Undefined>
```

→ *some\_variable* ainda não existe

```
IDL> some_variable=2
```

```
IDL> help,some_variable
```

```
SOME_VARIABLE INT = 2
```

→ Agora, é um int

```
IDL> some_variable=[2d0,5d0,74.327d0,!dpi]
```

```
IDL> help,some_variable
```

```
SOME_VARIABLE DOUBLE = Array[4]
```

→ Agora, é um array de 4 doubles

```
IDL> print,some_variable
```

```
2.0000000 5.0000000 74.327000 3.1415927
```

```
IDL> print,n_elements(some_variable)
```

```
4
```

```
IDL> some_variable=!null
```

```
IDL> help,some_variable
```

```
SOME_VARIABLE UNDEFINED = !NULL
```

→ Agora, é não definida de novo

```
IDL> print,n_elements(some_variable)
```

```
0
```

# Características de linguagens – estáticas X dinâmicas

**Introspecção** sobre variáveis permite que em linguagens dinâmicas todas as rotinas sejam genéricas com relação ao tipo e dimensões de seus argumentos.

O que simplifica muito escrever interfaces.

Em uma linguagem estática, todos os argumentos têm que ser declarados de forma idêntica na rotina os recebe, e onde aquela rotina é chamada.

Ex (Fortran):

```
function average(num1,num2)
implicit none
real, intent(in) :: num1, num2
real :: average
average=(num1+num2)*0.5d0
end function average

program use_average
implicit none
real :: num1=3d0, num2=4d0
real :: average
print '(E15.5)', average(num1,num2)
end program use_average
```

Declarações precisam ser compatíveis



Seria um erro ter usado

```
print '(E15.5)', average(3.0, 4)
```

Porque os argumentos são do tipo errado (um é precisão simples, o outro é inteiro).

# Características de linguagens – estáticas X dinâmicas

Em uma linguagem dinâmica, a função não especifica os tipos dos argumentos:

O mesmo exemplo em IDL:

```
function average, num1, num2  
return, (num1+num2)*0.5d0  
end
```

```
pro use_average  
num1=3d0  
num2=4d0  
print, format='(E15.5)', average(num1, num2)  
end
```

Seria perfeitamente válido ter usado

```
print format='(E15.5)', average(3.0, 4)
```

E até arrays (explcado na aula de contêiners):

```
print, format='(E15.5)', average([3d0, 9d0, 11d0], 4d0)
```

o que teria resultado em

```
3.50000E+00  
6.50000E+00  
7.50000E+00
```

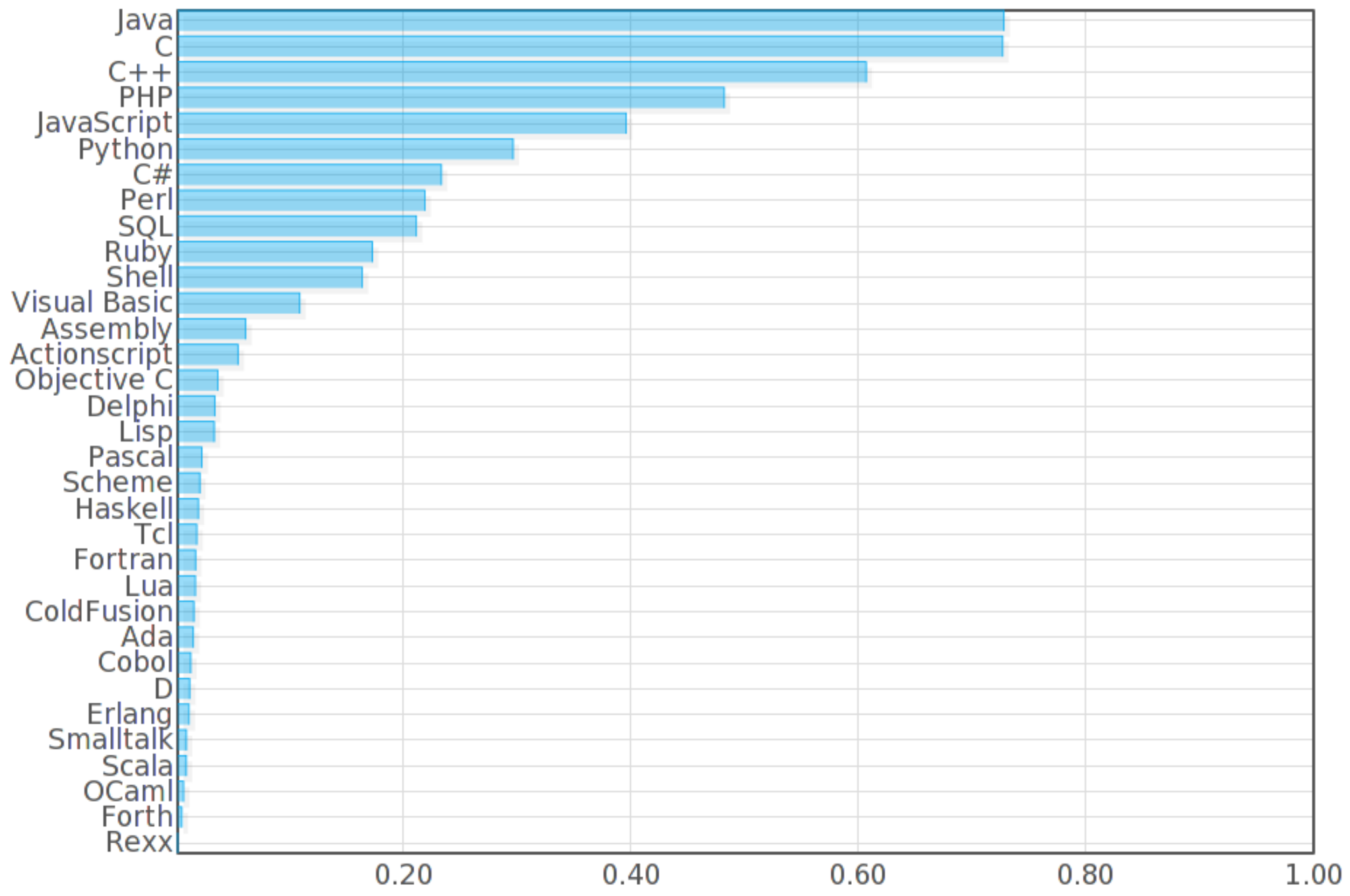
# Características de linguagens – popularidade

Sim, popularidade é relevante.

Para as linguagens dominantes:

- Há (boas e bem testadas) **rotinas prontas para quase toda tarefa geral**, e para muitas das tarefas comuns nas áreas em que ela são usadas.
- Há **muitas referências para as aprender**.
- Há **muitos usuários para fornecer ajuda** (e muitos que já fizeram as mesmas perguntas).
- Há **muitos usuários encontrando os bugs**, muitos dos quais já vão ter sido consertados.
- **É mais fácil compartilhar código**.
- **É mais fácil que elas estejam disponíveis / sejam instaladas**.

# Características de linguagens – popularidade (langpop.com)



As linguagens científicas (IDL, R MATLAB, Mathematica) nem aparecem na lista.

# Organização de código

Um dos fatores mais importantes para determinar a qualidade de um software:

- Quanto tempo (e sofrimento) ele consome para ser escrito / editado / analisado
- A robustez
- A reusabilidade

**Código-fonte deve ser compreensível por pessoas (não só pelo compilador / interpretador).**

**Documentação é também uma parte importante, mesmo sendo não executável.**

**Só porque o programa compila / executa não significa que esteja seja adequado:**

- O compilador / interpretador tem memória perfeita e muito maior que a das pessoas, por isso consegue não se perder, e manter a contabilidade de tudo que está acontecendo.
- Particularmente relevante quando o código for ser lido no futuro muito distante (daqui a um mês), quando nem o autor se lembra como ele funciona.

**Um programa que executa pode não fazer a coisa certa:**

- Inspeccionar o código é uma importante forma de saber se o resultado está certo.

# Organização de código

**Código mal organizado pode também ser pouco eficiente. Exs:**

- Gerando cópias desnecessárias de grandes volumes de dados
- Usando mais memória que o necessário
- Acessando disco / memória de forma desordenada (mais lenta que sequencial)
- Reexecutando linhas desnecessariamente.
- **Muito mais tempo é consumindo o escrevendo, testando, consertando e modificando.**

Uso de contêiners (variáveis que contém vários valores, como arrays, listas, etc.), objetos e vetorização são importantes para melhorar a organização de código. Mas não há tempo para discutir aqui.

# Organização de código - estruturação

**Estruturação significa dividir o código em unidades (rotinas, procedimentos) menores.**

Cada tarefa bem definida fica em uma rotina separada.

**O programa todo se torna hierarquizado.**

Motivos para estruturação (exemplos adiante):

- Facilita inspeção, teste, e edição: **nenhuma unidade é longa demais.**
- Unidades podem ser facilmente reaproveitadas.
- Sabe-se que o código de cada unidade está relacionado apenas ao que ela faz; não há linhas de outras tarefas misturadas no meio.
- Há menos variáveis em escopo: diminui a confusão, o uso de memória, e facilita a edição. Uma variável (local) não vai afetar código fora da unidade.
- **Unidades podem ser facilmente substituídas.**
- **Unidades podem ser testadas individualmente**, em geral de forma muito mais fácil e robusta que quando tudo fica misturado em uma longa rotina.
- Facilita colaboração, quando diferentes autores trabalham em diferentes unidades.

# Organização de código - estruturação (exemplo negativo)

**Um exemplo péssimo (caso real em uso em Astronomia):**

**Uma rotina (Fortran) de ~4500 linhas, onde as primeiras ~350 são declarações de variáveis, além de variáveis vindo de 12 módulos externos:**

- Até compilador e debugger têm dificuldade de lidar com ela.
- Como alguém pode entender tudo que a rotina faz, onde começa e termina cada pedaço do que ela faz, e onde são usadas as centenas de variáveis?
- Quanta memória seria economizada se não houvesse tantas variáveis em escopo simultaneamente?
- Como decidir que nome usar para criar uma nova variável?
- Ao o editar, como saber se um conjunto de 600 linhas que faz uma tarefa pode ser substituído por outra implementação da mesma tarefa?
  - Como saber se no meio destas 600 linhas não há linhas fazendo coisas que nada têm a ver com esta tarefa?
  - Como saber quais são todas as variáveis usadas nessas 600 linhas?
- Destas ~4500 linhas, há só ~1800 com comentários (não é incomum, em código bem organizado, haver mais linhas de documentação do que de código).

# Organização de código – estruturação

**Como definir a estrutura para o programa?**

**Antes de começar, é necessário ter uma idéia clara, para cada unidade, de:**

- 1) O que ela precisa receber de entrada
- 2) O que ela deve fazer
- 3) O que ela deve retornar a quem a chamou

(1) e (3) constituem a interface da rotina.

(2) é o corpo da rotina – **o que** ela faz, em geral através do uso de outras rotinas (que determinam **como** é feita a tarefa).

(1), (2) e (3) formam uma **especificação** da rotina: tudo que é necessário dizer para a programar.

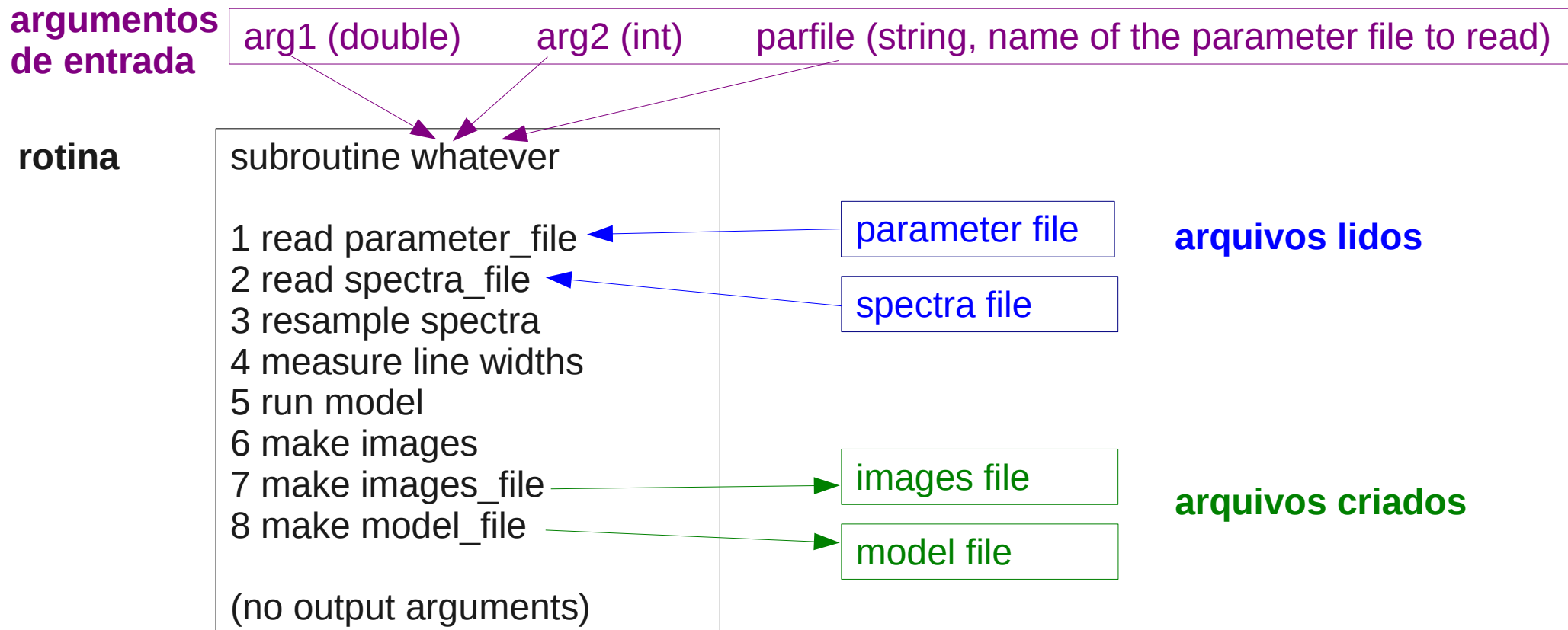
**Quem programa a rotina não precisa ser quem a especificou.** Mesmo que seja, é importante definir os 3 pontos acima de forma clara, antes de começar a escrever o código.

Freqüentemente é recomendado escrever estes 3 pontos para cada rotina a ser criada, e/ou desenhar um fluxograma de como o programa todo vai ser organizado.

# Organização de código – estruturação

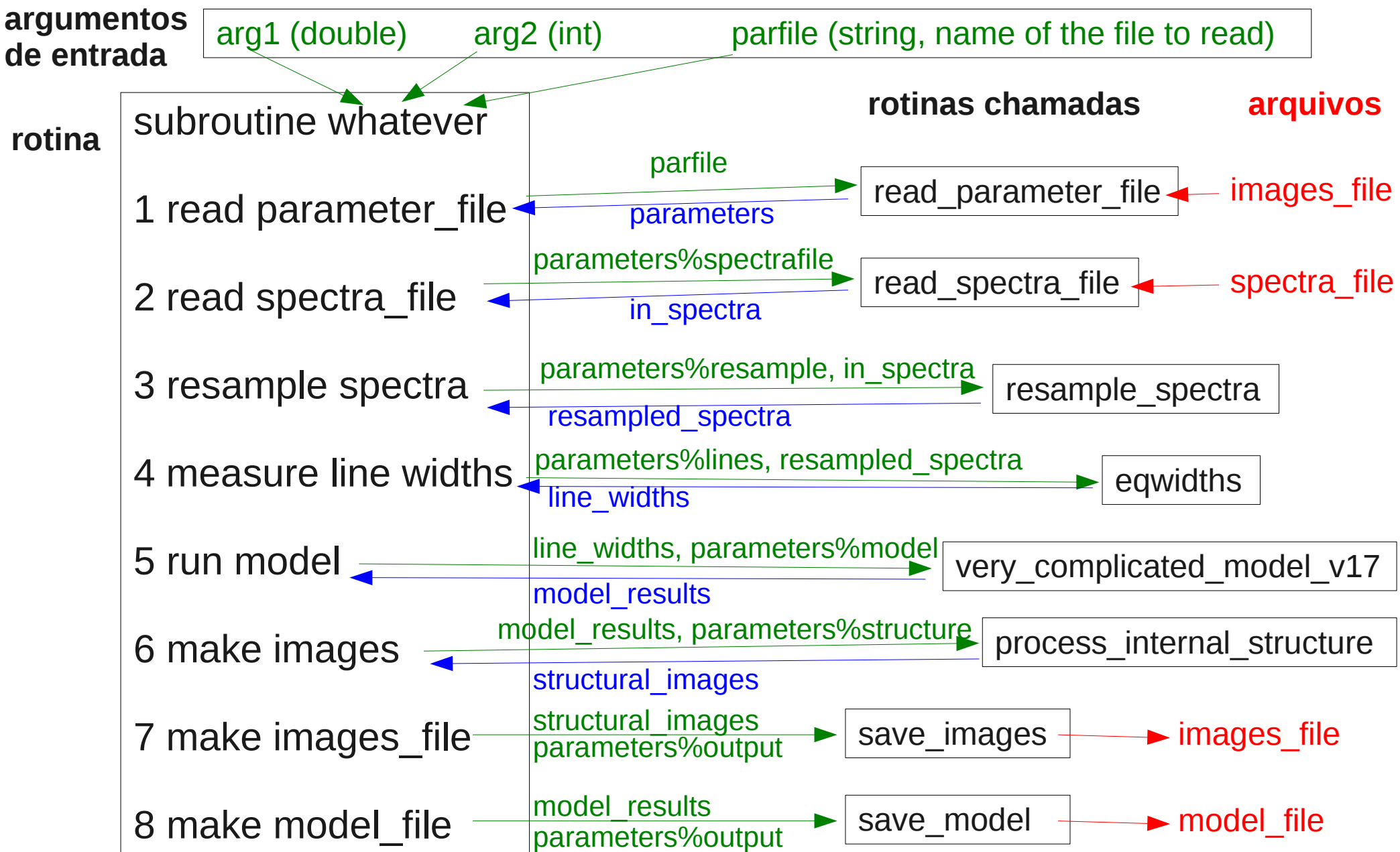
Um diagrama para pensar (visualizar) o que a rotina faz:

- Mostra tudo que a rotina faz (1-8).
- Não mostra como 1-8 são feitos, o que não importa agora.
- Mostra como ela se relaciona com o mundo (argumentos de entrada e saída, arquivos lidos e escritos).



# Organização de código – estruturação

Como fazer cada passo? Chamando uma rotina para cada (e cada uma destas pode chamar outras, para fazer partes específicas).



# Organização de código – estruturação – exemplo (Fortran)

```
subroutine whatever(arg1, arg2, parfile=parfile)  
(várias linhas de documentação)  
(só há 9 variáveis a declarar)  
  
!Read parameters from file  
call subroutine read_parameter_file(parfile, parameters)  
  
!Read the spectra to process  
call subroutine read_spectra_file(parameters%spectrafile, in_spectra)  
  
!Resample the input spectra to another resolution  
resampled_spectra=resample_spectra(in_spectra, parameters%resample)  
  
!Measure the equivalent width of the relevant lines in the spectra  
line_widths=eqwidths(resampled_spectra, parameters%lines)  
  
!Use the super fancy model that will answer everything  
call subroutine very_complicated_model_v17(parameters%model, line_widths,  
model_results)  
  
!Make images showing the structure of the object calculated by the model  
call subroutine process_internal_structure(model_results, parameters  
%structure, structural_images)  
  
!Write these nice results to files  
call subroutine save_images(structural_images, parameters%output)  
call subroutine save_model(model_results, parameters%output)  
end subroutine whatever
```

# Organização de código – estruturação – exemplo (Fortran)

**A versão estruturada é fácil de ser inspecionada:**

- Há só 19 linhas de código.
- Há documentação suficiente.
- Há só 9 variáveis em escopo, sendo 6 locais.

**Fica fácil ver que a rotina só faz 6 tarefas:**

- |                       |             |
|-----------------------|-------------|
| 1) Lê parâmetros      | (1 rotina)  |
| 2) Lê arquivos        | (1 rotina)  |
| 3) Processa os dados  | (2 rotinas) |
| 4) Calcula os modelos | (1 rotina)  |
| 5) Gera visualizações | (2 rotinas) |
| 6) Salva resultados   | (1 rotina)  |

**A complexidade de cada uma das 6 tarefas não fica exposta, nem misturada entre tarefas.**

**A complexidade está escondida (*encapsulada, compartimentalizada*) em 8 rotinas que são chamadas.**

**Fica claro como dados são passados entre tarefas.**

# Organização de código – estruturação – exemplo (Fortran)

**Fica muito melhor inspecionar / editar / testar o que é feito em cada uma das 8 rotinas:**

- O código de cada uma não se mistura com o código das outras.
- Não há variáveis das outras, que não interessam, presentes.
- Em cada uma fica explícito o que são as dependências externas (as variáveis que aparecem na interface), e as variáveis locais não são expostas.
- Cada rotina pode ser escrita e testada individualmente.

**É fácil reaproveitar uma destas rotinas para outros programas.**

**É fácil substituir uma destas rotinas por outra que faça algo diferente:**

- Não é mais necessário reamostrar o espectro? Troca-se a chamada de rotina  
`resampled_spectra=resample_spectra(in_spectra, paramters%resample)`  
Por  
`resampled_spectra=in_spectra.`
- É melhor reamostrar usando outro método? Troca-se pelo uso de outra rotina.

# Organização de código - interfaces

Interfaces têm grande importância sobre a organização do código.

Só com interfaces organizadas uma rotina é usável, sem sofrimento.

## Problemas comuns:

- Muitos argumentos posicionais. Exemplo (real, em Fortran):

```

SUBROUTINE DISORT( NLYR, DTAUC, SSALB, PMOM, TEMPER, WVNML0,
&                WVNMMHI, USRTAU, NTAU, UTAU, NSTR, USRANG, NUMU,
&                UMU, NPHI, PHI, IBCND, FBEAM, UMU0, PHI0,
&                FISOT, LAMBER, ALBEDO, HL, BTEMP, TTEMP, TEMIS,
&                DELTAM, PLANK, ONLYFL, ACCUR, PRNT, HEADER,
&                MAXCLY, MAXULV, MAXUMU, MAXCMU, MAXPHI, RFLDIR,
&                RFLDN, FLUP, DFDT, UAVG, UU, U0U, ALBMED,
&                TRNMED )

```

**47 argumentos posicionais** (correspondência é feita apenas pela sua ordem).  
É pior ainda: todos estes estão declarados estaticamente:

```

PARAMETER ( MXCLY =6, MXULV = 5, MXCMU = 48, MXUMU = 10,
&          MXPHI = 3, MI = MXCMU / 2, MI9M2 = 9*MI - 2,
&          NNLYRI = MXCMU*MXCLY, MXSQT = 1000 )

```

Qualquer mudança nas dimensões exige que se mude os valores nesta rotina, e na que a chama, e recompilar.

Se as declarações não forem iguais nos dois lugares, a rotina pode até rodar, mas fará coisas erradas.

# Organização de código - interfaces

Em linguagens estáticas, interfaces são particularmente trabalhosas: **Todos argumentos devem ter declarações compatíveis, na rotina e onde ela é usada:**

- Declarações copiadas entre lugares exigem manter as duas cópias sincronizadas.
- O problema é aliviado com módulos e tipos parametrizados (Fortran 90-2008, mais rudimentar) e *header files* (C, C++), *templates* (C++) e *genéricos* (Java).
- É possível fazer rotinas genéricas, que aceitam argumentos de tipos e tamanhos variáveis.
  - Mas é necessário ou lidar com eles explicitamente (só praticável para casos muito simples) ou usar *templates* / *genéricos* (maior complexidade).

**Linguagens dinâmicas facilitam muito a organização, pelo trabalho muito pequeno imposto pela interface de uma rotina:**

- Não há declarações.
- Não há problemas com tipos / tamanhos.
- Nos casos em que a rotina depende de tipos / tamanhos, ela pode o testar, por *introspecção* (rotinas que dão informações sobre as variáveis).
- Keywords (argumentos associados por nome), números variáveis de argumentos, manipulação dos conjuntos de argumentos, e defaults são feitos facilmente, em grande parte por introspecção.

# Organização de código - interfaces

**É comum uma rotina depender de muitas variáveis:**

- Para definir o que vai fazer
- Para retornar resultados.

**Ex: um modelo complicado pode depender de dezenas de parâmetros.**

**Como reduzir o número de argumentos passados (posicionais ou não)?**

- Alguns usam variáveis globais (*global / common*), que são variáveis visíveis em todas as rotinas dentro de um programa. **Que em geral é uma solução ruim:**
  - Variáveis globais fazem com que uma rotina dependa do ambiente externo de uma forma que não aparece em sua interface. Variáveis passam “por baixo do pano”.
  - Módulos (Fortran) e *includes* (inclusão de arquivos dentro de outros) são melhores, pois tudo que é passado escondido está em um mesmo lugar (o módulo ou arquivo incluído): **Coisas ficam escondidas, mas são mais fáceis de encontrar.**

# Organização de código - interfaces

## Como melhorar a passagem de muitos parâmetros?

### Duas soluções (não mutuamente exclusivas):

- **Contêiners** (arrays, estruturas, listas, mapas, objetos):
  - Carregam vários valores necessários, de forma organizada por ordem (listas, arrays) ou nome (estruturas, mapas, objetos).
  - O número de variáveis é pequeno, e estas são organizadas.
- **Argumentos associados por nome (*keywords*):**
  - No lugar de  

```
call subroutine(t, p, x, y, z, r, ...)
```

Se usa

```
call subroutine(x=x, y=y, z=z, r=r, temperature=t,  
pressure=p, ...)
```
  - **Principalmente úteis para argumentos opcionais**
  - Argumentos de entrada (substituídos por defaults quando não fornecidos)
  - Argumentos de saída (que nem sempre interessam; se não foram usados na chamada da rotina, esta pode pular as partes que geram estas saídas).

# Organização de código - interfaces

Voltando ao exemplo anterior, a interface original (Fortran)

```
SUBROUTINE DISORT( NLYR, DTAUC, SSALB, PMOM, TEMPER, WVNML0,  
&                WVNMMHI, USRTAU, NTAU, UTAU, NSTR, USRANG, NUMU,  
&                UMU, NPHI, PHI, IBCND, FBEAM, UMU0, PHI0,  
&                FISOT, LAMBER, ALBEDO, HL, BTEMP, TTEMP, TEMIS,  
&                DELTAM, PLANK, ONLYFL, ACCUR, PRNT, HEADER,  
&                MAXCLY, MAXULV, MAXUMU, MAXCMU, MAXPHI, RFLDIR,  
&                RFLDN, FLUP, DFDT, UAVG, UU, U0U, ALBMED,  
&                TRNMED )
```

Foi substituída por apenas (ainda Fortran)

```
subroutine disort_pp_run(dsv,dsr)
```

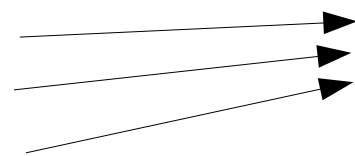
**Sem alocações estáticas** (dimensões declaradas no código fonte).

**Todas as dimensões são dinâmicas** (determinadas quando o programa executa).

# Organização de código - interfaces

A nova interface carrega as mesmas informações de entrada e saída, em duas estruturas. O seu uso fica apenas:

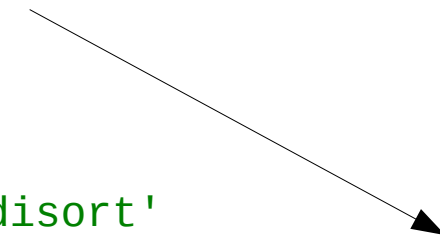
```
dsv%nlayers=nlayers
dsv%degree=degree
dsv%naz=naz
```



As dimensões do modelo são colocadas na estrutura dsv

```
call disortset(dsv,dsr) !aloca os componentes e ajusta os defaults em dsv,dsr
```

```
dsv%tau=tau
dsv%ssa=ssa
dsv%pmom=pmom
dsv%inccos=inccos(1)
dsv%az=azang
```



Estruturação do código: esta parte foi colocada em outra rotina (46 linhas), que apenas define os defaults para os 48 parâmetros (acessados por nome, obviamente)

```
write(6,*) 'calling disort'
```

```
call disort_pp_run(dsv,dsr)
```

Os 5 parâmetros que não vão ser default são colocados na estrutura dsv

A rotina é chamada, com apenas 2 variáveis:  
dsv para argumentos de entrada  
dsr para os de saída

# Organização de código - exemplo

Padrão exigido pela ESA para software de análise e processamento de dados do projeto Gaia (em Java):

- Documentação (formato javadoc)
- *Unit tests* (executados automaticamente todo dia)
- Os métodos (rotinas) não podem exceder:
  - 30 linhas de código
  - 5 argumentos
  - Profundidade 5
  - Complexidade ciclomática 10
- As classes não podem exceder:
  - 12 atributos
  - 20 métodos
  - Herança de até 5 classes

Todo o software é mantido em um repositório de controle de versão, com um gerenciador que executa os testes diariamente e gera relatórios da conformidade das classes (Hudson).

# Organização de código - controle de fluxo

Não é mito: *GOTO* nunca deve ser usado

October 1995



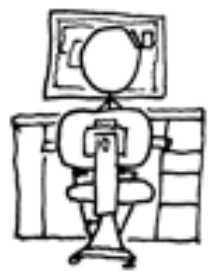
## **Go To Statement Considered Harmful** *Edsger W. Dijkstra*

---

Reprinted from *Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147-148. Copyright © 1968, Association for Computing Machinery, Inc.

# Organização de código - controle de fluxo

Não é mito: *GOTO* nunca deve ser usado



# Organização de código - controle de fluxo

**Não é mito:** *GOTO nunca deve ser usado*

É uma construção de baixo nível, remanescente de código de máquina:

- Só computadores (com memória perfeita e grande) são capazes de manter a contabilidade do fluxo usando saltos arbitrários (*gotos*).
- Tornam o fluxo do programa **arbitrário, e não estruturado**.
- Para entender o código fica necessário encontrar, lembrar de, e considerar **todas** as ocorrências de *gotos*.
- Editar e mover blocos fica muito difícil.

Sua impropriedade em linguagens de alto nível, para pessoas lerem, é conhecida há muito tempo (Dijkstra 1968).

**Não existe em muitas linguagens modernas (Java, Python).**

**A maior parte dos usos de *goto* por programadores hoje se deve a não considerar as estruturas de controle mais apropriadas.** Os mais apropriados em Fortran:

Condicionais (**if ... then ... else / case**)

Loops (**do, while, forall**)

Modificadores (**cycle, exit, return**)

# Organização de código – problemas com literais\* - exs. reais

## (IDL)

```

dlambda = fwhm / 17D0
interlam = lammin + dlambda * DINDGEN( LONG( (lammax-lammin)/dlambda+1 ) )
interflux = INTERPOL( flux, lam, interlam )
fwhm_pix = fwhm / dlambda
window = FIX( 17 * fwhm_pix )

```

Por que 17? Só mencionado em comentário 30 linhas acima  
 Em quantos lugares é necessário mudar o código para usar outro valor?  
 17 ou 17d0?  
 Uma variável deveria ter sido definida.

## (Fortran)

data pi/3.14159/ Precisão muito ruim; em geral há uma função ( $pi()$ ) ou constante ( $!pi$ ) para isso. Ou,  $pi=acos(-1d0)$ .

```

open(19, file='rtinput2-iso', status='unknown'
(...)
do q=1, nlay
  write(19,222) q, dtau(q), ssalb(q)
end do
(...)
close(19)

```

Além de ser um literal perdido no meio do código (linha 171), é uma dependência externa, fixa no código.

Além de ser um literal repetido (deveria ser uma variável), é algo de escopo global (a unidade aberta). Assume que o 19 estará livre.

**Se esta rotina é usada em outro lugar, o 19 pode já estar ocupado.**

\*constantes que aparecem *literalmente* dentro do código: 17d0, 5, 'some\_name', etc.

# Organização de código – problemas com literais e unidades

Geralmente, literais devem:

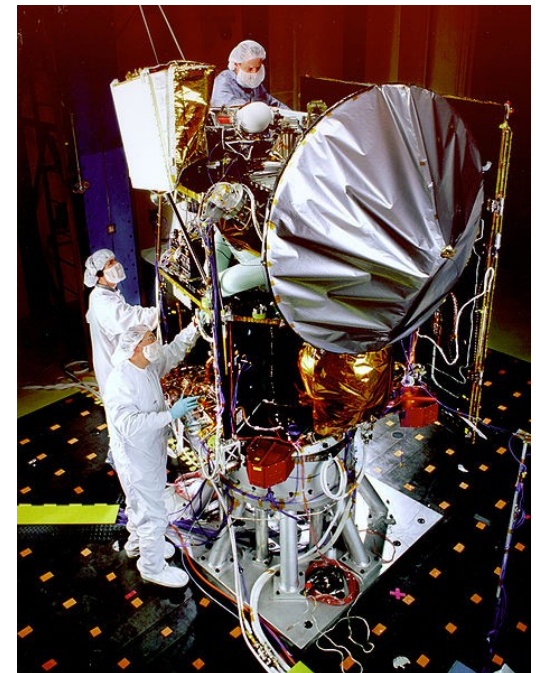
- ser atribuídos a variáveis (ponto único de edição),
- ser explicados com comentários,
- não ser usados para definir recursos globais, que podem não estar disponíveis (ex: número de unidade)

É comum ser desejável que todas as variáveis com literais sejam definidas no começo do código, localizando todas as dependências (de parâmetros, externas) onde são facilmente encontradas.

**Se o valor não é adimensional (literal ou não), comentários devem informar as unidades.**

Foi a causa da perda da Mars Climate Orbiter:

*The metric/US customary units mix-up that destroyed the craft was caused by a human error in the software development. The thrusters on the spacecraft, which were intended to control its rate of rotation, were controlled by a computer that underestimated the effect of the thrusters by a factor of 4.45. This is the ratio between a pound force (the standard unit of force in the United States customary units system) and a newton (the standard unit in the metric system).*



**É raro, mas há softwares (em geral, classes) que contabilizam as unidades de variáveis e suas transformações em operações matemáticas.**

# Documentação

**É essencial! Não é um detalhe para só considerar depois.**

Faz a diferença entre uma rotina ser ou não compreensível, reusável, verificável e útil.

**Há dois tipos importantes de documentação:**

- 1. Para explicar o código - voltada a quem lê o código-fonte** (especialmente o próprio autor), para entender como ele funciona, e por que foi feito daquela forma.
    - Código-fonte deve ser compreensível para pessoas (não só para o compilador/interpretador).**
    - Sempre que escrever uma linha / conjunto de linhas não triviais, escreva na hora um comentário os explicando.**
  - 2. Para explicar o uso do código - voltada aos usuários** (inclusive o próprio autor), para informar para que serve e como usar aquela rotina.
    - Em geral vem um pouco depois na criação do código, quando este se torna menos experimental e suas características estão melhor definidas. Por isso é freqüentemente negligenciada.**
- Se o código mudar, mude os comentários: comentários não atuais / errados são piores que sua ausência.**

# Documentação

**Apesar de negligenciada, a documentação que explica o propósito e uso do código costuma ser a mais necessária:**

- Pode ser lida não só por pessoas, mas também por software, que as usa para criar documentação bem formatada (HTML, LaTeX, PDF, com links para referências), e para as ajudas de contexto das IDEs.
- Para outros usuários do código, é a parte mais importante. Decide se a rotina interessa ou não:
  - Informa o objetivo do código e **o que** ele faz (**não como** ele o faz).
  - Informa se há dependências em outras coisas, e limitações de seu uso.
- Para saber como usar a rotina:
  - Explica todos os argumentos / keywords: seu propósito, o que contém, seu tipo / tamanho, obrigatoriedade, se é entrada e/ou saída.
  - Explica as formas como aquela rotina pode ser usada, e para quê.
  - **Dá exemplos de uso** – uma das partes mais importantes, principalmente mostrando de forma completa como usar a rotina, e que resultado ela deveria gerar quando usada daquela forma.

**Principais sistemas de processamento de documentação:**

- JavaDoc, IDLDoc, Doxygen (C, C++, Fortran, Java Python e muitas outras), Sphinx (Python).



# Tipos de variáveis

## Uma variável pode ser muito mais complicada que um número ou string:

- **Contêiner:** armazena vários valores, organizados por ordem, nome, ou hierarquia.
  - Exs: vetor, matriz, array, lista, mapa, árvore, etc.
  - (assunto da próxima aula)
- **Vários valores de tipos mais simples, agrupados - Estrutura** (adiante)
- **Referência a outra variável** - ponteiro (adiante)
- **Uma abstração que represente um conceito qualquer - objeto** (adiante)
  - Compreende dados, recursos, e formas de operar sobre eles
  - É uma variável ativa ("*inteligente*"): não é apenas um repositório estático de dados, é algo capaz de realizar operações.

# Tipos de variáveis - ex: como representar números complexos?

## Tipo complex

- Tipo não trivial (não é tipo básico em C, C++, Java) mais comum
- Contém **duas partes de outro tipo** (*float* ou *double*), para as partes real e imaginária
- **Funções e operadores sabem o que fazer com cada parte**

Se já não existisse o tipo complex, como usar números complexos?

# Tipos de variáveis - ex: como representar números complexos?

Se já não houvesse o tipo *complex*, como representar um número complexo?

Criar “*algo*”, que contenha as duas partes, e que as funções saibam como as usar, e que possa ser identificado (uma rotina tem como saber se a variável é um complexo).

Que “*algo*”? Uma possibilidade é uma **estrutura\***

- Um novo tipo de variável, que contém elementos (*campos*) dos tipos básicos, para representar um **conceito derivado deles**
- **Agrupar vários elementos relacionados em uma única variável**
- Rotinas/operadores **podem saber como operar de forma específica** para cada tipo, usando a relação conceitual entre os elementos
- Os elementos podem ser qualquer “coisa” já conhecida: escalares ou arrays de qualquer tipo (inclusive, estruturas, e objetos).
- Os **elementos são identificados pelo nome**, sem precisar saber a ordem.

\*Nomes variam: **structure** (IDL), **struct** (C, C++), **derived type** (Fortran), **type\*\*** (Python), **class\*\*** (Java)

\*\*Estruturas são um caso especial de **types** em Python, e de **classes** em Java.

# Tipos de variáveis - ex: como representar números complexos?

**Definição** (por parte real e imaginária):

Fortran:

```
type complexo  
  real :: re, im  
end type complexo  
type(complexo) :: a
```

Definição do tipo

Definição de uma variável  
daquele tipo

**Uso:**

Fortran:

```
a%re=2.0  
a%im=3.0
```

# Outros tipos simples

**Complex** é tão comum que já é um dos básicos em Fortran, IDL, Python, e presente nas bibliotecas padrão de C, C++, Java.

Os **tipos básicos** são só um conjunto pequeno de uso geral.

**Quase todos os outros tipos úteis são muito específicos**, não faz sentido complicar a definição das linguagens com eles.

Fica para **cada programa ou biblioteca definir o que é conveniente**.

Exemplos simples:

- Complexo pelo módulo e fase, no lugar de parte real e imaginária

# Outros tipos simples

- Funções de coordenadas:

- No lugar de uma função

```
func1(r, theta, phi)
```

- Onde r,theta,phi são vetores com `r/theta/phi` para cada um de n pontos:

```
type coord
  real :: r, theta, phi
end type coord
type(coord) :: b(n) !vetor de n elementos do tipo coord

b%r=r
b%theta=theta
b%phi=phi
```

- Então a função pode ser apenas

```
func1(b)
```

- Há apenas 1 argumento
- É garantido que as triplas `r/theta/phi` de cada ponto são mantidas juntas
- Não se misturam coordenadas de pontos diferentes por usar 3 variáveis separadas.

# Estruturas - exemplo – correspondência de muitas variáveis

Rotina com muitos argumentos posicionais (exemplo real, em Fortran):

```

SUBROUTINE DISORT( NLYR, DTAUC, SSALB, PMOM, TEMPER, WVNML0,
&                WVNMMHI, USRTAU, NTAU, UTAU, NSTR, USRANG, NUMU,
&                UMU, NPHI, PHI, IBCND, FBEAM, UMU0, PHI0,
&                FISOT, LAMBER, ALBEDO, HL, BTEMP, TTEMP, TEMIS,
&                DELTAM, PLANK, ONLYFL, ACCUR, PRNT, HEADER,
&                MAXCLY, MAXULV, MAXUMU, MAXCMU, MAXPHI, RFLDIR,
&                RFLDN, FLUP, DFDT, UAVG, UU, U0U, ALBMED,
&                TRNMED )

```

**47 argumentos posicionais** (correspondência é feita apenas pela sua ordem).  
É pior ainda: todos estes estão declarados estaticamente:

```

PARAMETER ( MXCLY =6, MXULV = 5, MXCMU = 48, MXUMU = 10,
&          MXPHI = 3, MI = MXCMU / 2, MI9M2 = 9*MI - 2,
&          NNLYRI = MXCMU*MXCLY, MXSQT = 1000 )

```

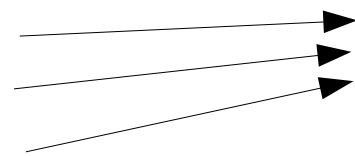
Qualquer mudança nas dimensões exige que se mude os valores nesta rotina, e na que a chama, e recompilar.

Se as declarações não forem iguais nos dois lugares, a rotina pode até rodar, mas fará coisas erradas.

# Estruturas - exemplo – correspondência de muitas variáveis

A nova interface carrega as mesmas informações de entrada e saída, em duas estruturas. O seu uso fica apenas:

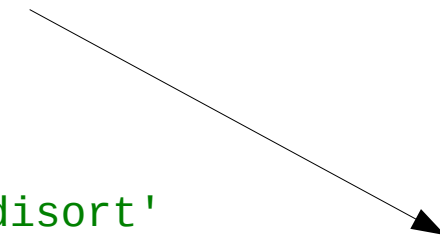
```
dsv%nlayers=nlayers
dsv%degree=degree
dsv%naz=naz
```



As dimensões do modelo são colocadas na estrutura dsv

**call disortset(dsv,dsr)** !aloca os componentes e ajusta os defaults em dsv,dsr

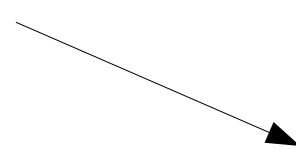
```
dsv%tau=tau
dsv%ssa=ssa
dsv%pmom=pmom
dsv%inccos=inccos(1)
dsv%az=azang
```



Estruturação do código: esta parte foi colocada em outra rotina (46 linhas), que apenas define os defaults para os 48 parâmetros (acessados por nome, obviamente)

```
write(6,*) 'calling disort'
```

```
call disort_pp_run(dsv,dsr)
```



Os 5 parâmetros que não vão ser default são colocados na estrutura dsv

A rotina é chamada, com apenas 2 variáveis:  
dsv para argumentos de entrada  
dsr para os de saída

# Estruturas - exemplo – correspondência de muitas variáveis

Interface decente para a mesma rotina (Fortran):

```
subroutine disort_pp(dsv,dsr)
  use disortutils
  type(disortvar),intent(inout) :: dsv
  type(disortres),intent(inout) :: dsr
```

→ Módulo onde são  
definidos os tipos usados

Todas as variáveis, e todas as dimensões, são colocadas dentro de apenas duas variáveis.

A associação de parâmetros e variáveis é pelo seu nome dentro das estruturas; não importa a ordem.

Onde são definidos os tipos? No módulo usado (**disortutils**).

# Estruturas - exemplo – módulo definindo tipos (Fortran)

```

module disortutils
!types and routines for a decent interface to DISORT
!used by disort_pp.f (decent wrapper to DISORT)
!disort_pp.f uses ErrPack.f, D1MACH.f, R1MACH.f, LINPACK.f
implicit none
integer,parameter :: ddp=selected_real_kind(15,307)
type disortpar
  integer :: mxcly,mxulv,mxcmu,mxumu,mxphi,mi,mi9m2,nnlyri
end type disortpar
type disortvar
!type to carry all the input variables to DISORT
  type(disortpar) :: dsp
  integer :: nlayers,degree,naz,numu,nutau
  real(ddp),allocatable :: tau(:),ssa(:),pmom(:,:),t(:)
  real(ddp) :: wavs(2),topemis,botemis,toptemp,bottemp
  real(ddp) :: albedo,fluxisot,fluxbeam,inccos
  real(ddp),allocatable :: az(:),emcoss(:)
  real(ddp),allocatable :: hl(:)
  logical :: useemis,usedeltam,lamber,onlyfl,usrang,usrtau
  real :: accur
  real(ddp) :: incaz
  logical :: prnt(7)
  character(127) :: header
  integer :: ibcnd
  real(ddp), allocatable :: uemcoss(:),utau(:)
end type disortvar
type disortres
!type to carry all the output variables from DISORT
  real(ddp),allocatable :: albedo(:),trans(:)
  real(ddp),allocatable ::
fluxup(:),fluxdifdown(:),fluxdirdown(:),fluxdiv(:),meanint(:),azavint(:,:)
  real(ddp),allocatable :: intens(:,,:,:)
end type disortres
(...)

```

Definição do tipo disortpar

Definição do tipo disortvars (que contém um elemento do tipo disortpar)

Definição do tipo disortres

# Estruturas - exemplo – correspondência de muitas variáveis

Como usar a rotina para múltiplos casos, para  $n$  comprimentos de onda diferentes?

Na interface arcaica, redefinindo todas as 47 variáveis para ter uma dimensão a mais?

Na interface nova, bastaria:

```
type(disortvar) :: dsv(n)
type(disortres) :: dsr(n)
(...)
do i=1,n
  call disort_pp(dsv(i),dsr(i))
Enddo
```

E se fossem  $m$  modelos para  $n$  comprimentos de onda?

Apenas:

```
type(disortvar) :: dsv(n,m)
type(disortres) :: dsr(n,m)
(...)
do j=1,m
  do i=1,n
    call disort_pp(dsv(i,j),dsr(i,j))
  enddo
enddo
```

# Outros usos de estruturas

**Implementação de outras estruturas de dados:** Listas, pilhas, árvores, mapas.  
(discutidas na aula de estruturas de dados)

**Manter a associação de informações relacionadas.** Exs:

- Campos de dados, mantidos associados a campos de dados relacionados, e campos para indicar atributos dos dados: espécie, unidades, nome da origem (objeto, modelo), dimensões.

**Essencial para objetos:**

- Objetos são estruturas com propriedades adicionais e com funções que acessam os dados. Discutidos adiante.

# Ponteiros - conceitos

**Ponteiros são um dos conceitos que mais variam entre linguagens.**

O conceito de ponteiros da maioria das pessoas é o usado em C, C++, que **difere do usado em linguagens dinâmicas ou baseadas em objetos**, como IDL, Python, Java, R.

Fortran usa um conceito diferente, de baixo nível como C, C++, mas com dereferência implícita, e desajeitado por não existir arrays de ponteiros.

**Universalmente, ponteiros são variáveis que são apenas referências a outras** (os *alvos* dos ponteiros: ponteiros *apontam* para os alvos):

- Um ponteiro pode ser alterado, para apontar para outro alvo (ou para nada).
- Mais de um ponteiro (ou nenhum) pode apontar para um mesmo alvo.
- O alvo do ponteiro é acessado por um operador de *dereferência* (em geral, \*).

# Ponteiros - conceitos

**Linguagens de alto nível têm um novo e mais útil conceito de ponteiros:**

- São simplesmente referências a outras variáveis.
- Não importa para o programador se eles são endereços de memória (em geral não são).
- **Muitas das necessidades de ponteiros de linguagens como C não acontecem em linguagens de alto nível** (esp. arrays e passagem de argumentos por referência).
- **Nas linguagens de alto nível mais completas quase não sobra uso para ponteiros.**
- Algumas linguagens têm como forte deficiência a falta de ponteiros, ou terem ponteiros desajeitados. Exs: R e Fortran.
- Não há os riscos de corrupção, acessos errados, e segfaults de C, C++, Fortran.

# Para que são necessários ponteiros? - “arrays” não regulares

“Arrays” onde cada linha tem um número diferente de elementos:

- Elementos de
  - Famílias de asteróides
  - Aglomerados de estrelas / galáxias
  - Sistemas estelares múltiplos
  - Sistemas planetários
  - Organizações hierárquicas
- Vizinhos de
  - Asteróides (identificação de famílias, classificação de composições)
  - Estrelas (classificações por cores ou indicadores espectrais)
  - Galáxias (classificações por cores, indicadores espectrais, velocidades, forma)
- Linhas / bandas em espectros
  - Linhas / bandas de mesma origem (mesmo íon / elemento / molécula / partícula)
  - Vizinhos (identificação de linhas dominantes e contaminações)
- Observações / resultados de modelos
  - Repetidas observações do mesmo objeto.
  - Diferentes observações do mesmo objeto (diferentes instrumentos / modos).
- Grades não regulares
  - Parâmetros de modelos (modelos calculados para diferentes valores dos parâmetros).
  - Espaçamento não regular em grades espaciais.
  - Modelos com diferentes números de objetos / espécies em diferentes células.

# Ponteiros - ex. 1 – “arrays” com dimensões não constantes:


Um modelo contendo com  $m$  objetos, cada um com um número diferente de pontos. Como armazenar a temperatura de cada ponto de cada objeto?

Poderia-se concatenar os  $m$  vetores com as temperaturas de cada objeto. (Ex. Fortran):

Com um array  $np$  indicando o número de pontos de cada objeto.

Sendo  $m=3$ ,  $np=(/2, 4, 1/)$ , poderia ser :

```
real :: t(7)
t=(/7.9, 5.8, 9.6, 3.6, 1.4, 7.5, 3.2/)
```



t1
t2
t3

2 temperaturas do objeto 1:  $t1=t(1:2)$

4 temperaturas do objeto 2:  $t2=t(3:6)$

1 temperatura do objeto 3:  $t3=t(7:7)$

Mas é desajeitado de usar. Exige carregar junto o vetor  $np$ , e ficar calculando os índices de começo e fim da parte de cada objeto.

# Ponteiros - ex. 1 – “arrays” com dimensões não constantes:

Seria muito melhor criar um array onde cada linha tivesse um número diferente de elementos:

```
t= 7.9 5.8           (t1)
    9.6 3.6 1.4 7.5 (t2)
    3.2              (t3)
```

Uma solução arcaica seria fazer **t** com a largura da maior linha, e deixar o que sobra vazio.

Desajeitado:

- Desperdiça memória e tempo de processamento
- Necessário saber antecipadamente qual é a maior largura
- Exige saber quantos elementos cada linha tem (ex: para não confundir um 0.0 de um elemento vazio com uma temperatura 0.0).

Dá para fazer o array de linhas de largura variável?

Seria um “array de arrays”: cada elemento **v[i]** seria o array com as **np[i]** temperaturas do objeto **i**.

**Com ponteiros ou listas é possível.**

# Ponteiros - conceito

**Como um guarda-volumes, ou bibliotecário:** o bibliotecário coloca as coisas (**t1**, **t2**, **t3**) em prateleiras, e entrega um bilhete, para usar para os acessar depois.

Este tipo de trabalho (contabilidade de onde estão os vetores **t1**, **t2**, **t3**, quais os seus tamanhos) **é trabalho para computador, não para gente.**

**Para isso existem ponteiros:**

- Quando se tem algo a guardar, entrega-se o conteúdo (**t1**) para ele. Ele encontra uma caixa vazia, coloca uma cópia de **t1** lá, e entrega o bilhete. O bilhete é o ponteiro, que é a informação que o guardador precisa para encontrar **t1** no futuro.
- **Independente da forma do que está na caixa, o bilhete tem sempre a mesma forma.** Então pode-se guardar um conjunto de bilhetes (ponteiros) em arrays. No caso, o array **t** teria 3 ponteiros, cada um apontando para um array diferente.
- Quando se precisa do conteúdo, mostra-se o ponteiro para o bibliotecário, que entrega uma cópia do conteúdo da caixa apontada por aquele ponteiro.
- Quando não interessar mais acessar **t1** de novo, pode-se dizer ao bibliotecário para apagar aquela cópia, liberando o espaço (memória) para uso futuro.
- **Isso é tudo que interessa para o usuário em alto nível.** É problema do compilador / interpretador saber como guardar as coisas, e o que significam as referências. **Não interessa ao usuário endereços de memória.**

# Ponteiros - exemplo - arrays irregulares

Cria-se um array para os **n=3** ponteiros a serem carregados (ex. Fortran):

```
!declare a type to make an array of pointers to reals  
type pointerarray_real  
  real, pointer :: p(:)  
end type pointerarray_real  
!declare a variable of that type, to hold the irregular array  
type(pointerarray_real), allocatable :: t(:)  
allocate(t(3))
```

Neste momento, os 3 ponteiros (**t(1)%p**, **t(2)%p**, **t(3)%p**) existem, mas **apontam para lugar nenhum**. São bilhetes em branco, para preencher quando algo for guardado.

Guarda-se os valores de cada linha em array apontado por cada ponteiro (após os alocar):

```
t(1)%p=(/7.9,5.8/) !first row  
t(2)%p=(/9.6,3.6,1.4,7.5/) !second row  
t(3)%p=(/3.2/) !third row
```

Agora os valores estão guardados no guarda-volumes.

Programa completo:

[http://www.ppenteado.net/ast/irr\\_array.f90](http://www.ppenteado.net/ast/irr_array.f90)

# Ponteiros - exemplo – programa completo

```

program irr_array
!example of making an irregular array in Fortran, through pointers
implicit none
!declare a type to make an array of pointers to reals
type pointerarray_real
  real, pointer :: p(:)
end type pointerarray_real
!declare a variable of that type, to hold the irregular array
type(pointerarray_real), allocatable :: t(:)
!declare the other variables
integer m,i
integer, allocatable :: np(:)
m=3 !set number of rows for the array
allocate(t(m),np(m)) !allocate the array t, and the array np with the number
of columns on each line
np=(/2,4,1/) !set values for the number of column each line has
!allocate each line of the array
do i=1,m
  allocate(t(i)%p(np(i)))
enddo
!set the values of the elements in the array
t(1)%p=(/7.9,5.8/) !first row
t(2)%p=(/9.6,3.6,1.4,7.5/) !second row
t(3)%p=(/3.2/) !third row
!now just show the array
do i=1,m
  write(6,*) 'row ',i,' contains'
  write(6,*) t(i)%p
enddo
end program irr_array

```

# Ponteiros - exemplo – resultado

Quando executado este programa, o resultado é

```
row          1  contains
  7.900000    5.800000
row          2  contains
  9.600000    3.600000    1.400000    7.500000
row          3  contains
  3.200000
```

# Objetos - conceitos

## Objetos são o próximo passo em abstração de tipos:

- **Inteiros**

- Um valor, representado diretamente pela sua representação binária.

- **Reais**

- Um valor, representado por um padrão onde vários valores (sinal, expoente, mantissa) são armazenados.

- Conhecido pelas rotinas, que sabem como operar sobre ele, inclusive os casos especiais (NaN, infinity, overflow, underflow, denormais).

- **Estruturas**

- Vários valores (campos) agrupados, identificados por nomes, em geral de tipos e dimensões diferentes.

- **Rotinas precisam saber especificamente o que fazer com os campos. Se recebem uma estrutura do tipo errado, podem fazer a coisa errada.**

# Objetos - conceitos

- **Objetos**

- Objetos são *instâncias* de *classes*.
- A classe é o **tipo** do objeto.
- Uma instância é um exemplo de um tipo:
  - **2** é uma instância do tipo **inteiro**
  - **1.0** é uma instância do tipo **real**
- São estruturas, com a adição das rotinas (*métodos*) que operam sobre eles.
- Em geral, **apenas os métodos da própria classe têm acesso aos seus dados (nos campos de sua estrutura)**.
  - Todo o acesso externo é feito por meio dos métodos, que podem controlar o acesso e garantir que propriedades dos campos são mantidas.
- **Rotinas não precisam saber identificar que tipo de variável estão usando e como as usar.**
- **As rotinas pertencem às classes, portanto elas sempre sabem o que são os campos e como os usar.**
- **Objetos são variáveis ativas**, não são só repositórios de dados: eles contêm dados e operam sobre eles (inclusive controlando o acesso aos dados).

# Para que são necessários objetos?

Programação orientada a objetos (OOP) foi a principal inovação da década de 1990.\*

Com relação à forma de organizar e pensar em código, são **o próximo nível de abstração, depois de programação estruturada**:

- **Programas não estruturados** – uma longa seqüência de operações sobre variáveis.
  - › A forma mais simples de pensar em um programa: como fazer a tarefa, por uma seqüência de operações.
  - › Não há modularidade; todas as variáveis são acessíveis de qualquer lugar.
  - › Reuso, manutenção, e verificação são difíceis.
- **Programas estruturados** - operações sobre variáveis são hierarquizadas em rotinas.
  - › A separação em unidades provê localidade de variáveis, e **facilita muito** testes, edição e reuso.
- **Programas orientados a objetos** - os objetos realizam as operações.
  - › **Objetos são variáveis, mas não são só receptáculos passivos.** Eles executam operações variadas - entre elas, armazenar, processar, e fornecer dados.
  - › **Rotinas são permanentemente associadas a tipos (classes)**, e ficam subordinadas às classes.
  - › **São as variáveis (objetos) que invocam as rotinas (métodos).**

\*Os conceitos de OOP começaram a ser desenvolvidos na década de 1950, (esp. Simula-67 e Smalltalk-76) mas só se tornaram populares com a chegada de C++ e Java.

# Classes – uso X criação / edição

**Programar com objetos** (usar classes) não é o mesmo que **programar objetos** (escrever classes).

Em programas simples, **é comum apenas usar as classes prontas** (da biblioteca padrão ou de terceiros), sem chegar a criar classes novas.

Mesmo nas linguagens mais orientadas a objetos, **é possível (às vezes, comum) que grande parte do código não use objetos**, ou que o programa seja procedural estruturado, com apenas alguns usos de objetos no meio.

**Raras linguagens forçam o programa a ser completamente orientado a objetos.**

Usar objetos é em geral simples (mais simples que a alternativa sem objetos).

**Como muita funcionalidade de muitas bibliotecas em muitas linguagens é fornecida por classes, é essencial conhecer sua semântica, para poder os usar.**

Criar do nada uma funcionalidade simples criando uma classe pode dar um pouco mais trabalho que o fazer apenas com rotinas simples, por ser necessário definir as variáveis e métodos que compõem a classe.

**Mas é mais fácil criar um resultado robusto, e principalmente, o usar, compartilhar, testar e modificar no futuro, usando objetos. E é muito mais fácil de criar quando é baseado em algo que já existe (*herança*, discutida adiante).**

# Classes – uso X criação / edição

Um tipo passivo (estrutura) nada faz. O usuário tem que juntar várias variáveis,

saber como as usar, e fazer o trabalho as usando juntas.

# Classes – uso X criação / edição

Um tipo passivo (estrutura) nada faz. O usuário tem que juntar várias variáveis,



saber como as usar, e fazer o trabalho as usando juntas.

Já um tipo ativo (classe) é capaz de fazer o trabalho. O usuário só tem que colocar a louça (dados) dentro, e a ligar:

# Classes – uso X criação / edição

Um tipo passivo (estrutura) nada faz. O usuário tem que juntar várias variáveis,



saber como as usar, e fazer o trabalho as usando juntas.

Já um tipo ativo (classe) é capaz de fazer o trabalho. O usuário só tem que colocar a louça (dados) dentro, e a ligar:

- Se alguém lhe dá uma lavadora, é muito menos trabalho para usar que se alguém lhe dá uma pia, esponja e sabão.

- Se tem que os criar, escrever uma lavadora dá mais trabalho que escrever uma pia, uma esponja e sabão.

- Mas em programação é necessário também escrever o código que faz o trabalho, usando a pia, esponja e sabão. O que faz o trabalho total de criação ser maior que o da lavadora. E a lavadora pode ser reusada no futuro de forma muito mais fácil.

- É comum que a classe seja criada por herança, aproveitando coisas já prontas, ou que sirva para ser herdada por outras coisas no futuro.



# Objetos - uso

Como os métodos (rotinas) são subordinados aos objetos, eles são **chamados com a semântica inversa da funcional**. Exs (IDL):

## Funcional:

IDL> `a=indgen(3)` Criação de um array (vetor) de 3 elementos

IDL> `n_a=n_elements(a)` Invocação de função: `n_elements()` recebe o argumento **a**

IDL> `plot, a` Invocação de procedimento: `plot` recebe o argumento **a**

## Objetos:

IDL> `l=list(0,1,2)` Criação

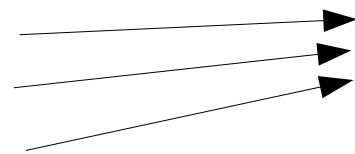
IDL> `n_l=l.count()` Invocação de função: o método `count()` do objeto **l**

IDL> `l.reverse` Invocação de procedimento: o método `reverse` do objeto **l**

# Objetos – algumas vantagens do seu uso

Não é necessário transportar, de uma só vez, muitas variáveis em chamadas de rotinas: **Os objetos carregam todos os dados, e os fornecem quando são necessários**, pois são estruturas. Como no exemplo anterior(Fortran):

```
dso%nlayers=nlayers
dso%degree=degree
dso%naz=naz
```



As dimensões do modelo são colocadas no objeto **dso**.

```
call dsv%disortset !aloca os componentes e ajusta os defaults em dsv,dsv
```

```
dso%tau=tau
dso%ssa=ssa
dso%pmom=pmom
dso%inccos=inccos(1)
dso%az=azang
```

```
write(6,*) 'calling disort'
```

```
call dso%disort_pp_run
```

Chama uma rotina (**disortset**) que pertence à classe (*type-bound*), através do objeto da classe (**dso**). Esta rotina só precisa saber como funciona esta classe: ela nunca vai ser chamada com objetos de outras classes.

Pode haver uma rotina **disortset** de outra classe, mas elas nunca vão se misturar: cada uma só é chamada na classe que a tem.

Os 5 parâmetros que não vão ser default são colocados na estrutura **dsv**

A rotina é chamada, sem precisar de argumentos: todos os valores de entrada e saída estão dentro do objeto **dso**.

# Objetos - quais são as vantagens do seu uso?

3 - **Mais expressividade e conveniência** por *overloading* de operadores / rotinas. Ex (IDL):

```
IDL> l1=list(4,9,16,25)
```

```
IDL> l2=list(36,49)
```

```
IDL> l3=l1+l2
```

```
IDL> help, l3
```

```
L3          LIST <ID=110  NELEMENTS=6>
```

```
IDL> print, l1 eq l2
```

```
0 0
```

```
IDL> l=list()
```

```
IDL> if (1) then print, 'list is true' else print, 'list is false'
```

```
list is false
```

```
IDL> l.add, 9
```

```
IDL> if (1) then print, 'list is true' else print, 'list is false'
```

```
list is true
```

Cada operador / rotina em vermelho foi *overloaded*: foi definido na classe o que significa os usar com objetos daquela classe.

# Objetos - quais são as vantagens ao os criar / modificar?

O último conceito fundamental de objetos é mais relevante ao criar / modificar uma classe:

**Herança (*inheritance*)** - Uma classe pode ser criada herdando uma (ou mais, em algumas linguagens) outra classe:

- Ao herdar uma classe, a **classe derivada** (ou **subclasse**) funciona exatamente como a classes que herdou (sua **superclasse**), apenas tendo outro nome.
- Ela **herda todos os métodos e todos os campos** (elementos de dados, como em estruturas) das superclasses. Objetos desta classe podem ser usados como se fossem das superclasses: **objetos da subclasse são também objetos da superclasse.**

Qual é a utilidade de herança? Só ter outra classe igual, de nome diferente, não é vantagem.

- Pode-se **escrever para a subclasse métodos adicionais** (não presentes nas superclasses), dando mais funcionalidade.
- Pode-se **escrever para a subclasse métodos com o mesmo nome de métodos das superclasses**; estes métodos *override* os métodos de mesmo nome da superclasse, e são chamados no lugar deles (a procura por um método com um nome começa da classe imediata, subindo pelas suas superclasses, até encontrar o primeiro método com o nome).

**É extremamente simples adicionar funcionalidade ou modificar o comportamento de uma classe, sem alterar o seu código:** herança nem precisa do código-fonte da superclasse.

**Uma superclasse pode ser usada para fornecer funcionalidade comum a várias classes diferentes**, que vão todas herdar esta classe; as subclasses contém só as partes que precisam variar entre elas.

# Referências (apenas algumas principais)

O quão recente é a referência costuma ser muito relevante.

- *Modern IDL*, de Michael Galloy (ainda não publicado; estimado para dez/2010)
- *Object-oriented programming with IDL*, de Ronn Kling
- *IDL Primer*, de Ronn Kling (menos atual)
- *Coyote's Guide to IDL Programming*  
<http://www.dfanning.com/> (algumas práticas são antiquadas)
- <http://michaelgalloy.com/>
- <http://groups.google.com/group/comp.lang.idl-pwwave/topics>  
(grupo ativo, lido pelos principais especialistas)
- *R in a Nutshell*, de Joseph Adler
- *Java in a Nutshell*, de David Flanagan
- *Programming in Python 3*, de Mark Summerfield
- *Python Scripting for Computational Science*, de Hans Petter Langtangen
- *Matplotlib for Python Developers*, de Sandro Tosi
- *The C++ Programming Language*, de **Bjarne Stroustrup**

# Referências (apenas algumas principais)

- *C Programming Language*, de W. Kernighan e **D. Ritchie**
- *Fortran 95/2003 Explained*, de Metcalf, Reid e Cohen
- *Introduction to Programming with Fortran: with coverage of Fortran 90, 95, 2003 and 77*, de I. D. Chivers (menos atual)
- *Programming Perl*, de **Larry Wall**, T. Christiansen, J. Orwant
- *Software Carpentry*  
*Helping scientists make better software since 1997*  
<http://software-carpentry.org/>
- *Comprehensive Perl Archive Network (CPAN)*  
<http://www.cpan.org/>
- *Comprehensive R Archive Network (CRAN)*  
<http://www.r-project.org/>
- Boost C++ libraries  
*“provides free peer-reviewed portable C++ source libraries.”*  
<http://www.boost.org/>
- Safari Books

Muitos livros sobre programação online (incluindo alguns citados acima). Podem ser comprados individualmente, por assinatura, ou **acessados pela assinatura da FAPESP**

<http://proquestcombo.safaribooksonline.com/>

# Organização de código – programming pearls

[http://users.erols.com/blilly/programming/Programming\\_Pearls.html](http://users.erols.com/blilly/programming/Programming_Pearls.html)

***If you can't write it down in English, you can't code it.***

Peter Halpern, Brooklyn, New York

*Less than 10% of the code has to do with the ostensible purpose of the system; the rest deals with input-output, data validation, data structure maintenance, and other housekeeping.*

Mary Shaw, Carnegie-Mellon University

*Don't write a new program if one already does more or less what you want. And if you must write a program, use existing code to do as much of the work as possible.*

Richard Hill, Hewlett-Packard S.A., Geneva, Switzerland

*Whenever possible, steal code.*

Tom Duff, Bell Labs

***[One Page Principle] A {specification, design, procedure, test plan} that will not fit on one page of 8.5-by-11 inch paper cannot be understood.***

Mark Ardis, Wang Institute

# Sumário

## Linguagens de programação

- Características de linguagens
- Escolha de linguagens

## Organização de código

- Estruturação
- Documentação

## Variáveis: além de números

- Estruturas
- Ponteiros
- Objetos

Mais detalhes sobre estes assuntos em <http://www.ppenteado.net/pea>

### HOW TO WRITE GOOD CODE:

