Centro Tecnológico
Departamento de Informática

Prof. Veruska Zamborlini
veruska.zamborlini@inf.ufes.br
http://www.inf.ufes.br/~veruska.zamborlini

Aula 9
Subprogramas
2021/2

Material adaptado
Prof.s Vitor Souza e Eduardo Zambon

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Abstrações em programação

- Abstração de processos (Cap. 9 - Subprogramas)
  - Conceito central de LPs desde a década de 1960
  - Sistemas atuais não existiriam sem essa abstração
  - Imagine cada programa como milhões de linhas de código sem separação

- Abstração de dados (Cap. 11 - TADs e Encapsulamento)
  - Ganhou importância a partir da década de 1980
  - Muito fortalecida com Programação Orientada a Objetos
  - Foco em encapsulamento e ocultamento de informações

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Subprogramas (exceto co-rotinas)

- Possuem um único ponto de entrada
- Suspendem a execução de quem chama
  - Execução sequencial
- Retornam ao ponto de chamada ao terminar
  - Evita saltos arbitrários no código
  - Logo, evita código macarrônico

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Subprogramas

- São formados por:
  - Uma definição (cabeçalho – header)
    - Estabelece a interface com o chamador
  - Um corpo
    - Define as ações realizadas

- Cabeçalho (protótipos) define a assinatura (protocolo) da função:
  - sum : int x int -> int
  - perfil de parâmetros: int x int
  - permite verificação de tipos
    - int x = sum(4.2, "abc") está errado

Utilidade do protótipo

Código C

```
int f(int a);

void g(void) {
        int b =
f(42);
        …
}

int f(int a) {
        return 2 *
a;
}
```

Passada simples

Código Java

```
class Example {

    void g(void) {
                int b =
f(42);
        …
    }

    int f(int a) {
                return 2
* a;
    }
}
```

Múltiplas passadas

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Procedimentos e funções

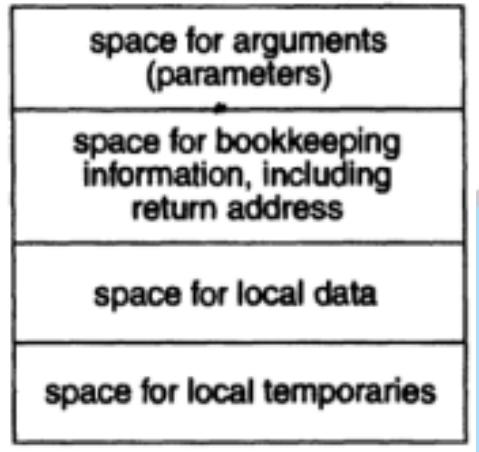
- Diferem basicamente se retornam um valor ou não
  - **Função:** abstração de uma expressão
  - **Procedimento:** abstração de um comando
- Poucas LPs exigem distinção com keywords diferentes
- Na prática: tudo são funções
  - Funções são procedimentos retornam void ou um tipo vazio equivalente

- Daqui para frente: tudo função

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

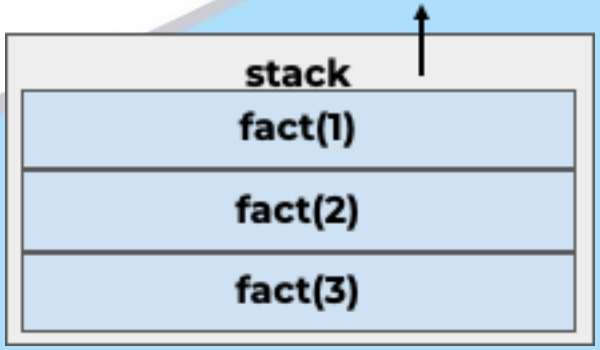
Frames de funções

- Uma chamada a uma função a torna ativa:
  - Funções ativas são trechos de código em execução
- Precisam de algum lugar na memória para trabalhar
  - ⇒ Registro de ativação de função (frame)
- Armazena todas as informações de uma função em execução
  - Argumentos
  - Variáveis locais
  - Endereço de retorno
  - Etc



Funções, chamadas e *frames*

- Qual é a relação entre uma função e um *frame*?
- Pode ser 1:1, isto é, uma função declarada gera só um *frame*
  - Chamado de ambiente totalmente estático
  - *Frame* é alocado na área estática de memória
  - Não permite recursão!
- Pode ser 1:N, isto é, cada chamada gera um *frame*
  - Feito assim para permitir recursão
  - Neste caso, *frames* precisam ser alocados dinamicamente
  - Geralmente isto ocorre na pilha (*stack*)
  - Daí vem o nome usual: *stack frame*



Ativando subprogramas

- Requer a conexão entre os parâmetros:
  - Parâmetros formais (cabeçalho)
  - Parâmetros reais ou argumentos

- Questão essencial: como essa conexão é feita?
  - Associação básica é posicional (pela ordem)
  - Também pode ser por palavra-chave
  - Valores default (devem ser os últimos da lista)

```
int soma (int a[], int inicio = 0, int fim = 7,
                                        int incr = 1) {
  int soma = 0;
  for (int i = inicio; i < fim; i += incr) soma += a[i];
  return soma;
}

int main() {
  int[] pontuacao = {9, 4, 8, 9, 5, 6, 2};
  int ptotal, pQuaSab, pTerQui, pSegQuaSex;
  ptotal = soma(pontuacao);
  pQuaSab = soma(pontuacao, 3);
  pTerQui = soma(pontuacao, 2, 5);
  pSegQuaSex = soma(pontuacao, 1, 6, 2);
}
```

```
sumer(length = my_length,
      list = my_array,
      sum = my_sum)
```

```
sumer(my_length,
      sum = my_sum,
      list = my_array)
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Parâmetros: reais, formais e argumentos

- Parâmetro formal:
  - Identificadores listados no cabeçalho do subprograma e usados no seu corpo;
- Parâmetro real:
  - Valores, identificadores ou expressões utilizados na chamada do subprograma;
- Argumento:
  - Valor passado do parâmetro real para o parâmetro formal.

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Parâmetros: reais, formais e argumentos

Parâmetro formal.

```
float area (float r) {
  return 3.1416 * r * r;
}

main() {
  float diametro, resultado;
  diametro = 2.8;
  resultado = area(diametro / 2);
}
```

Parâmetro real = diametro / 2
Argumento = 1.4

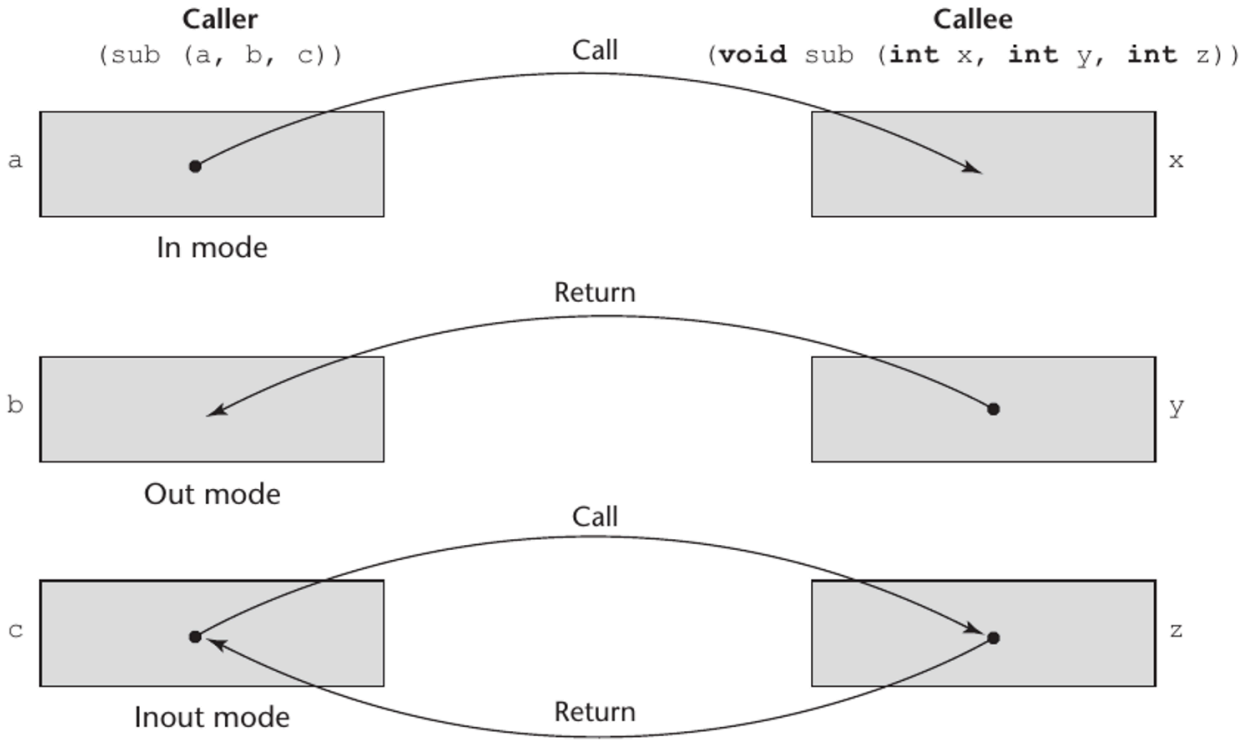
UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Passagem de parâmetros

- Conexão entre o formal e o real
- Essencial distinguir entre a **semântica da passagem e formas de implementação**
- Ada foi a primeira LP a fazer essa distinção de forma clara

Semântica: in, out e inout

- Entrada, saída e entrada-e-saída

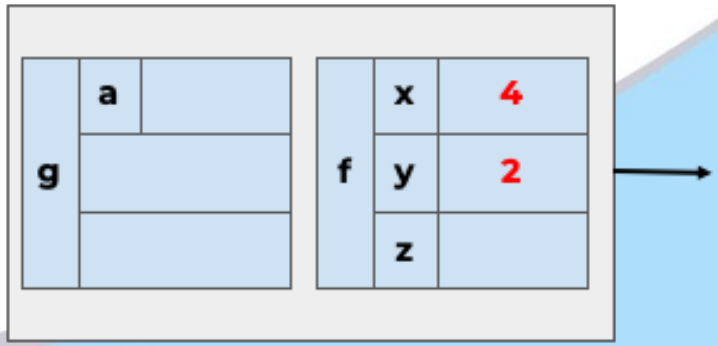


Passagem por valor

- Implementa semântica de *in mode*
- Valor do argumento copiado para o parâmetro formal
- Eficiente para tipos simples
- Não é adequado para tipos compostos (ineficiente)

```
void f(int x, int y) {
        int z;
        …
}

void g(void) {
        int a;
        f(4, 2);
}
```

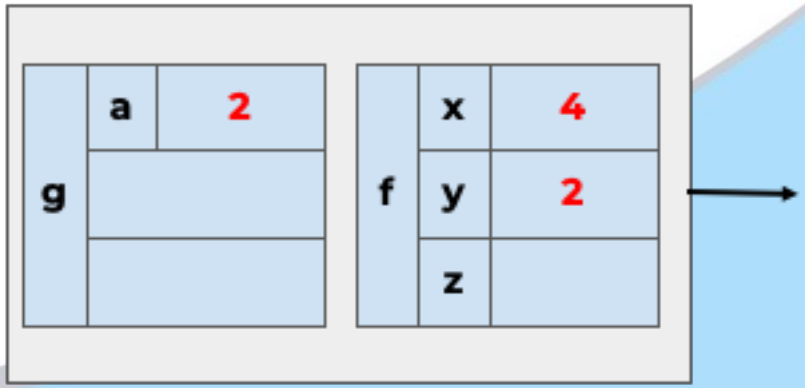


Passagem por resultado

- Implementa semântica de *out mode*
- Valor do argumento copiado do parâmetro formal
- Parâmetro real precisa ser uma variável, e não uma expressão
- Como lidar com colisões (aliases)? *E.g. f(a,a) se ambos são out*

```
void f(int x, out int y) {
        int z;
        y = 2;
}

void g(void) {
        int a;
        f(4, a);
}
```



UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Passagem por valor-resultado (cópia)

- Implementa semântica de *inout mode*
- Valor do argumento copiado para o parâmetro formal no início
- Valor do argumento copiado do parâmetro formal no final
- Combina os problemas de ambos os casos anteriores

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Passagem por nome

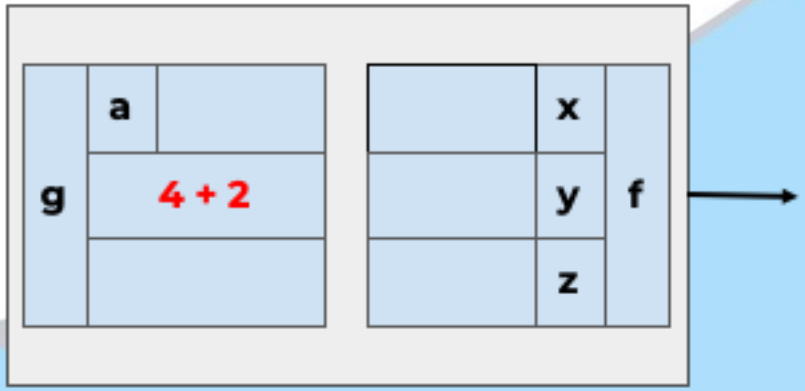
- Não se usa hoje em dia (muito confuso)
- Avaliação dos argumentos ocorre no uso e não na chamada
- Lembra macros de C

Passagem por referência

- Também implementa semântica de *inout mode*
- Eficiente para qualquer tipo de dado em termo de espaço
- Leve ineficiência no acesso por causa de indireção
- Problema: como fazer com expressões como argumentos?

```
void f(int& x, int& y) {
        int z;
        …
}

void g(void) {
        int a;
        f(a, 4+2);
}
```



UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Aspectos de Passagem (outra visão)

- Direção da passagem;
- Mecanismo de implementação;
- Momento no qual a passagem é realizada.

Direção de Passagem

Direção da passagem	Forma do parâmetro real (R)	Atribuição do parâmetro formal (F)	Fluxo	
Entrada variável	Variável, constante ou expressão	Sim	R → F	C, Java
Entrada constante	Variável constante ou expressão	Não	R → F	C++, Ada
Saída	Variável	Sim	R ← F	Ada
Entrada e saída	Variável	Sim	R ↔ F	C++, Java, Ada

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Direção de Passagem

- C usa passagem unidirecional de entrada variável:

```
void naoTroca (int x, int y) {
  int aux;
  aux = x;  x = y; y = aux;
}

void troca (int* x, int* y) {
  int aux;
  aux = *x; *x = *y;  *y = aux;
}

int main() {
  int a = 10, b = 20;
  naoTroca(a, b);
  troca(&a, &b);
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Direção de Passagem

- C++ tem unidirecional de entrada constante e bidirecional:

```
int triplica (const int x) {
  // x = 23;
  return 3*x;
}

void troca (int& x, int& y) {
  int aux;
  aux = x;  x = y; y = aux;
}

int main() {
  int a = 10, b = 20;
  b = triplica(a);
  troca(a, b);
  // troca(a, a + b);
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Direção de Passagem

- Java usa passagem unidirecional de entrada;
- Se os parâmetros são objetos, porém, pode-se considerar passagem bidirecional do objeto ou passagem unidirecional da referência.

```
void preencheVet(final int[] a, int i, final int j) {
  while (i <= j) a[i] = i++;
  // j = 15;
  // a = new int [j];
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Direção de Passagem

- Ada usa unidirecional de entrada constante, unidirecional de saída e bidirecional

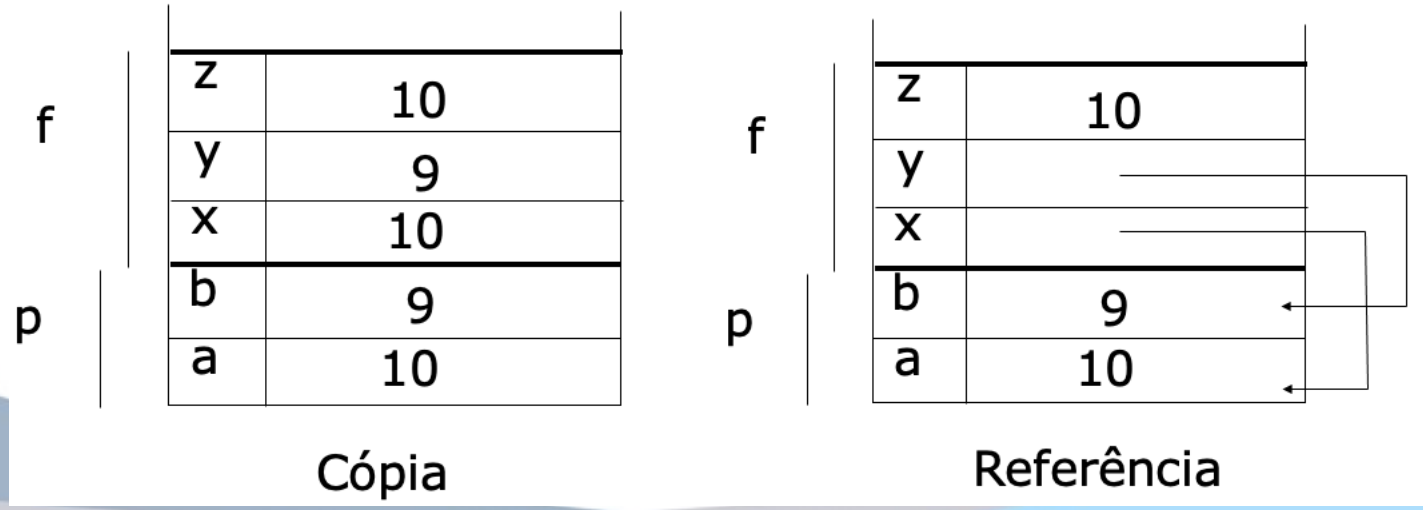
```
function triplica (x: in integer; out erro: integer)
return integer;

procedure incrementa (x: in out integer; out erro:
integer);
```

Mecanismos de passagem

```
void f(int x, int y) {
  int z = 10;
}

void p() {
  int a = 10, b = 9;
  f(a, b);
}
```



UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Mecanismos de passagem

- Cópia:
  - Viabiliza a passagem unidirecional de entrada variável;
  - Facilita a recuperação do estado do programa em interrupções inesperadas;
- Referência:
  - Proporciona semântica simples e uniforme na passagem de todos os tipos (ex.: passar função como parâmetro);
  - Mais eficiente por não envolver cópia de dados;
  - Pode ser ineficiente em implementação distribuída;
  - Permite a ocorrência de sinonímia:

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Mecanismos de passagem

- **Passagem unidirecional de entrada por cópia** é conhecida como **passagem por valor**;
- Passagem unidirecional de entrada constante por referência equivale a por valor;
  - Tem como vantagem de não demandar cópias de grandes volumes de dados.

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Mecanismos de passagem

- C oferece apenas passagem por cópia;
- C++ e Pascal oferecem mecanismos de cópia e referência;
- Ada usa cópia para primitivos e referência para alguns tipos. Outros tipos são definidos pelo compilador e podem ser cópia ou referência;
- Java adota passagem por cópia. Quando os parâmetros são objetos, pode-se considerar:
  - Passagem por referência dos objetos; OU
  - Passagem por cópia das referências (endereço de memória).

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Momento da passagem

- Normal (eager): avaliação na chamada do subprograma;
- Por nome (by name): avaliação quando parâmetro formal é usado;
- Preguiçosa (lazy): avaliação quando parâmetro formal é usado pela primeira vez.

- Uso:
  - Maioria das LPs (C, Pascal, Java, Ada): modo normal;
  - ALGOL-60 permite escolher: normal e por nome;
  - SML entre normal e preguiçoso;
  - Haskell e Miranda usam preguiçoso;
  - Python: pacotes lazypy e objproxies.

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Momento da passagem

```
int caso (int x, int w, int y, int z) {
  if (x < 0) return w;
  if (x > 0) return y;
  return z;
}

caso(p(), q(), r(), s());
```

- Avaliação normal:
  - Avaliação desnecessária de funções (dependendo do valor de x);
  - Pode reduzir eficiência e flexibilidade.

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Momento da passagem

```
int caso (int x, int w, int y, int z) {
  if (x < 0) return w;
  if (x > 0) return y;
  return z;
}

caso(p(), q(), r(), s());
```

- Avaliação normal:
  - Avaliação desnecessária de funções (dependendo do valor de x);
  - Pode reduzir eficiência e flexibilidade.

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Momento da passagem

```
int caso (int x, int w, int y, int z) {
  if (x < 0) return w;
  if (x > 0) return y;
  return z;
}

caso(p(), q(), r(), s());
```

- Avaliação por nome:
  - Somente uma de q(), r() ou s() seria avaliada;
  - Porém, p() poderia ser avaliada duas vezes;
  - Problema mais grave se p() produzir efeitos colaterais!

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Momento da passagem

```
int caso (int x, int w, int y, int z) {
  if (x < 0) return w;
  if (x > 0) return y;
  return z;
}

caso(p(), q(), r(), s());
```

- Avaliação preguiçosa:
  - Única execução de p() e somente uma de q(), r() ou s();
  - Pode não ser intuitivo que as funções não sejam avaliadas.

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Funções permitem isolamento

- Funções devem minimizar modificações de memória externa
- Minimiza estado ⇒ facilita entendimento do programa
- Variedades de implementação mostram que não há consenso
- Importante é entender e dominar os conceitos de cada LP

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Funções como elementos de primeira classe (first-class)

- Um cidadão (objeto, entidade, elemento, etc) é considerado de primeira classe em uma LP se:
  - 1.Ele pode ser passado como um argumento para funções
  - 2.Ele pode ser retornado como um resultado de uma função
  - 3.Ele pode ser atribuído a variáveis e outras estruturas
  - 4.Ele pode ser testado para igualdade com outro elemento
- ***Funções*** de primeira-classe:
  - Sempre satisfazem items 1, 2 e 3
  - Item 4 é substituído por outro específico para funções:
    - Suporte a funções anônimas e aninhadas
    - Sebesta não considera este item, Scott sim

1. Funções de alta ordem (passando funções como argumentos)

Haskell

```haskell
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

C

```c
void map(int (*f)(int), int x[], size_t n) {
    for (int i = 0; i < n; i++)
        x[i] = f(x[i]);
}
```

4. Funções anônimas (lambda)

Haskell

```haskell
main = map (\x -> 3 * x + 1) [1, 2, 3, 4, 5]
```

C

```c
int f(int x) {
    return 3 * x + 1;
}

int main() {
    int list[] = {1, 2, 3, 4, 5};
    map(f, list, 5);
}
```

- não suporta funções anônimas então é preciso uma amarração a um nome

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Variáveis não-locais e *closures*

- O termo *closure*
  - Tradução correta: fechamento ou fecho
  - Tradução incorreta: clausura (corrompido de cloister)
  - Também chamado de fecho léxico ou fecho funcional
- Técnica para fazer
  - Amarração de nomes
  - Com escopo léxico aninhado
- Um fecho é
  - Função que define a computação
  - Ambiente que associa valores às variáveis livres da função

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Exemplo de closure

```python
def f(x):
    def g(y):
        return x + y
    return g  # Return a closure.

def h(x):
    return lambda y: x + y  # Return a closure.

# Assigning specific closures to variables.
a = f(1)
b = h(1)

# Using the closures stored in variables.
assert a(5) == 6
assert b(5) == 6

# Using closures without binding them to variables first.
assert f(1)(5) == 6  # f(1) is the closure.
assert h(1)(5) == 6  # h(1) is the closure.
```

Funções aninhadas e amarrações/vinculações

- **Amarração rasa (shallow / dinâmica)**
  - Ambiente que cerca a chamada da função
  - Chamada de sub2 em sub4 ⇒ imprime 4
  - Usado em LISP e outras LPs, como JS
- **Amarração ad hoc**
  - Ambiente onde a função foi passada como argumento
  - Chamada de sub3 ⇒ imprime 3
  - Não se usa, muito confuso
- **Amarração profunda (deep / estática)**
  - Ambiente que cerca a definição da função
  - Chamada de sub2 ⇒ imprime 1
  - Padrão para LPs estáticas

```
function sub1() {
  var x;
  function sub2() {
    alert(x);  // Create
    };
  function  sub3() {
    var  x;
    x = 3;
    sub4(sub2);
    };
  function  sub4(subx) {
    var  x;
    x = 4;
    subx();
    };
  x = 1;
  sub3();
  };
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Sobrecarga de funções

- **Sobrecarga:**
  - Mesmo nome
  - Assinaturas diferentes
- Algumas LPs olham só os parâmetros
- Outras consideram o valor de retorno também
- Segundo caso só para LPs sem coerção implícita
  - Senão fica impossível saber qual função usar

```
int Volume(int s) {  // Volume of a cube.
  return s * s * s;
}

double Volume(double r, int h) {  // Volume of a cylinder.
  return 3.1415926 * r * r * static_cast<double>(h);
}

long Volume(long l, int b, int h) {  // Volume of a cuboid.
  return l * b * h;
}

int main() {
  std::cout << Volume(10);
  std::cout << Volume(2.5, 8);
  std::cout << Volume(100l, 75, 15);
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Corotinas

- Melvin Conway (1958)
- Subprogramas são casos especiais de corotinas
- Subprogramas
  - Só possuem um ponto de entrada
  - Retornam e terminam
  - Não guardam estado entre chamadas
  - Distinção entre caller-callee
- Corotinas
  - Múltiplos pontos de entrada
  - Podem ceder a execução (yield) para outra corotina
  - Guardam estado entre chamadas
  - Relação simétrica entre as corotinas

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Corotinas

- Exemplo: produtor/consumidor
- Costuma ser um exemplo de multithreading
- Mas pode ser implementado sem threads
- Corotinas são multitarefas cooperativas
- Threads são multitarefas preemptivas
- Assunto para SO

```
var q := new queue

coroutine produce
    loop
        while q is not full
            create some new items
            add the items to q
        yield to consume

coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
        yield to produce

call produce
```