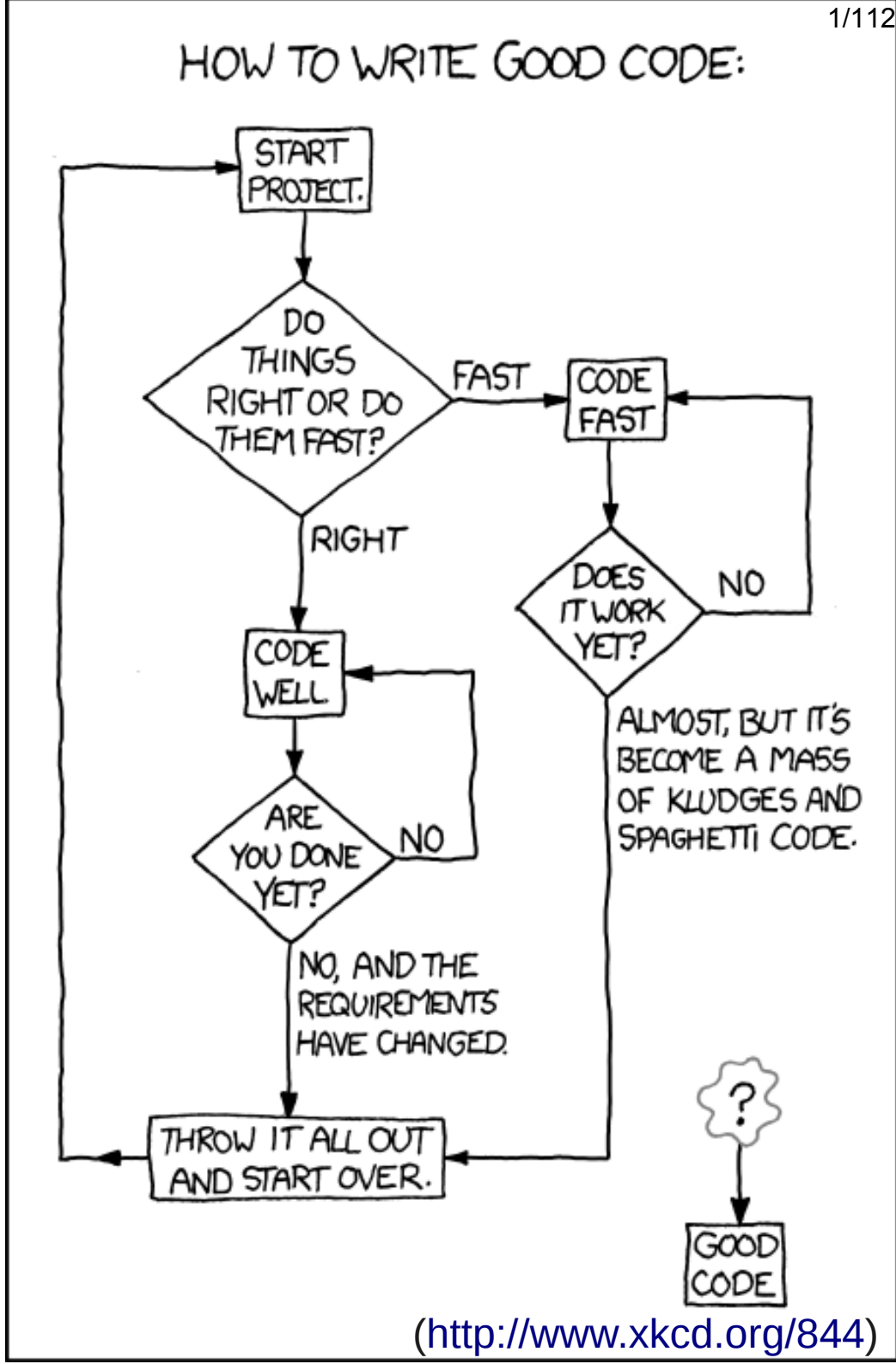
# Programação em astronomia: indo além de loops e prints

## 3 - Variáveis

Paulo Penteado

http://www.ppenteado.net/pea




(http://www.xkcd.org/844)

# Programa

1 – Slides em http://www.ppenteado.net/pea/pea01_linguagens.pdf
- Motivação
- Tópicos abordados
- Tópicos omitidos
- Opções e escolha de linguagens
- Uso de bibliotecas
- Referências

2 – Slides em http://www.ppenteado.net/pea/pea01_organizacao.pdf
- Organização de código
- Documentação
- IDEs
- Debug
- Unit testing

3 – Slides em http://www.ppenteado.net/pea/pea02_variaveis.pdf
- Tipos de variáveis
- Representações de números e suas conseqüências
- Ponteiros
- Estruturas
- Objetos

# Programa

4 – Slides em http://www.ppenteado.net/pea/pea03_conteiners.pdf
- Contêiners
- Arrays
- Listas
- Mapas
- Outros contêiners
- Vetorização
- Escolha de contêiners

5 – Slides em http://www.ppenteado.net/pea/pea04_strings_io.pdf
- Strings
- Expressões regulares
- Arquivos

# Tipos de variáveis

## 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 portável

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

# Tipos de variáveis

**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.)*

*Strings serão discutidos só na última aula

# 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 comuns

**Alguns tipos padrão (*standard, built-in, primitive*)*:**

**IDL**: int, string, float, double, byte, complex, ptr, obj

**Fortran**: integer, character, logical, real, double precision, complex, pointer

**C, C++, Java**: int, char, float, double

**Python**: int, long, float, complex, str

São tipos comuns em todas as linguagens, mas não necessariamente são os mesmos. Exs:

- Em IDL ***int*** é 16 bits, em Fortran ***integer*** costuma ser 32 bits (varia com o compilador)
- ***Real*** em Fortran pode ser 32 bits (precisão simples) ou 64 bits (precisão dupla)
- ***Float*** de Python em geral corresponde a ***double*** de outras (precisão dupla)

*Só os mais comuns; não são todos os tipos padrão destas linguagens

# Tipos comuns - variedades para idéias semelhantes

**Tipos não diferem só na "natureza" da abstração. Exs:**

- Diferem em tamanhos de inteiros:
  - **byte** (IDL): 8 bits, armazena inteiros de **0** a **255** ($2^8$-1)
  - **int** (IDL): 16 bits, armazena inteiros de -**32768** (-($2^{15}$)) a **32767** ($2^{15}$-1)
- Precisão de reais: **simples** (*single precision, float*) e **dupla** (*double*):
  - `1.0+1e-8 é 1.0` (32bits, 6 ou 7 dígitos significativos)
  - `1d0+1d-8 é 1.00000001` (64 bits, ~14 dígitos significativos)

**A mesma função/operador em geral é diferente com tipos diferentes (ex. IDL):**

- `3/2` é `1`, enquanto `3.0/2.0` é `1.5`
- `sqrt(-1.0)` é `-NaN`, enquanto `sqrt(complex(-1.0))` é `(0.0,1.0)`

# Tipos de variáveis – tipo vazio

Da mesma forma que o zero não existia nos sistemas numéricos mais primitivos, em linguagens de tipos dinâmicos existe o tipo vazio / nulo, de grande importância:

- **Para indicar a ausência de algo** (exemplos concretos na aula de contêiners):
  - Ex: em uma lista onde cada elemento tem os vizinhos de um objeto, para os objetos que não têm vizinhos.

- **Para indicar que não foram encontrados resultados.**
  - Ex: em uma função que retorna os elementos de um contêiner que satisfaçam a alguma propriedade, quando nenhum é encontrado.

- **Para variáveis / elementos não definidos**:
  - Indicar que estes devem ser substituídos por defaults (argumentos de entrada)
  - Indicar que não interessa calcular a coisa que dependa deles (argumentos de saída)

Em linguagens de tipos estáticos, um recurso comum à falta do tipo vazio é um ponteiro nulo (ponteiro que aponta para nada, discutido adiante).

Exempos:
- **!null** (IDL ≥8)
- **None** (Python)
- **NULL** (R, C++, Perl)
- **null** (Java)

# Representações de números e suas conseqüências

**Números em variáveis não são o mesmo que o conceito matemático abstrato de números.**

**Uma variável tem um espaço limitado** de memória. Por isso o número de dígitos é limitado:

- A quantidade de números diferentes que podem ser representados é finita.
- Os números que podem ser representados são predeterminados pelo modelo usado.
- A precisão dos números e a faixa de valores que podem assumir são limitados.

Tipos numéricos básicos (inteiros, reais) das linguagens costumam ser os mesmos que os tipos suportados nativamente pelos processadores.

**Em geral tipos têm tamanhos fixos na memória.** Exs: 8, 16, 32, 64 bits (1, 2, 4, 8 bytes).

Cada bit (*binary digit*) é uma posição na memória, que só pode ter o valor **0** ou **1**.

A quantidade de valores diferentes que um tipo de **n** bits pode representar é $\mathbf{2^n}$.

Os tipos mais comuns representam **256**, **65536**, $\mathbf{\sim 4.3 \times 10^9}$ (4 giga, no sentido binário), ou $\mathbf{\sim 1.8 \times 10^{19}}$ (16 exa, no sentido binário) valores diferentes.

# Representações de números - inteiros

Há tipos só para inteiros positivos (*unsigned*) e tipos para negativos e positivos.

**Positivos são simplesmente o número escrito em binário.**

**Ex: com 8 bits, só há os números de 0 a 255:**

| Decimal | representação na memória |
|---|---|
| 0 | 00000000 |
| 1 | 00000001 |
| 2 | 00000010 |
| 255 | 11111111 |

Tipos que aceitam negativos são feitos (em geral) dividindo-se os números representáveis aproximadamente à metade, com os positivos no começo, e os negativos no final:

**Ex: com 8 bits, só há os números -128 a +127:**

| Decimal | representação na memória |
|---|---|
| 0 | 00000000 |
| 1 | 00000001 |
| 127 | 01111111 |
| -128 | 10000000 |
| -127 | 10000001 |
| -126 | 10000010 |
| -2 | 11111110 |
| -1 | 11111111 |

# Representações de inteiros - nomes e tamanhos comuns*

**8 bits:**
- byte (IDL, só positivos)
- byte (Java)
- char (C, C++)

**16 bits:**
- int (IDL)
- short int (C++)
- short (Java)

**32 bits:**
- integer (Fortran, R)
- long (IDL)
- int (C, C++, Java, Python)
- long int (C, C++)

**64 bits:**
- long (Python)
- long64 (IDL)

Estes costumam ter os equivalente só para positivos, nomeados com ***unsigned*** (***unsigned int***) ou apenas ***u*** (***uint***).

*Em algumas linguagens, o padrão não especifica o tamanho de cada nome, e cada implementação pode escolher o que usar; os listados são os mais comuns.

# Representações de inteiros - nomes e tamanhos comuns

Literais* costumam ter um tipo default, e podem ser de outros com modificadores (exs. IDL):

- 9 9 do tipo default de inteiro
- 8L 8 do tipo *long* (32 bits)
- 25B 25 do tipo *byte* (8 bits)
- 12UL 12 do tipo *unsigned long* (64 bits)

Algumas linguagens têm tipos de inteiros decimais, e de inteiros de qualquer tamanho (precisão arbitrária):

- **São extremamente ineficientes**
- Só devem ser usados quando estritamente necessários
- Servem predominantemente para teoria de números (ex: criptografia), e alguns usos financeiros.

*constantes que aparecem *literalmente* dentro do código

# Representações de números - conseqüências (inteiros)

O que acontece quando se tenta colocar em uma variável um número que não cabe nela?

- Em um tipo *byte*, que só comporta de **0** a **255**, quanto vale **255B+1B**? E **0B-1B**?
- Em um tipo *short*, que só comporta de **-32768** a **+32767**, quanto vale **-32767S-2S**?

# Representações de números - conseqüências (inteiros)

O que acontece quando se tenta colocar em uma variável um número que não cabe nela?

- Em um tipo *byte*, que só comporta de **0** a **255**, quanto vale **255B+1B**? E **0B-1B**?
- Em um tipo *short*, que só comporta de **-32768** a **+32767**, quanto vale **-32768S-1S**?

Ocorre um *overflow* (ou *rollover*). Como um odômetro de carro (ex. IDL):

```
IDL> a=255B
IDL> help,a
A               BYTE      =  255

IDL> a=a+1B
IDL> help,a
A               BYTE      =    0

IDL> print,0B-1B
 255

IDL> print,-32768S-1S
   32767
```

Internamente (binário):

```
 11111111

 11111111
+00000001
100000000
=00000000

1000000000000000
-0000000000000001
=0111111111111111
```

# Representações de números - conseqüências (inteiros)

**Não considerar o tamanho dos números é um erro comum.**

Ex: Em IDL, onde inteiros default são do tipo int (16 bits):

```
IDL> print,10^4
   10000

IDL> print,10^5
?
```

# Representações de números - conseqüências (inteiros)

**Não considerar o tamanho dos números é um erro comum.**

Ex: Em IDL, onde inteiros default são do tipo int (16 bits):

```
IDL> print,10^4
   10000

IDL> print,10^5
  -31072
```

**O resultado de $10^5$ não está errado:**

- **$10^5$** é maior que o maior inteiro que cabe em 16 bits (**32767**)
- Depois de **32767** vem **-32768**, depois **-32767**, etc.

**Com um tipo maior, não há overflow para este número:**

```
IDL> print,10L^5
      100000
```

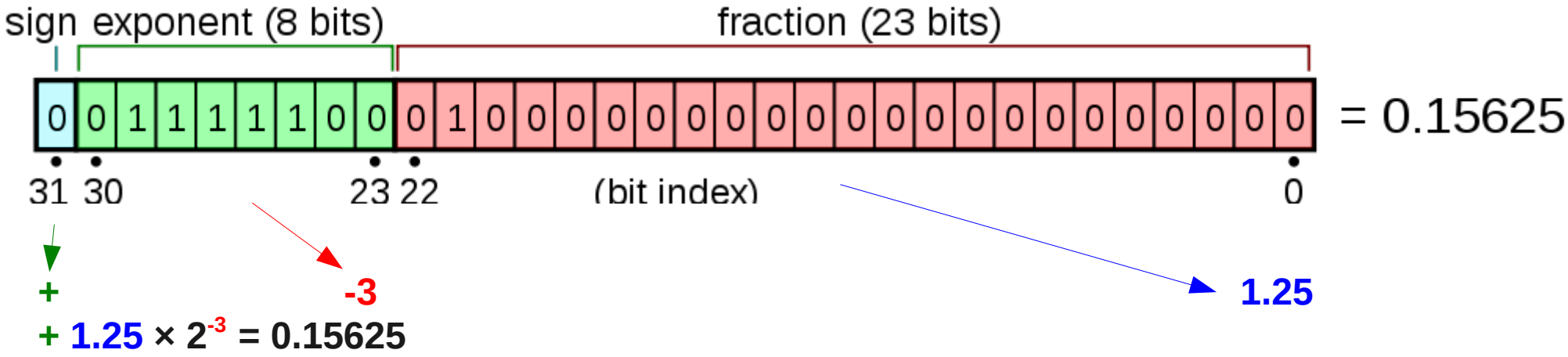
# Representações de números - reais

Tipos usuais são do padrão IEEE 754 para representação e manipulação de números por ponto flutuante:

- ***single precision / float / real*** (32 bits)
- ***double precision / double*** (64 bits)

Os números são representados por uma mantissa e um expoente, de forma análoga a notação científica (ex: **0.31416E1**), mas com dígitos binários.

Divididos em 3 partes: **sinal**, **mantissa** e **expoente**:




| | Sinal | Expoente | Mantissa | Faixa | Dígitos decimais significativos |
|---|---|---|---|---|---|
| Float | 1 bit | 8 bits | 23 bits | ~$\pm 10^{38}$ | 6 a 7 |
| Double | 1 bit | 11 bits | 52 bits | ~$\pm 10^{308}$ | ~14 |
| Quad* | 1 bit | 15 bits | 112 bits | ~$\pm 10^{4932}$ | ~33 |

*Raramente implementado

# Representações de números - reais

***Floats* (precisão simples) ainda são o tipo default mais comum**, por motivos históricos.

**Mas para quase todas as aplicações científicas é insuficiente.**

**Literais como 1.0 e 1e5 são normalmente interpretados como *floats*.**

***Doubles* exigem escrever literais como 1.0d0 e 1d5** (o *d* é usado no lugar do *e* que indica a potência de 10).

**Atenção aos tipos usados em literais:** nenhum software pode adivinhar se um número escrito como um tipo tinha a intenção de ser outro (exs. IDL):

```
IDL> print,1/3
       0

IDL> print,1.0/3.0
     0.333333

IDL> print,1d0/3d0
      0.33333333

IDL> print,16d0^(1/2)
       ?
```

# Representações de números - reais

***Floats* (precisão simples) ainda são o tipo default mais comum**, por motivos históricos.

**Mas para quase todas as aplicações científicas é insuficiente.**

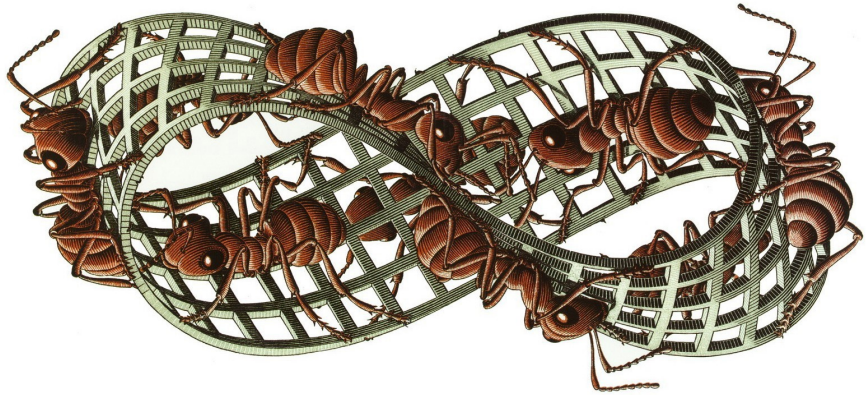
**Literais como 1.0 e 1e5 são normalmente interpretados como *floats*.**

***Doubles* exigem escrever literais como 1.0d0 e 1d5** (o *d* é usado no lugar do *e* que indica a potência de 10).

**Atenção aos tipos usados em literais:** nenhum software pode adivinhar se um número escrito como um tipo tinha a intenção de ser outro (exs. IDL):

```
IDL> print,1/3
       0

IDL> print,1.0/3.0
     0.333333

IDL> print,1d0/3d0
      0.33333333

IDL> print,16d0^(1/2)
       1.0000000

IDL> print,16d0^(1d0/2d0)
       4.0000000
```

# Representações de números: +*Infinity* e -*Infinity*

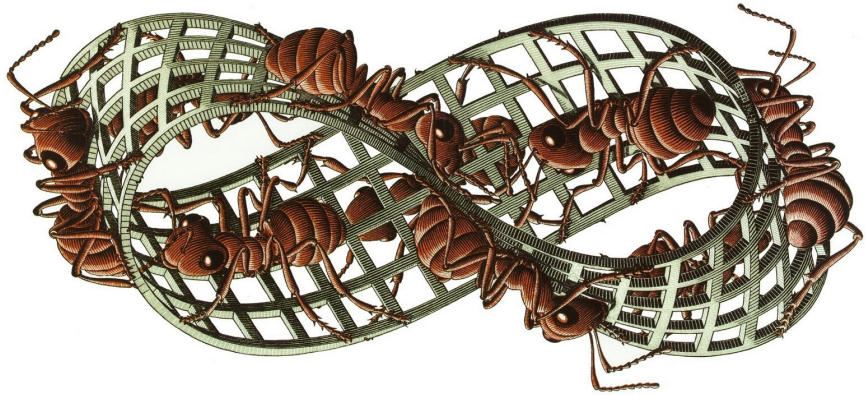


# Representações de números: +*Infinity* e -*Infinity*

Retornados em muitas funções e overflows (número muito grande para o tipo). Exs (IDL):

```
IDL> help,1.0/0.0
<Expression>    FLOAT     =           Inf
% Program caused arithmetic error: Floating divide by 0

IDL> print,exp(-!values.f_infinity)
      0.00000

IDL> help,atan(1d0/0d0)/!dpi
<Expression>    DOUBLE    =       0.50000000
% Program caused arithmetic error: Floating divide by 0

IDL> print,!values.d_infinity gt 0. ;Infinity is larger than 0
   1

IDL> print,!values.d_infinity eq !values.d_infinity ;Inf is equal to itself
   1
IDL> print,exp(89.0)
          Inf
% Program caused arithmetic error: Floating overflow
```

Apenas avisos; não são erros.



# Representações de números: *+NaN* e *-NaN*

# Representações de números: *+NaN* e *-NaN*


**N**ot **a** **N**umber

Resultados inválidos. Exs (IDL):

```
IDL> help,0./0.
<Expression>    FLOAT     =           -NaN
% Program caused arithmetic error: Floating illegal operand

IDL> help,!values.d_infinity/!values.d_infinity
<Expression>    DOUBLE    =              -NaN
% Program caused arithmetic error: Floating illegal operand

IDL> print,sqrt(-1d0)
             NaN
% Program caused arithmetic error: Floating illegal operand

IDL> print,sqrt(complex(-1d0))
(      0.00000,      1.00000)

IDL> print,!values.d_nan gt 0d0 ;NaN não é maior que qualquer coisa
   0

IDL> print,!values.d_nan le 0d0 ;NaN não é menor que qualquer coisa
   0

IDL> print,!values.f_nan eq !values.f_nan ;NaN não é igual a si mesmo
   0
```

Apenas avisos; não são erros.

# Representações de números: *+NaN* e *-NaN*

Uso comum para indicar dados faltantes ou que não fariam sentido.Ex:

- Pixels com defeito ou onde foram detectados problemas.

- Medidas que não foram tomadas ou consideradas:
  - área do céu ainda não observada
  - medida não feita para o objeto
  - região não incluída no modelo

**Melhor que a prática comum de escolher um valor arbitrário, como -1 ou 0:**

Um valor “especial” depende de saber que aquele valor significa algo especial.
E se não houver um valor que seja especial (se qualquer número fizer sentido)?

Muitas rotinas ignoram **NaNs** em dados de entrada:

- Deixam um buraco na linha em um gráfico.

- Os ignoram quando aparecem em um array, ao calcular máximo, mínimo, etc.

**Na maior parte das operações com NaN o resultado (apropriadamente) é NaN:**

- Adicionar um número a uma imagem não deve magicamente transformar os pixels inválidos (NaNs) em algum número.
- Multiplicar um número por uma imagem não deve magicamente transformar os pixels inválidos (NaNs) em algum número.
- **NaN não é 0, 1, ou qualquer outro elemento neutro.**

# Representações de números: zeros (reais)

**Há dois zeros (+0 e -0):**

São iguais em comparações, mas mostram sua diferença em limites.

Exs. (IDL):

```
IDL> print,1d0/0d0
         Infinity
% Program caused arithmetic error: Floating divide by 0


IDL> print,1d0/(-0d0)
        -Infinity
% Program caused arithmetic error: Floating divide by 0


IDL> print,0d0 eq -0d0
   1
```

Apenas avisos; não são erros.

# Representações de números - reais - conseqüências

Assim como para inteiros, também é necessário considerar os **limites de faixas de valores** de reais. E reais também têm **limite de precisão**. Exs. (IDL):

```
IDL> print,exp(-103.)
  1.40130e-45
% Program caused arithmetic error: Floating underflow

IDL> print,exp(-104.)
% Program caused arithmetic error: Floating underflow
      0.00000

IDL> print,1.0,1e-8,1.0+1e-8
      1.00000  1.00000e-08       1.00000

IDL> print,(1.0+1e-8) eq 1.0
   1

IDL> print,1.0d0,1d-8,1.0d0+1d-8
       1.0000000   1.0000000e-08       1.0000000

IDL> print,(1.0d0+1d-8) eq 1.0d0
   ?
```

Apenas avisos; não são erros.

# Representações de números - reais - conseqüências

Assim como para inteiros, também é necessário considerar os **limites de faixas de valores** de reais. E reais também têm **limite de precisão**. Exs. (IDL):

```
IDL> print,exp(-103.)
  1.40130e-45
% Program caused arithmetic error: Floating underflow

IDL> print,exp(-104.)
% Program caused arithmetic error: Floating underflow
      0.00000

IDL> print,1.0,1e-8,1.0+1e-8
      1.00000  1.00000e-08       1.00000

IDL> print,(1.0+1e-8) eq 1.0
   1

IDL> print,1.0d0,1d-8,1.0d0+1d-8
       1.0000000   1.0000000e-08       1.0000000

IDL> print,(1.0d0+1d-8) eq 1.0d0
   0
```

Apenas avisos; não são erros.

# Representações de números - reais - conseqüências

Os dígitos mostrados quando um real é impresso não necessariamente se estendem até o último dígito significativo.

**Podem terminar antes ou depois**, dependendo de como o número foi impresso.

**Os formatos default costumam usar menos dígitos que a precisão do tipo.** Ex. (IDL):

```
IDL> print,1.0d0+1d-8
       1.0000000

IDL> print,1.0d0+1d-8,format='(E22.15)'
 1.00000001000000E+00
```

Como a representação é binária, **apenas os números que são racionais em binário** (somas de potências de 2) podem ser representados exatamente. Ex. (IDL):

```
IDL> print,1.0,0.1,0.7
      1.00000      0.100000      0.700000

IDL> print,1.0,0.1,0.7,format='(3E19.9)'
        ?                ?                ?
```

# Representações de números - reais - conseqüências

Os dígitos mostrados quando um real é impresso não necessariamente se estendem até o último dígito significativo.

**Podem terminar antes ou depois**, dependendo de como o número foi impresso.

**Os formatos default costumam usar menos dígitos que a precisão do tipo.** Ex. (IDL):

```
IDL> print,1.0d0+1d-8
       1.0000000

IDL> print,1.0d0+1d-8,format='(E22.15)'
 1.00000001000000E+00
```

Como a representação é binária, **apenas os números que são racionais em binário** (somas de potências de 2) podem ser representados exatamente. Ex. (IDL):

```
IDL> print,1.0,0.1,0.7
      1.00000      0.100000      0.700000

IDL> print,1.0,0.1,0.7,format='(3E19.9)'
    1.000000000E+00    1.000000015E-01    6.999999881E-01
```

# Representações de números - reais - conseqüências

**Em ciências computacionais, em geral a precisão de *floats* é insuficiente:**

- **Inverter uma matriz costuma não funcionar em precisão simples** (a matriz parece singular quando não é).

- Mesmo que os dados não tenham precisão melhor que 6 dígitos (em geral não têm), é necessário precisão dupla, pois **operações sucessivas podem acumular erros muito grandes**.

- Com freqüência aparecem números com potências além de ±38:
  - $h=6.62\times10^{-34}$ J×s
  - $M_{\odot}=1.99\times10^{33}$g

- Datas julianas precisam de muitos dígitos (1 s é $1.16\text{x}10^{-5}$ d).
  - Ex: **2455563.024502**
  - 7 dígitos para precisão de 1 dia
  - + 5 dígitos para precisão de ~1s

# Representações de números - reais - conseqüências

Tipos **inteiros têm mais dígitos significativos** (mas menor faixa de valores) que reais:

- Em um inteiro de 32 bits sem sinal cabem exatamente todos os números entre **0** e **4294967295** (4 giga -1 , no sentido binário).
- Em um real de 32 bits, cabem números até ~$10^{38}$, mas **4294967295** não existe (precisaria de 10 dígitos decimais de precisão).

**Exs. (IDL):**

```
IDL> a=4294967295UL

IDL> print,a,format='(I0)' & print,float(a),format='(F0)'
4294967295
4294967296.000000

IDL> print,a-25, format='(I0)' & print,float(a-25),format='(F0)'
4294967270
4294967296.000000
```

Comparando os tipos de 64 bits:

```
IDL> a=123456789012345678 90ULL

IDL> print,a,format='(I0)' & print,double(a),format='(F0)'
12345678901234567890
12345678901234567168.000000
```

# Representações de números - reais - conseqüências

**Atenção a comparações com valores reais.** Exs (IDL):

```
IDL> a=dindgen(3)*!dpi
```

Gera um array com elementos 0π, 1π, 2π

```
IDL> print,a
       0.0000000       3.1415927       6.2831853
```

```
IDL> print,where(a eq 3.1415927,/null)
!NULL
```

Nenhum elemento é igual a **3.1415927**

```
IDL> print,where(a eq !dpi,/null)
           1
```

O elemento **1** é igual a **!dpi**

# Representações de números - reais - conseqüências

**Atenção a comparações com valores reais.** Exs (IDL):

```
IDL> a=dblarr(100000)+!dpi

IDL> b=(total(a)/n_elements(a))

IDL> print,b
       3.1415927

IDL> print,b eq !dpi
   ?
```

**a** é um array com 100000 elementos iguais a **!dpi**

**b** é a soma dos elementos de a, dividida pelo número de elementos de a

**b** é igual a **!dpi** (?)

# Representações de números - reais - conseqüências

**Atenção a comparações com valores reais.** Exs (IDL):

```
IDL> a=dblarr(100000)+!dpi

IDL> b=(total(a)/n_elements(a))

IDL> print,b
       3.1415927

IDL> print,b eq !dpi
   0

IDL> print,b - !dpi
  -2.6694202e-12
```

**a** é um array com 100000 elementos iguais a **!dpi**

**b** é a soma dos elementos de a, dividida pelo número de elementos de a

**b** é igual a **!dpi** (?)

Não!

Em geral, só se pode esperar que reais sejam iguais **se um é uma cópia do outro, e nenhum processamento foi aplicado** a eles.

**Até a associatividade pode deixar de ser verdade:** A+(B+C) pode ser diferente de (A+B)+C.

Resultados podem não ser idênticos, mesmo com dados idênticos, em:

- Implementações diferentes do mesmo algorítimo.
- Execuções diferentes do mesmo programa paralelo.

# Exercícios (perguntas reais, do newsgroup do IDL)

*1)*
*This may be a stupid question, but I really want to know why.*
*Please, see below and explain. Thanks.*
*IDL> print, 132*30*
*3960*
*IDL> print, 132*30*10*
*-25936*

*2)*
*There's something I can not explain to myself, so maybe someone can enlighten me?*
*IDL> print, fix(4.70*100)*
*469*

*To try and find where the problem is, we tried the following lines:*
*IDL> a = DOUBLE(42766.080001)*
*IDL> print,a,FORMAT='(F24.17)'*
*42766.07812500000000000*
*As you see, the number we get out isn't the same as the number we entered.*

*3)*
*I have a problem related to float-point accuracy*
*If I type in: 50d - 1d-9, I get 50.000000*
*And here lies my problem, I'm doing a numerical simulation where such an arithmetic is common place, and as a result i get a lot or errors. I know for example, that if i simply type print, 50d - 1d-9, format = '(f.20.10)' , i'll get:*
*49.9999999990*
*But how can I convince IDL to do it on its own during computations?*

# Exercícios (perguntas reais, do newsgroup do IDL)

*4)*
*Hi guys,*
*IDL> print,((10^5)/(exp(10)*factorial(5)))*
*The actual result of the above line is 0.0378332748*
*But when we run it in IDL we get the result as -0.011755556*

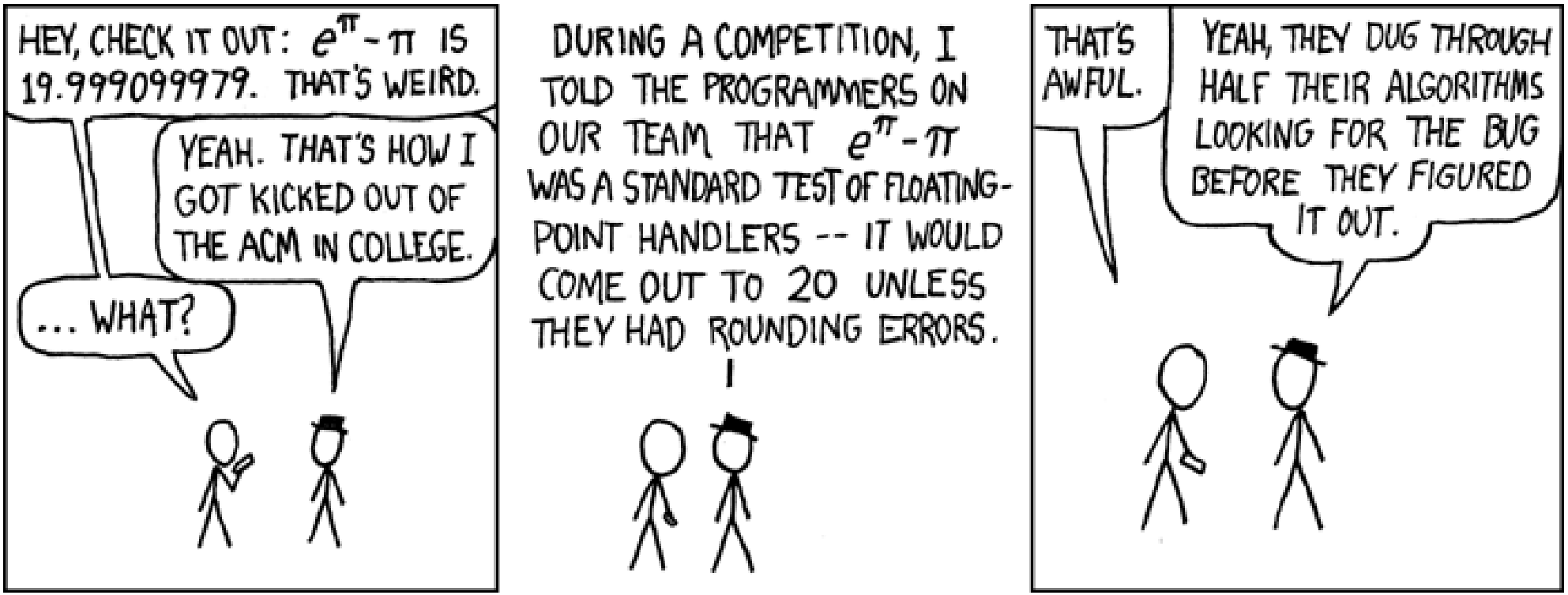
*5)*
*I ran into a number transformation error yesterday that is still confusing me this morning. The problem is that the number 443496.984 is being turned into the number 443496.969 from basic assignments using Float() or Double(), despite the fact that even floats should easily be able to handle a number this large (floats can handle "±10^38, with approximately six or seven decimal places of significance").*

# Representações de números - referências

*Help! The sky is falling!*
http://www.dfanning.com/math_tips/sky_is_falling.html

*What every programmer should know about floating-point arithmetic*
*or*
*Why don't my numbers add up?*
http://floating-point-gui.de/

*What every computer scientist should know about floating-point arithmetic*
http://docs.sun.com/source/806-3568/ncg_goldberg.html




http://www.xkcd.org/217

# 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?

## 2 exemplos de soluções (ex. IDL):

### 1 - Um vetor de dois reais, o primeiro para a parte real, o segundo para a imaginária

- Definição:
  - `a=[1.0,0.0]`
- Adição e subtração seriam automáticas, mas seria necessário escrever funções para todo o resto, inclusive multiplicação e divisão
  - `b=csqrt(a)`, depois de definir a função `csqrt`
- Como usa um vetor para os dois elementos, **sempre adiciona uma dimensão**:
  - Um escalar complexo seria um vetor de 2 elementos
  - Um vetor de n complexos seria um array:
    - **(2,n) ou (n,2)?** Seria necessário cuidado com a ordem das dimensões
  - Um array de (n,m) complexos seria um array:
    - **(2,n,m) ou (n,m,2)**: fica mais desajeitado ainda, mais cuidado é necessário com a ordem
- O código que os usa teria que seguir a convenção (parte real seguida da imaginária). **Não haveria como saber se uma variável é um complexo ou um vetor de dois reais.**

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

**2 - Variáveis separadas:**

- Definição:
  - `ar=1.0 & ai=0.0`
- As rotinas precisariam de 2 argumentos para entrada, e (normalmente) 2 para a saída:
  - Não poderiam ser funções simples (retornando apenas uma variável)
  - Seriam desajeitadas
    - `csqrt,ar,ai,br,bi` (muito pior que `csqrt(a)`, da solução 1)
  - Não criam dimensões a mais, mas dobram o número de variáveis.
- **O código tem que seguir a convenção das duas variáveis.** É necessário manter os pares juntos, sempre sincronizados (não mudar só a parte real, por exemplo).

**Ambas as soluções são desajeitadas.**

**O que seria melhor?**

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

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**
- **Agrupa 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):

IDL, com nome: `a={complexo,real:1.0,imag:0.0}`

IDL, anônimo: `a={re:1.0,im:0.0}`

Definição da estrutura e de uma variável ao mesmo tempo

Fortran:

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

Definição do tipo

Definição de uma variável daquele tipo

**Uso**:

IDL:

```
a.re=2.0
a.im=3.0
```

Fortran:

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

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

Exemplo de função (IDL):

```
function cmultip,a,b
  ret=a ;apenas para criar uma variável do tipo certo
  ret.re=a.re*b.re-a.im*b.im ;parte real do produto
  ret.im=a.re*b.im+a.im*b.re ;parte imaginária do produto
  return,ret
end
```

Que poderia então ser usada como:

```
IDL> a={re:1.0,im:0.0}

IDL> b={re:0.0,im:1.0}

IDL> c=cmultip(a,b)

IDL> print,c
{      0.00000      1.00000}

IDL> help,c
** Structure <18c5268>, 2 tags, length=8, data length=8, refs=2:
   RE              FLOAT           0.00000
   IM              FLOAT           1.00000
```

# 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:

```
a={r:0.0,theta=0.0,phi=0.0}
b=replicate(a,n)                ;vetor de n cópias de a
b.r=r ;equivalente a 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 - uso para agrupar variáveis

**Mais importante se forem muitas as variáveis**. Exs (IDL):

Uma função que use 3 pontos em 3D (para calcular o plano que passa pelos 3):

Definindo

```
p={p,x:0.,y:0.,z:0.}
```

Poderia ser usada a função

```
plano1(p1,p2,p3)
```

Com **`p1,p2,p3`** de tipo **`p`**:

- `p1.x,p1.y,p1.z` sendo `x,y,z` de `p1`
- semelhante para `p2,p3`.

Ou ainda

```
plano2(ps)
```

Com **`ps`** sendo um vetor de 3 elementos do tipo **`p`**

- **`ps=[p1,p2,p3]`**

# Estruturas - uso para agrupar variáveis

Ambas as soluções anteriores são melhores que usar todos os nove valores separados:

```
plano3(x1,y1,z1,x2,y2,z2,x3,y3,z3)
```

Ainda seria fácil confundir a ordem com

```
plano3(x1,x2,x3,y1,y2,y3,z1,z2,z3)
```

Também melhor que um vetor `pv(3,3)`:

```
plano4(pv)
```

Pois **`x,y,z`** do ponto 1 seriam **`pv[0,*]`**, ou **`pv[*,0]`**? Fácil confundir.

E se no lugar de **`n`** triplas fossem **`(n,m)`** triplas?

- Um array **`(n,m,3)`** ou **`(3,n,m)`**?
- Poderiam ser usadas com **`plano2`** em um simples array de **`(n,m)`** triplas
- Em algumas linguagens (como IDL) nem seria necessário reescrever **`plano2`** para aceitar arrays no lugar de escalares

# Estruturas - exemplo - agrupar tipos diferentes: read_ascii (IDL)

**É simples ler uma tabela só de números como:**

| wavl | CH4=3.3 | wavl | CH4=2.5 | wavl | CH4=1.0 | wavl | CH4=0.8 | wavl | CH4=0.5 | wavl | CH4=0.26 | wavl | CH4=0.2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 477.330000 | 0.090130 | 477.330000 | 0.091110 | 477.330000 | 0.089250 | 477.330000 | 0.087000 | 477.330000 | 0.087140 | 477.330000 | 0.090080 | 477.330000 | 0.088110 |
| 480.040000 | 0.090930 | 480.040000 | 0.091930 | 480.040000 | 0.090160 | 480.040000 | 0.087930 | 480.040000 | 0.088110 | 480.040000 | 0.090950 | 480.040000 | 0.089090 |
| 482.750000 | 0.091710 | 482.750000 | 0.092730 | 482.750000 | 0.091060 | 482.750000 | 0.088850 | 482.750000 | 0.089080 | 482.750000 | 0.091810 | 482.750000 | 0.090060 |
| 485.450000 | 0.092530 | 485.450000 | 0.093570 | 485.450000 | 0.092000 | 485.450000 | 0.089810 | 485.450000 | 0.090100 | 485.450000 | 0.092730 | 485.450000 | 0.091090 |

Apenas com

```
a=read_ascii('specs_27s_n.txt',data_start=1,header=header)
```

O que resulta em a contendo um array **(14,4)** e a linha de cabeçalho na variável header.

Mas e um arquivo com colunas de texto, inteiros e reais como:

| NAME | CALMPOS | FILNAME | ECHLPOS | DISPPOS | TARGNAME | POSDIR | CLASS | MJD-OBS | ITIME | COADDS |
|---|---|---|---|---|---|---|---|---|---|---|
| dec18s0001 | 0 | NIRSPEC-5-AO | 62.6300 | 36.4500 | HD85258 | NIRSPEC-5-AO/p1 | STAR | 54087.57421875 | 100.00000 | 1 |
| dec18s0002 | 0 | NIRSPEC-5-AO | 62.6300 | 36.4500 | HD85258 | NIRSPEC-5-AO/p1 | STAR | 54087.57421875 | 100.00000 | 1 |
| dec18s0003 | 1 | NIRSPEC-5-AO | 62.6300 | 36.4500 | HD85258 | NIRSPEC-5-AO/p1 | FLAT | 54087.57812500 | 4.60000 | 5 |
| dec18s0004 | 1 | NIRSPEC-5-AO | 62.6300 | 36.4500 | HD85258 | NIRSPEC-5-AO/p1 | DARK | 54087.57812500 | 4.60000 | 5 |

**Não dá para fazer um array, porque em um array todas as colunas teriam que ter o mesmo tipo.**

11 variáveis, uma para cada coluna? Desajeitado.

Solução: **estruturas**.

# Estruturas - exemplo - agrupar tipos diferentes: read_ascii (IDL)

Por exemplo, a coluna com o nome do objeto, a coluna 6, está em:

```
IDL> print,a.field06
HD85258 HD85258 HD85258 HD85258
```

Strings

Já o tempo de integração, a coluna 10, está em:

Floats

```
IDL> print,a.field10
      100.000      100.000      4.60000      4.60000
```

# Estruturas - exemplo - agrupar tipos diferentes: read_ascii (IDL)

O arquivo é lido em uma estrutura

```
IDL> a=read_ascii('filesearch_scam.txt',template=templ)
```

(mais detalhes sobre a leitura na aula sobre arquivos; aqui só interessa o resultado)

```
IDL> help,a,/structure
** Structure <19e5368>, 11 tags, length=416, data length=416, refs=1:
   FIELD01         STRING    Array[4]
   FIELD02         LONG      Array[4]
   FIELD03         STRING    Array[4]
   FIELD04         FLOAT     Array[4]
   FIELD05         FLOAT     Array[4]
   FIELD06         STRING    Array[4]
   FIELD07         STRING    Array[4]
   FIELD08         STRING    Array[4]
   FIELD09         FLOAT     Array[4]
   FIELD10         FLOAT     Array[4]
   FIELD11         LONG      Array[4]
```

É uma estrutura, com um campo para cada coluna.

Cada campo é um vetor de 4 elementos, um com o valor daquela coluna em cada linha.

# Estruturas - exemplo - agrupar tipos diferentes: read_ascii (IDL)

**read_ascii** empacotou todas as 11 variáveis dentro de uma única variável.

Se a rotina read_ascii para funcionasse com qualquer arquivo usando uma variável separada para cada coluna, ela teria um número grande e variável de argumentos (nesse caso, 12), e de tipos diferentes:

- Não seria elegante.
- O programa ficaria com 11 variáveis no lugar de uma, que precisariam de nomes a inventar, e encheriam o programa de variáveis.
- É comum se precisar ler arquivos assim onde só uma ou duas colunas interessam; várias variáveis seriam inventadas para não ser usadas.
- Este exemplo tinha poucas colunas, para caber na largura da página da apresentação. Usar 11 variáveis na mão seria ruim, mas factível. E se fossem 200 colunas?

Também é possível, em IDL, acessar os elementos de uma estrutura pela ordem. No caso,

- `a.field01` é equivalente a `a.(0)`
- `a.field02` é equivalente a `a.(1)`

Rotinas de *Introspecção* dão informações sobre os campos (número, nomes, tipos, dimensões).

# Estruturas - exemplo – agrupamento de muitas variáveis

Como ler vários arquivos que contém 35 arrays cada, que dependem de 14 dimensões diferentes, com um só comando e sem criar 49 variáveis por arquivo?

```
$ ncdump -h refspec_g01_0.nc
netcdf refspec_g01_0 {
dimensions:
        nlay = 51 ;
        nwn = 400 ;
        nleg = 33 ;
        numu = 2 ;
        nlev = 52 ;
        nphi = 3 ;
        ngas = 2 ;
        nwnc = 1 ;
        scal = 1 ;
        v3 = 3 ;
        dnl = 1 ;
        nk = 1 ;
        tdisr = UNLIMITED ; // (0 currently)
        na = 16 ;
variables:
        float alb(nwn) ;
        float z(nlay) ;
        float t(nlay) ;
        float p(nlay) ;
        float wl(nwn) ;
        float iof(nwn) ;
        float mtau(nwn) ;
        float htau(nwn) ;
        float hctaus(nwn) ;
        float gtau(nwn) ;
        float outcos(scal) ;
        float phi(nleg) ;
        float phic(scal) ;
        float umu(numu) ;
        float flux(v3, nwn, nlev) ;
        float mix(nlay, ngas) ;
        float c(nlay) ;
        float psat(nlay) ;
        float ga(nwnc) ;
        float wlc(nwnc) ;
        float inc(scal) ;
        int dm(scal) ;
        int ord(scal) ;
        float fbeam(scal) ;
        float wn(nwn) ;
        float tautot(nwn, nlay) ;
        float htaus(nwn, nlay) ;
        float htaux(nwn, nlay) ;
        float taug(nwn, nlay) ;
        float phase(nwn, nlay, nleg) ;
        float ssa(nwn, nlay) ;
        float uu(nwn, nphi, nlev, numu) ;
        float u0u(nwn, nlev, numu) ;
        float tray(nwn, nlay) ;
        float gtauo(tdisr, nwn, nlay) ;
}
```

Conteúdo de um arquivo NetCDF (mostrado por conversão para CDL):

- Há 14 variáveis usadas como dimensões, para as 35 variáveis que são arrays.
- **`alb(nwn)`** indica que **`alb`** é um array de **`nwn`** elementos.
- **`nwn`** está armazenado como uma das dimensões, e é **400.**

(mais sobre NetCDF e CDL na aula sobre arquivos)

# Estruturas - exemplo – agrupamento de muitas variáveis

**readncdfs** (minha rotina) lê todas as dimensões e variáveis de um netcdf.

Com uma estrutura, ela retorna todas as variáveis do arquivo em uma só variável:

```
IDL> a=readncdfs('refspec_g01_0.nc',/long)
```

```
IDL> help,a,/structure
** Structure <1883988>, 4 tags, length=4112896, data length=4112896,
refs=1:
   NCDFNAME        STRING    'refspec_g01_0.nc'
   D               STRUCT    -> <Anonymous> Array[1]
   V               STRUCT    -> <Anonymous> Array[1]
   N               STRUCT    -> <Anonymous> Array[1]
```

# Estruturas - exemplo – agrupamento de muitas variáveis

Esta estrutura contém outras estruturas: As dimensões são campos de **a.d**:

```
IDL> help,a.d,/structure
** Structure <18823f8>, 14 tags, length=28, data length=28, refs=2:
   NLAY            INT              51
   NWN             INT             400
   NLEG            INT              33
   NUMU            INT               2
   NLEV            INT              52
   NPHI            INT               3
   NGAS            INT               2
   NWNC            INT               1
   SCAL            INT               1
   V3              INT               3
   DNL             INT               1
   NK              INT               1
   TDISR           INT               0
   NA              INT              16
```

Os campos de **a.n** indicam as dimensões de cada variável:

```
IDL> print,a.n.uu
numu nlev nphi nwn

IDL> print,a.n.alb
nwn
```

# Estruturas - exemplo – agrupamento de muitas variáveis

**As variáveis são campos de `a.v`:**

```
IDL> help,a.v,/structure
** Structure <1882698>, 35 tags, length=4112004, data
length=4112004, refs=2:
   ALB             FLOAT     Array[400]
   Z               FLOAT     Array[51]
   T               FLOAT     Array[51]
   P               FLOAT     Array[51]
   WL              FLOAT     Array[400]
   IOF             FLOAT     Array[400]
   MTAU            FLOAT     Array[400]
   HTAU            FLOAT     Array[400]
   HCTAUS          FLOAT     Array[400]
   GTAU            FLOAT     Array[400]
   OUTCOS          FLOAT          0.992707
   PHI             FLOAT     Array[33]
   PHIC            FLOAT           29.2098
   UMU             FLOAT     Array[2]
   FLUX            FLOAT     Array[52, 400, 3]
   MIX             FLOAT     Array[2, 51]
   C               FLOAT     Array[51]
   PSAT            FLOAT     Array[51]
   INC             FLOAT          0.957029
(...)
```

# Estruturas - exemplo – correspondência de muitas variáveis

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

```
      SUBROUTINE DISORT( NLYR, DTAUC, SSALB, PMOM, TEMPER, WVNMLO,
     &                   WVNMHI, 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
```

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)

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

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

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

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

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.

- Toda variável em IDL é, internamente, uma estrutura, para informar para toda função/rotina quais são suas dimensões e tipos.

**Essencial para objetos**:

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

•Hierarquia para organizar muitas variáveis e evitar poluição do espaço de nomes (ex. IDL):

```
  void={PPRT, $
   number_type:5, $ ;type to use for real variables internally (4=float,
5=double)
   nlayers:0L, $ ;number of layers
   atmin: { pprt_atmin, $ ;atmospheric parameters
   opd:ptr_new(), $ ;optical depth of each layer
   ssa:ptr_new(), $ ;single scattering albedo at each layer
   phases:ptr_new()}, $ ;structure with phase function or its moments at each
layer
   emcoss:ptr_new(), $ ;cosines of emission angles for output
   bound:{ pprt_bound, $ ;boundary conditions
   fluxisot:0d0, $ ;isotropic flux on the top boundary (W/(m^2) if using
thermal emission)
   fluxbeam:1d0, $ ;beam flux on the top boundary (W/(m^2) if using thermal
emission)
   inccos:0d0, $ ;cosine of incidence angle (of fluxbeam)
   albedo:1d0}, $ ;albedo of bottom boundary
   emiss:{ pprt_emiss, $ ;variables for thermal emission
   use:0B, $ ;include thermal emission?
   temperature:ptr_new(), $ ;temperature (K) of each boundary (level,
nlayers+1)
   tepw:ptr_new(), $ ;thermal emission power (W*m^-2) of each boundary
(level, nlayers+1)
   tepb:0d0, $ ;thermal emission power (W*m^-2) of surface
   toptemp:0d0, $ ;temperature (K) of top boundary
   bottemp:0d0, $ ;temperature (K) of bottom boundary
   topemis:0d0, $ ;emissivity of top boundary
   botemis:0d0, $ ;emissivity of bottom boundary
   wavs:[1d0,1d0]}} ;wavelength (cm^(-1)) interval being used
```

# 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 e C++, que **difere do usado em linguagens dinâmicas ou baseadas em objetos**, como IDL, Python, Java, R.

**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

**A forma como a referência é feita varia entre linguagens.**

Ponteiros de baixo nível (C, C++): são simplesmente **endereços na memória** (em geral, endereços de variáveis):

- Pode-se somar e subtrair inteiros a eles, movendo-se entre elementos na memória.
- Os vetores de C (são de muito baixo nível) são o mesmo que ponteiros para o primeiro elemento do vetor.
- São associados a tipos (ex: um ponteiro que aponta para inteiros não pode apontar para floats).
- Podem apontar também para funções
- Existe o operador inverso ao operador de dereferência (*): & (operador de referência)
- Qualquer variável comum pode ser o alvo de um ponteiro
- É quase impossível programar em C, C++ sem usar ponteiros
- São uma das principais fontes de erros no código e programas que travam ou segfault.

# Ponteiros - conceitos

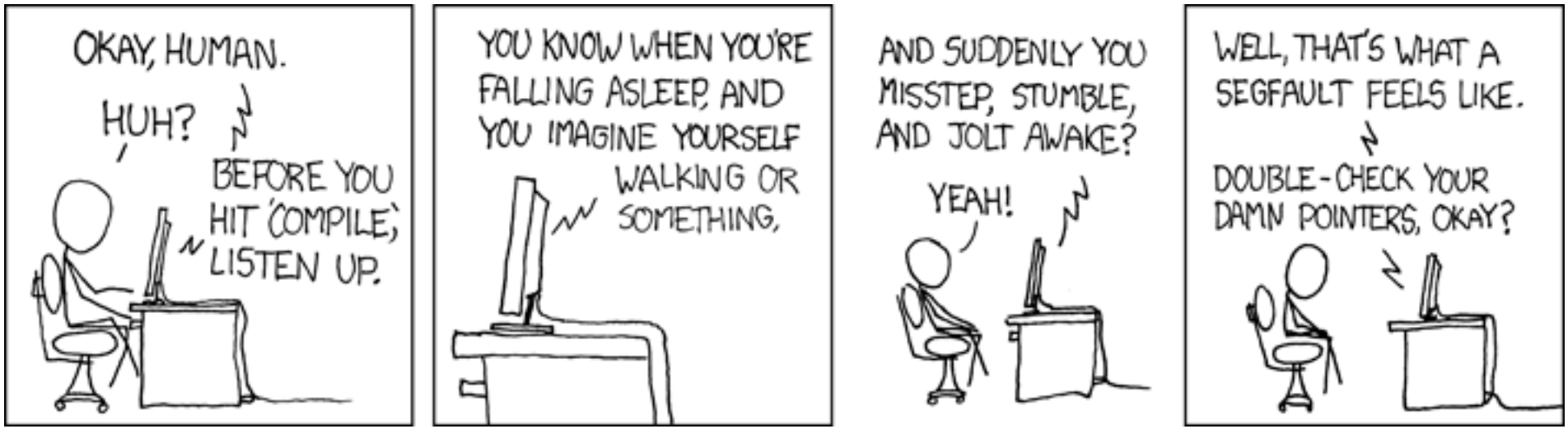
**Linguagens de alto nível têm um novo e mais útil conceito de ponteiros:**

- 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 muito limitados. Exs: R e Fortran.
- Não há os riscos de corrupção, acessos errados, e segfaults de C, C++:

# 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 muito limitados. Exs: R e Fortran.
- Não há os riscos de corrupção, acessos errados, e segfaults de C, C++:




http://www.xkcd.org/371/

# 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 IDL):

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

Sendo **`m=3`**, **`np=[2,4,1]`**, poderia ser **`t=fltarr(7)`**:

```
t=[7.9, 5.8, 9.6, 3.6, 1.4, 7.5, 3.2]
```


2 temperaturas do objeto 0: `t0=t[0:1]`
4 temperaturas do objeto 1: `t1=t[2:5]`
1 temperatura do objeto 2: `t2=t[6:6]`

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           (t0)
    9.6  3.6 1.4 7.5  (t1)
    3.2               (t2)
```

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.

*Listas serão discutidas na próxima aula

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

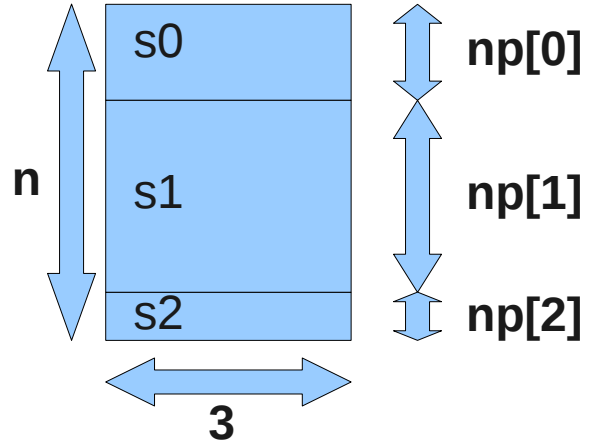
E se o mesmo modelo anterior, de **m** objetos, cada um com um número diferente de pontos, precisa de um array com as coordenadas 3D de cada ponto de cada objeto?

Se todos os objetos tivessem o mesmo número **o** de pontos, seria fácil:

- **`a=fltarr(3,o,m)`**
- as 3 coordenadas do ponto **i** do objeto **j** seriam **`a[0,i,j], a[1,i,j], a[2,i,j]`** (a linha **`a[*,i,j]`**).

Mas e com **np[i]** não constante?

Uma opção seria formar um array **(3,n)**, empilhando as linhas de cada objeto (cada linha é as 3 coordenadas de um ponto). Ex. (IDL):




`a=fltarr(3,n):`

`np[0]` pontos do objeto 0: **`s0=a[*,0:np[0]-1]`**
`np[1]` pontos do objeto 1: **`s1=a[*,np[0]:np[0]+np[1]-1]`**
`np[2]` pontos do objeto 2: **`s2=a[*,np[0]+np[1]:np[0]+np[1]+np[2]-1]`**

É desajeitado de usar. Exige carregar junto o array **np**, e ficar calculando os índices de começo e fim da parte de cada objeto.

# Ponteiros – exemplo – array irregular

Como então implementar o array de largura variável

```
t= 7.9 5.8          (t0)
   9.6 3.6 1.4 7.5  (t1)
   3.2              (t2)   ?
```

Criar os arrays **`t0(2), t1(4), t2(1)`**, e então criar um array para guardar uma referência a cada um?

Por exemplo, pelo nome:

```
t=['t0','t1','t2']
```

Que depois poderia ser usada se houvesse alguma função que retornasse a variável de nome dado:

```
t=variavel_de_nome('t1')
```

Problemas:

- → É desajeitado
- → Em muitas linguagens esta função não existe
- Como inventar os nomes? Deveria haver uma forma automática de os criar (não se sabe antecipadamente quantas serão as variáveis).

# Ponteiros - conceito

**Como um guarda-volumes, ou bibliotecário**: o bibliotecário coloca as coisas (**`t0, t1, t2`**) em prateleiras, e entrega um bilhete, para usar para os acessar depois.

Este tipo de trabalho (contabilidade de onde estão os vetores **`t0, t1, t2`**, 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

Como então fazer e usar o array do exemplo 1

```
t=  7.9  5.8           (t0)
    9.6  3.6  1.4  7.5  (t1)
    3.2                (t2)   ?
```

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

```
t=ptrarr(3)
```

Neste momento, os 3 ponteiros (`t[0], t[1], t[2]`) 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:

```
t[0]=ptr_new([7.9,5.8])
t[1]=ptr_new([9.6,3.6,1.4,7.5])
t[2]=ptr_new([3.2])
```

- Agora os valores estão guardados no guarda-volumes.

# Ponteiros - exemplo - arrays irregulares

O que exatamente cada bilhete diz (o valor dos ponteiros) não importa ao programador:

```
IDL> print,t[0]
<PtrHeapVar106>
```

Só importa ao guardador.

`t[0]` pode dizer, por exemplo, `“caixa 75”`, enquanto `t[1]` pode dizer `“caixa 29”`.

```
IDL> print,t
[<PtrHeapVar106><PtrHeapVar107><PtrHeapVar109>]
```

```
[7.9,5.8]     [9.6,3.6,1.4,7.5]       [3.2]
```

A cada vez que o programa for usado, a caixa onde cada vetor é colocado vai ser em geral diferente.

Usa-se os valores, usando o operador de dereferência (*****):

```
IDL> for i=0,2 do print,*t[i]
      7.90000      5.80000
      9.60000      3.60000      1.40000      7.50000
      3.20000
```

# Ponteiros - exemplo - arrays irregulares

```
IDL> t1=*t[1]

IDL> print,t1
      9.60000      3.60000      1.40000      7.50000

IDL> t1=t1+2.

IDL> print,t1
      11.6000      5.60000      3.40000      9.50000

IDL> *t[1]=t1

IDL> print,*t[1]
      11.6000      5.60000      3.40000      9.50000

IDL> print,mean(*t[0])
      6.85000

IDL> (*t[1])[2]=0.5

IDL> print,*t[1]
      11.6000      5.60000     0.500000      9.50000
```

# Mais propriedades de ponteiros

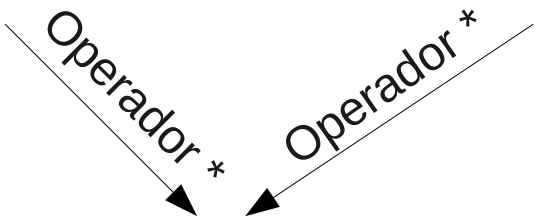
**Um alvo de ponteiro (em IDL, *heap variable*) pode ser apontado por qualquer número de ponteiros (inclusive 0). Ex (IDL):**

```
IDL> a=ptr_new(2.0) ;cria um ponteiro, e o alvo (float 2.0)

IDL> print,*a
      2.00000
IDL> b=ptr_new() ;cria um ponteiro que aponta para nada (nulo)

IDL> print,b
<NullPointer>

IDL> print,ptr_valid(b) ;verifica se o ponteiro aponta para algo
   0

IDL> b=a ;faz b apontar para o mesmo alvo que a

IDL> print,*b ;*b=*a:
      2.00000
```

# Mais propriedades de ponteiros

```
IDL> *a=3.0 ;muda o valor do alvo de a, que é tambem o alvo de b

IDL> print,*b
      3.00000

IDL> *b=5.0 ;muda o valor do alvo de b, que é também o alvo de a

IDL> print,*a
      5.00000

IDL> print,a,b
<PtrHeapVar2><PtrHeapVar2>
```

**a=<PtrHeapVar2>  b=<PtrHeapVar2>**

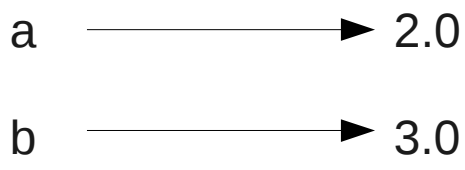



“HeapVar2”=*a=*b

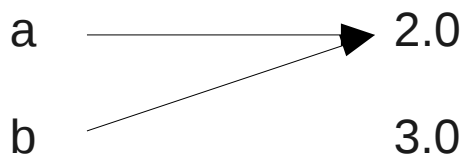
# Mais propriedades de ponteiros

Um ponteiro que já aponta para algo pode ser reapontado, igualando-o a outro ponteiro (inclusive reapontado para o nada, igulando-o a um ponteiro nulo). Ex (IDL):

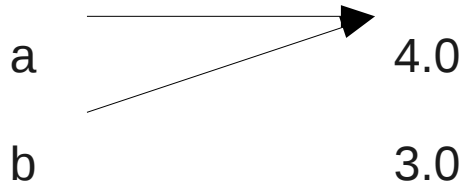
```
IDL> a=ptr_new(2.0) ;cria um ponteiro, e o alvo (float 2.0)

IDL> b=ptr_new(3.0) ;cria um ponteiro, e o alvo (float 3.0)

IDL> print,a,b
<PtrHeapVar1><PtrHeapVar2>
```




```
IDL> print,*a,*b
      2.00000      3.00000

IDL> b=a ;reaponta o b para o alvo de a

IDL> print,a,b
<PtrHeapVar1><PtrHeapVar1>
```




```
IDL> print,*a,*b
      2.00000      2.00000

IDL> *a=*a+2.0

IDL> print,*a,*b
      4.00000      4.00000
```




E agora quem aponta para **“HeapVar2” (3.0)**? Ninguém.

# Alvos órfãos e vazamento de memória

**Alvo órfão: apontado por nenhum ponteiro.**

Destruição de alvos órfãos é parte do que se chama coleta de lixo (GC - *garbage collection*).

Coleta de lixo automática só é implementada em algumas linguagens.

- Fortran, C, C++ não o fazem.
- Java, Python, R e IDL ≥ 8 o fazem.

Ponteiros têm o mesmo escopo que qualquer variável normal.

**Mas os alvos são sempre globais:** Se uma rotina cria um alvo, sem GC, o alvo NÃO é destruído ao sair da rotina.

Sem GC, alvos existem (e ocupam memória):

- Enquanto não forem destruídos chamando uma rotina para isso
- Saindo do programa (Fortran, C, C++)
- Saindo ou reiniciando a sessão (IDL, R, Python)

Alvos órfãos é uma forma de “vazamento de memória/variáveis” (*leaking heap variables*, em IDL):

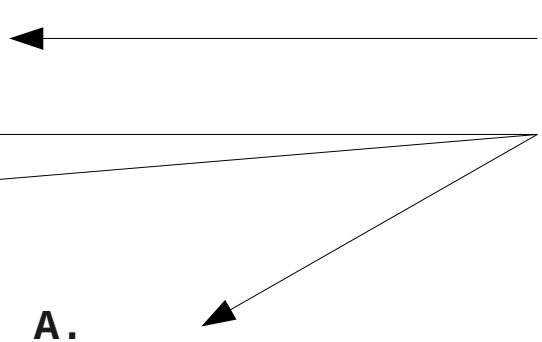
- Se um programa/rotina/função vaza memória, a cada vez que é usado, ele consome mais memória que não é liberada (como se criasse um vazamento).

Nas linguagens com GC, esses pontos são normalmente irrelevantes.

# Ponteiros pendentes (*dangling*)

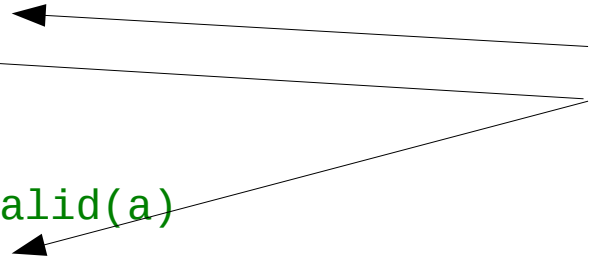
O que acontece com o ponteiro depois que o alvo dele é destruído?

```
IDL> a=ptr_new(5.0)

IDL> print,a,*a
<PtrHeapVar3>      5.00000

IDL> ptr_free,a

IDL> print,a
<PtrHeapVar3>

IDL> print,*a
% Invalid pointer: A.
% Execution halted at: $MAIN$

IDL> print,ptr_valid(a)
   0

IDL> a=ptr_new()

IDL> print,a
<NullPointer>

IDL> print,ptr_valid(a)
   0

IDL> print,*a
% Unable to dereference NULL pointer: A.
% Execution halted at: $MAIN$
```



O alvo de a é destruído

a ainda aponta para o alvo (que não existe mais): a agora é um ponteiro pendente (*dangling*)



a é apontado para o nada: a agora é um ponteiro nulo

# Outras propriedades de ponteiros - linguagens dinâmicas

Ponteiros em linguagens dinâmicas (ex. IDL) podem apontar para variáveis de qualquer tipo (inclusive ponteiros, estruturas e objetos), e qualquer dimensão.

- Mesmo se o ponteiro já apontava para algo, ao ser reapontado ele pode passar a apontar para uma variável de qualquer tipo e tamanho.

- Ponteiros em um mesmo array podem apontar para alvos de diferentes tipos e dimensões.

- Ex (IDL): ao ler a tabela com colunas de tipos diferentes (do exemplo de estruturas), ao invés de usar uma estrutura, poderia ser usado um vetor de ponteiros. Onde se usava a estrutura:

```
IDL> help,a,/structure
** Structure <19e5368>, 11 tags, length=416, data length=416,
refs=1:
   FIELD01         STRING    Array[4]
   FIELD02         LONG      Array[4]
   FIELD03         STRING    Array[4]
   FIELD04         FLOAT     Array[4]
   (...)
```

- Poderia ser usado `a=ptrarr(11)`, onde `a[0]` seria um ponteiro para um array de 4 strings, `a[1]` apontaria para um array de 4 inteiros (*long*), etc.

# Outros exemplos: campos de estruturas

Para definir uma estrutura com arrays, basta incluí-los na definição (ex. IDL):

```
    a={intensities:fltarr(300),coords:
[1.74,3.25,7.18],t:19.5,image:fltarr(100,100)}
```

Mas como fazer se as dimensões destes componentes (`intensities`, `image`) não forem conhecidas na hora?

E se for necessário variar as dimensões?

Resolvido com ponteiros:

```
    a={intensities:ptr_new(),coords:[1.74,3.25,7.18],t:19.5,image:ptr_new()}

    (...)

    a.intensities=ptr_new(fltarr(n))

    (...)

    image=fltarr(n,o)
    a.image=ptr_new(image)
```

Usado em campos de estruturas de objetos, já que sempre têm que ser definidos antes de o objeto ser usado (discutido adiante).

# Outros exemplos: passagem de argumentos

Em algumas situações (exs: C, C++) argumentos são passados por valor, não por referência:

- A rotina/função/programa recebe uma cópia do conteúdo da variável usada como argumento, não a própria variável.

- Mas pode ser de interesse usar referência, para receber valores de volta, ou para evitar copiar 1 milhão de elementos.

Como fazer? Ao invés de usar o argumento em questão, usa-se um ponteiro para ele.

O ponteiro é passado como valor, mas é só um “nome”.

Dentro da rotina/função pode-se acessar então o alvo do ponteiro, já que ele é global.

Importante como substituto de variáveis globais.

Em linguagens dinâmicas como IDL, R, Python, argumentos que são variáveis não qualificadas já são passados por referência mesmo.

# 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 contém 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

A criação de um objeto (instância) é feita criando uma variável daquele tipo (classe):

Em linguagens dinâmicas, por atribuição (ex. IDL):

```
IDL> l=list()
IDL> h=hash()
IDL> w=window()
IDL> help,h
H               HASH  <ID=16972  NELEMENTS=0>
IDL> help,w
W               GRAPHICSWIN <16974>
IDL> help,l
L               LIST  <ID=16971  NELEMENTS=0>
IDL> help,l,/object
** Object class LIST, 2 direct superclasses, 3 known methods
   Superclasses:
      IDL_CONTAINER  <Direct>
      IDL_OBJECT  <Direct>
   Known Function Methods:
(...)
```

Em linguagens estáticas, por declaração (com ou sem atribuição embutida):

```
vector<string> v;   //C++
```

Um **vector** (de strings, vazio)

```
ArrayList al=new ArrayList(); //Java
```

Um **ArrayList** (vazio)

# 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 |
| `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 - uso

Em algumas linguagens (IDL, R, Python), objetos são implementados como referências (como ponteiros):

Se não há coletor de lixo automático (ex: IDL≤7), é necessário os destruir, assim como ponteiros.

Fazer uma cópia de um objeto gera uma outra referência ao mesmo alvo*. Ex. (IDL):

```
IDL> l=list(0,1,2)

IDL> l2=l

IDL> l.add,3

IDL> print,l2
       0
       1
       2
       3
```

*Uma *shallow copy*. Para quando não é o desejado, costuma existir *deep copy*.

# Objetos - quais são as vantagens do seu uso?

## 1 - Rotinas são subordinadas aos tipos

Ao invocar um método a busca por rotinas daquele nome acontece apenas entre os métodos da classe (e de suas superclasses, se não encontrado nela). Não há confusão entre rotinas de mesmo nome feitas para classes diferentes. Ex (IDL):

```
IDL> l=list(5,2)
IDL> h=hash([5,2])
IDL> l.remove,1
IDL> print,l
       5
IDL> h.remove,2
IDL> print,h
5: !NULL
```

Método `remove` da classe `list` (a classe do objeto `l`): `list::remove`

Método `remove` da classe `hash` (a classe do objeto `h`): `hash::remove`

Cada rotina `remove` foi **feita para apenas uma classe**: `list::remove` só sabe o que é uma `list`, e só sabe fazer o seu trabalho (remover elementos) dentro classe `list`.

Não é necessário ter uma única função `remove`, global, **que saiba lidar com qualquer coisa que precise de um remove** (listas, hashes, conjuntos, etc).

**Sem classes, houveria apenas uma função global:**

- Ela teria que ser alterada / aumentada sempre que um tipo fosse alterado / criado.
- Cada usuário que criasse um novo tipo (ou mudasse um existente) teria a sua própria versão da rotina `remove`.
- Como compartilhar código com outro usuário que teria sua outra remove, diferente?

# Objetos - quais são as vantagens do seu uso?

2 - 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.** Ex (IDL):

```
;Create the object, reading the cube file
a=pp_readcube('CM_1553510065_1_ir.cub')
;Get the core and its wavelengths
a.getproperty,core=core,wavelengths=wavs

(muitas linhas de código)

;Find out the names of the backplanes
print,a.getproperty(/backnames)
;Make a contour plot of the latitudes
c=contour(a.getsuffixbyname('LATITUDE'))

(muitas linhas de código)

;Get the band with wavelength nearest to 2.1 (in the units used in the cube)
selband=a.getbandbywavelength(2.1,wavelengths=selwavs)

(muitas linhas de código)

;Get the start time of the cube
print,a.getfromheader('START_TIME')
;"2007-084T10:00:57.286Z"
```

No lugar de ter apenas, no começo, algo como:

```
pp_readcube,'CM_1553510065_1_ir.cub',core=core,wavelengths=wavs,backnames=bna
mes,latitude=latitude,start_time=start_time,lines=lines,samples=samples,...
```

# 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 (l) then print,'list is true' else print,'list is false'
list is false

IDL> l.add,9

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

# Objetos - quais são as vantagens do seu uso?

3 - **Mais expressividade e conveniência** por *overloading* de operadores / rotinas:

```
IDL> print,l1[2]
      16

IDL> p=plot(/test)  ;cria um objeto da classe plot

IDL> help,p['xaxis'],/objects
** Object class AXIS, 1 direct superclass, 9 known methods
(...)
```

Todos os elementos marcados em vermelho são casos de *overloading:*

- A classe define o que acontece quando se usa algum elemento overloaded
- **[ ]** também são operadores, que também foram *overloaded* nesta classe.

# Objetos - quais são as vantagens do seu uso?

4 - **Toda a complexidade necessária para que o objeto funcione fica guardada** (encapsulada) dentro do código da classe.

O usuário só tem que chamar os métodos da classe para ter toda a funcionalidade, sem se preocupar em como ela é feita.

As variáveis são ativas (ou inteligentes).

Ex (IDL): No lugar de chamar uma rotina com 8 argumentos, tendo-se que os decidir todos na hora:

```
IDL> p=plot(cos(dindgen(401)/1d2*!dpi))
IDL> p.title='Some plot'
IDL> p.color='red'
IDL> print,p.xrange
       0.0000000       400.00000
IDL> p.xrange=[25.,350.]
IDL> p.symbol='circle'
IDL> p.yrange=[-2.,2.]
IDL> p.ytitle='$\lambda^-5$'
IDL> p.ytitle='$\lambda^{-5}$'
IDL> p.sym_size=0.5
IDL> p.symbol='square'
IDL> p.save,'some_file.pdf'
```

Interativamente, viu-se que não produziu o desejado
Então o título foi trocado

Mudança de idéa sobre o símbolo

# Objetos - quais são as vantagens do seu uso?

4 - **Toda a complexidade necessária para que o objeto funcione fica guardada** (encapsulada) dentro do código da classe.

Note que coisas como `p.color='red'` e `print, p.xrange` não são acessos diretos a campos dentro do objeto: são usos do operador (.) overloaded, que estão chamando métodos:

`p.color='red'` → `p.setproperty,color='red'`

print, `p.xrange` →
```
p.getproperty,xrange=_some_temporary_variable_
print,_some_temporary_variable_
```

# Objetos - quais são as vantagens do seu uso?

Note também a simplicidade de:

```
;Get the band with wavelength nearest to 2.1 (in the units used in the cube)

selband=a.getbandbywavelength(2.1,wavelengths=selwavs)
print,selwavs
;2.1003400
```

No lugar de ter que saber como os dados estão armazenados, para pensar em como descobrir qual é a banda com comprimento de onda mais próximo de 2.1 µm e a selecionar.

Com *overloading*, pode ser melhor ainda:

```
selband=a[2.1]
```

Invoca o método que retorna a banda de comprimento de onda mais próximo

```
print,a.wavelength(2.1)+' '+a.units
;2.1003400 micrometer
```

# Objetos - quais são as vantagens do seu uso?

**5 - Dados são mantidos consistentes e válidos:**

Em estruturas, o usuário pode alterar o valor de qualquer campo, para qualquer coisa.

Mas em muitas situações não é qualquer valor que seria válido para um campo, e é necessário manter a consistência com outros.

Ex (IDL): uma estrutura contendo um cubo de dados:

```
IDL> cube={nlines:100, ncolumns:200, nbands:300, data:ptr_new()}
IDL> cube.data=ptr_new(dblarr(cube.ncolumns,cube.nlines,cube.nbands))
```

Até aqui, tudo bem. Mas:

Não faz sentido

```
IDL> cube.nlines=-5

IDL> cube.data=ptr_new(dblarr(12,78,47))

IDL> help,cube
** Structure <b9591678>, 4 tags, length=12, data length=10, refs=1:
   NLINES          INT              -5
   NCOLUMNS        INT             200
   NBANDS          INT             300
   DATA            POINTER   <PtrHeapVar2>

IDL> help,*cube.data
<PtrHeapVar2>   DOUBLE    = Array[12, 78, 47]
```

Não são coerentes

# Objetos - quais são as vantagens do seu uso?

Objetos podem forçar que **só sejam usados dados válidos, e manter todos os dados internos consistentes** (em geral, manter os metadados consistentes com as características dos dados). Ex (IDL):

```
IDL> cube=pp_editablecube(file='CM_1503394149_1_ir_eg.cub')

IDL> print,cube.lines,cube.samples,cube.bands
          64          64         256

IDL> help,cube.core
<Expression>    FLOAT     = Array[64, 64, 256]

IDL> cube.core=dblarr(100,100,256)
```

**A linha acima não é uma atribuição**: é uma invocação de método. É equivalente a

```
cube.setproperty,core=dblarr(100,100,256)
```

É o trabalho da função `setproperty` decidir se a propriedade passada (`core`) faz sentido, e, se fizer, fazer todo o trabalho necessário a esta mudança.

# Objetos - quais são as vantagens do seu uso?

No caso, como é uma mudança nas dimensões dos dados, há muito trabalho a ser feito, em muitos metadados (como dimensões e unidades).

Isso tudo é trabalho dos métodos da classe.

Para o usuário, é tudo automático, e os dados da classe são válidos o tempo todo:

```
IDL> help,cube.core
<Expression>    DOUBLE    = Array[100, 100, 256]

IDL> print,cube.lines,cube.samples,cube.bands
         100         100         256
```

# 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.

# Objetos - herança de classes - exemplo

Alguém acha inconveniente ter que testar se uma lista é vazia com (ex. IDL)

```
if (n_elements(list) eq 0) then ....
```

Seria melhor ter um método que o informasse diretamente:

```
if list.isempty() then ...
```

Uma classe com este método pode ser feita em 6 linhas:

```
function my_new_list::isempty
  return, (n_elements(self) eq 0)
end

pro my_new_list__define
  !null={my_new_list, inherits list}
end
```

Definição do novo método.

Definição do novo tipo: apenas herda o tipo **`list`**; neste caso não é necessário adicionar mais campos.

Se a classe `my_new_list` definisse métodos com o mesmo nome de métodos de sua superclasse (`list`), estes métodos seriam chamados no lugar dos métodos de `list` (method *overriding*): É a forma de se alterar alguma característica já existente.

# Objetos - herança de classes - exemplo

O tipo `my_new_list` herda o tipo `list`. Ele tem todas as características do `list`, também é um `list`.

Testa se o objeto `l` é da classe `my_new_list`

Testa se o objeto `l` é da classe `list`

```
IDL> l=my_new_list()
IDL> print,isa(l,'my_new_list')
   1
IDL> print,isa(l,'list')
   1
IDL> help,l,/object
** Object class MY_NEW_LIST, 1 direct superclass, 3 known methods
   Superclasses:
      LIST  <Direct>
      IDL_CONTAINER
      IDL_OBJECT
(...)
```

E tem um método adicional (`isempty()`) que list não tem:

```
IDL> print,l.isempty()
   1
IDL> l.add,42
IDL> print,l.isempty()
   0
```

Este exemplo é artificial e simples demais. Exemplos mais detalhados e concretos serão mostrados nos contextos das próximas aulas.

# Sumário

3 – Slides em http://www.ppenteado.net/pea/pea02_variaveis.pdf

- Tipos de variáveis
- Representações de números e suas conseqüências
- Ponteiros
- Estruturas
- Objetos

Mais exemplos destes conceitos serão mostrados nos assuntos das próximas aulas.

Exercícios sugeridos (soluções estarão online mais tarde):

Responder às perguntas:

*1)*
*This may be a stupid question, but I really want to know why.*
*Please, see below and explain. Thanks.*
*IDL> print, 132*30*
*3960*
*IDL> print, 132*30*10*
*-25936*

# Perguntas a responder

*2)*
*There's something I can not explain to myself, so maybe someone can enlighten me?*
*IDL> print, fix(4.70*100)*
*469*

*To try and find where the problem is, we tried the following lines:*
*IDL> a = DOUBLE(42766.080001)*
*IDL> print,a,FORMAT='(F24.17)'*
*42766.07812500000000000*
*As you see, the number we get out isn't the same as the number we entered.*

*3)*
*I have a problem related to float-point accuracy*
*If I type in: 50d - 1d-9, I get 50.000000*
*And here lies my problem, I'm doing a numerical simulation where such an arithmetic is common place, and as a result i get a lot or errors. I know for example, that if i simply type print, 50d - 1d-9, format = '(f.20.10)' , i'll get:*
*49.9999999990*
*But how can I convince IDL to do it on its own during computations?*

# Perguntas a responder

*4)*
*Hi guys,*
*IDL> print,((10^5)/(exp(10)*factorial(5)))*
*The actual result of the above line is 0.0378332748*
*But when we run it in IDL we get the result as -0.011755556*

*5)*
*I ran into a number transformation error yesterday that is still confusing me this morning. The problem is that the number 443496.984 is being turned into the number 443496.969 from basic assignments using Float() or Double(), despite the fact that even floats should easily be able to handle a number this large (floats can handle "±10^38, with approximately six or seven decimal places of significance").*

# Perguntas a responder

Exercício mais avançado (uma solução estará online mais tarde):

Escrever um programa para calcular a integral de uma gaussiana:

$$\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}$$

Da forma mais simples: por áreas de retângulos de largura constante.

Questões a considerar:

- Até onde continuar somando? (não dá para somar até o infiinito)
- Importa a ordem da soma?
- Qual a diferença entre usar precisão simples e dupla?

A ser respondido na próxima aula:

- É possível calcular esta integral sem loops? Em menos de uma dúzia de linhas de código?

# Próxima aula

4 – Slides em http://www.ppenteado.net/pea/pea03_conteiners.pdf

- Contêiners
- Arrays
- Listas
- Mapas
- Outros contêiners
- Vetorização
- Escolha de contêiner

http://www.ppenteado.net/pea