Linguagens de Programação

Subprogramas

Baseado em Conceitos de Linguagens de Programação – Robert W. Sebesta

Prof. Lucas Ismaily

Universidade Federal do Ceará
Campus Quixadá

Roteiro

1. Introdução
2. Fundamentos de Subprogramas
3. Ambientes de Referência local
4. Métodos de Passagem de Parâmetros
5. Subprogramas Sobrecarregados
6. Subprogramas Genéricos
7. Compilação Separada e Independente
8. Questões de projecto referentes a funções
9. Acesso a Ambientes não locais
10. Sobrecarga de operadores pelo utilizador
11. Co-rotinas

Introdução

- Tipos de Abstracção em L.P.:
  - Abstracção de Processos – Subprogramas abstractos que podem ser reutilizados;
  - Abstracção de Dados – Estruturas de dados abstractas que podem ser reutilizadas;

Fundamentos de Subprogramas

- Subprogramas – são unidade de programação para construção de programas.

  Características Fundamentais de Subprogramas

  - Possuem um único ponto de entrada;
  - O invocador (ou chamador) tem a sua execução suspensa durante a execução de um subprograma por si invocado (ou chamado);
  - Quando um subprograma invocado termina a sua execução, o controle retorna sempre para o invocador.

Definições Básicas

- Uma definição de subprograma descreve a interface e as acções que o subprograma implementa.
- Uma chamada (invocação) de subprograma é uma requisição explicita para que o subprograma seja executado.
- O cabeçalho do subprograma é a primeira linha de sua definição, incluindo o nome, o tipo do subprograma e os parâmetros formais.

Definições Básicas

- O perfil dos parâmetros dum subprograma é a lista de parâmetros formais, incluindo o número, ordem e seus tipos.
- O protocolo de um subprograma é o seu perfil de parâmetros, e no caso de função, conjuntamente com o tipo de retorno.
- Uma declaração de subprograma (protótipo) fornece o protocolo mas não o corpo do subprograma.

Definições Básicas

- Um parâmetro formal é uma variável fictícia, definida no cabeçalho de um subprograma. O seu escopo é geralmente igual ao do subprograma.
- Um parâmetro real (ou actual) representa o valor (ou endereço) das variáveis ou constantes, utilizadas no ponto de invocação do subprograma.

Definições Básicas

Correspondência entre parâmetros Formais e Actuais

- Parâmetros Posicionais:
  - A vinculação é efectuada pela ordem dos parâmetros formais, isto é, o 1º parâmetro actual é vinculado ao 1º parâmetro formal, e assim por diante.
  - Por palavra-chave (ou nomeado):
  - Ex. em Visual Basic:

```
ShowMesg(Mesg:="Hello World", MyArg1:=7)
```

  - Vantagem: ordem dos parâmetros é irrelevante;
  - Desvantagem: programador tem de conhecer os nomes dos identificadores dos parâmetros formais.

Fundamentos de Subprogramas

- Valores por omissão – são valores definidos nos parâmetros formais, de forma a serem utilizados na falta destes valores nos parâmetros reais.
  - Valores por omissão existem em: ADA, C++, FORTRAN 90 e Visual Basic.
  - Ex. em C++:

// calculo do valor máximo de 1, 2 ou 3 nº positivos

Protótipo:

```
int maximo(int x,int y=0,int z=0);
```

Invocação:

```
cout << maximo(5) << maximo(5, 7);
cout << maximo(4, 8, 9);
```

Nota: Um valor por omissão é utilizado caso não seja especificado o corresponde parâmetro actual.

Fundamentos de Subprogramas

- Tipos de Subprogramas:
  - Procedimentos – Conjunto de instruções parameterizadas que definem uma determinada abstracção.
  - Funções – Semelhante aos procedimentos, mas geralmente modelam funções matemáticas. Se não produzirem efeitos colaterais, o valor devolvido é o seu único efeito.

Fundamentos de Subprogramas

- Considerações de Projecto de subprogramas:
  I. Variáveis locais são estáticas ou dinâmicas?
  II. Que métodos de passagens de parâmetros existem?
  III. Os tipos dos parâmetros formais são verificados com os tipos dos parâmetros actuais?
  IV. Parâmetros formais podem ser do tipo subprograma?
  V. Pode-se ter aninhamento de definição de subprogramas?

Fundamentos de Subprogramas

VI. Se um subprograma é transmitido como parâmetros, os seus parâmetros são verificados em relação ao tipo?

VII. Os subprogramas podem ser sobrecarregados?

VIII. Subprogramas podem ser genéricos (em relação ao tipo dos seus parâmetros formais)?

IX. É possível a compilação separada ou independente?

Ambientes de Referência Local

- Variáveis locais – são variáveis que são definidas dentro de um subprograma, e geralmente têm o mesmo escopo do subprograma.

  Implementação de variáveis locais:

  - Em ambiente de Pilha dinâmica:
    - O ambiente local é criado em cada activação;
    - Desvantagens: Tempo de reservar e libertar memória; Endereçamento indirecto; Subprogramas não são sensíveis à historia.
    - Vantagens: Permitem recursividade; A memória utilizada pela pilha pode ser compartilhada entre subprogramas.

Ambientes de Referência Local

- Em ambiente local estático;
  - Desvantagens: Não permite recursividade,
  - Vantagens: não há endereçamento indirecto (mais eficiente); não existe reserva e libertação de memória; é sensível à história.
  - Ex. de programa sensível à história em C++:

```
int rand (void) // gerar nºs aleatórios
{static int seed =5005;//variável estática
 seed=(((seed*214013L+2531011L)>>16)&0x7fff);
 return seed;
} //ex. adaptado da biblioteca do C++
```

Ambientes de Referência Local

- Linguagens que permitem ambiente estático:
  - COBOL, muitas versões de FORTRAN, opção especial em PL/I e Algol.
  - FORTRAN 77 e 90 – Quase sempre são alocações estáticas (pode haver alocação dinâmica Pilha).
  - C - Por omissão é dinâmico de pilha mas variáveis podem ter atributo static;
- Linguagens que permitem ambiente de Pilha dinâmica:
  - C/C++, Pascal, Ada, Lisp, APL, SNOBOL, etc.

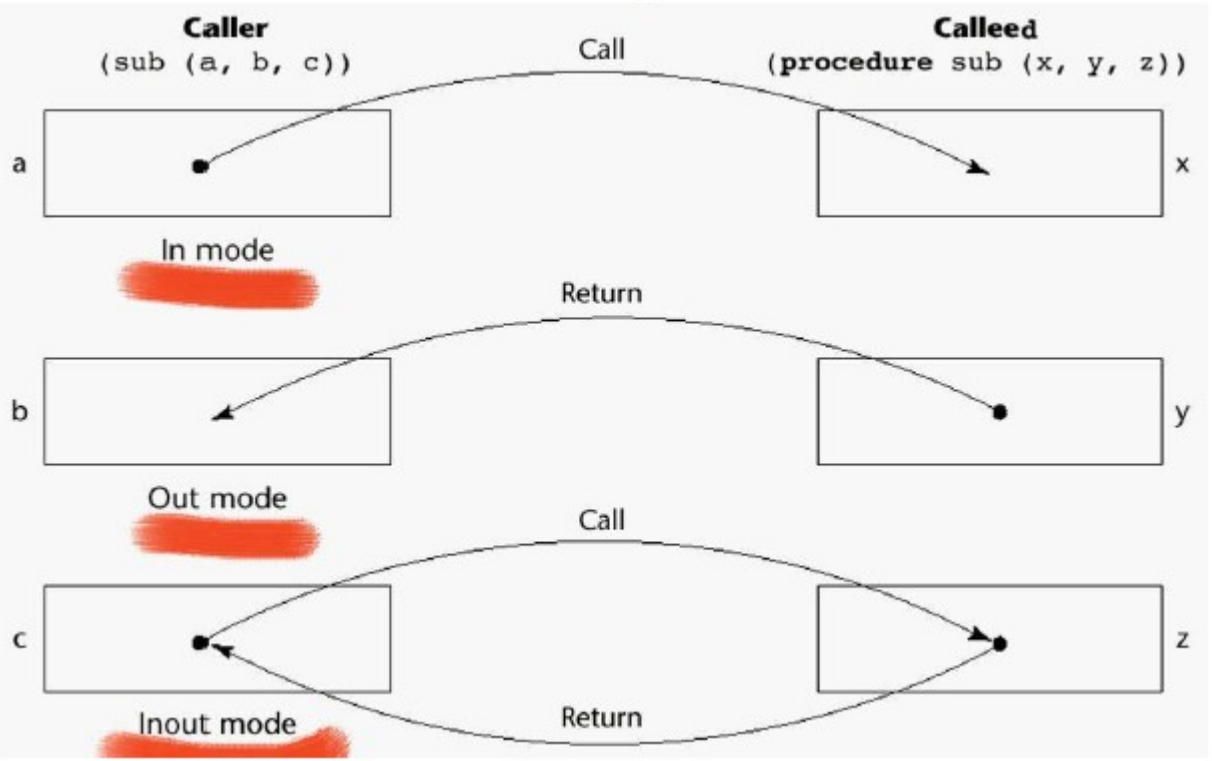
  É a situação geral de linguagens estruturadas.

Passagem de Parâmetros

- Métodos de Passagem de Parâmetros – formas de transmitir os parâmetros para os subprogramas.
- Modelo Semântico de Paragem de Parâmetros:
  - Modo de entrada (*in mode*);
  - Modo de saída (*out mode*);
  - Modo entrada/saída (*inout mode*).

Modelos de Implementação P. P.

Modelo Semântico de Paragem de Parâmetros



Modelos de Implementação P. P.

- Modelo Conceptual na Transmissão de Parâmetros:
  - Transferir fisicamente valores (copiar valores);
  - Transferir caminho de acesso (copiar endereço).
- Modelos de Implementação de Passagem de Parâmetros:
  1. Passagem por valor (modo de entrada)
  2. Passagem por resultado (modo de saída)
  3. Passagem por valor/resultado (modo de entrada/saída)
  4. Passagem por Referência (modo entrada/saída)
  5. Passagem por nome (modo múltiplo)

Modelos de Implementação P. P.

1. Passagem por valor (modo de entrada):
   - Cópia física - o parâmetro actual é avaliado e o seu valor é copiado para o parâmetro formal.
   - Se transmitido por Referência:
     - Necessário proteger parâmetro contra escrita pelo subprograma;
     - Acessos aos parâmetros são mais caros (via endereçamento indirecto);
   - Desvantagens passagem por valor:
     - Desperdício de memória (ex. duplicação de um array);
     - Custo da transferência (ex. tempo de cópia do array).

Modelos de Implementação P. P.

2. Passagem por resultado (modo de saída)
   - Valores do subprograma são transmitidos de volta para o invocador;
   - Geralmente é utilizado Passagem por Valor;
   - Deve ser assegurado que o valor do parâmetro formal não é utilizado no subprograma invocado;
   - Desvantagens:
     - Tempo e espaço de mem. (Passagem por Valor);
     - Colisão de parâmetros actuais. Ex.: sub(p1, p1).

Modelos de Implementação P. P.

3. Passagem por valor/resultado (modo de entrada/saída)
   - Transferência de valores em ambas as direcções;
   - Também conhecido por transmissão por cópia;
   - Desvantagens:
     - Mesmas que transmissão por resultado;
     - Mesmas que transmissão por valor.

Modelos de Implementação P. P.

4. Passagem por Referência (entrada/saída)
   - Transmite o caminho de acesso (endereço);
   - Parâmetro real é partilhado com o subprograma invocado;
   - Vantagem:
     - A transmissão de parâmetros é eficiente.
   - Desvantagens:
     - Acesso menos eficiente aos parâmetros (endereçamento indirecto);
     - Pode permitir aliasing.

Modelos de Implementação P. P.

- Ex. de problemas de *aliasing*:
  - I. Colisão de parâmetros actuais (em C++):

```
void fun(int * x,int * y)
{ x = 0;
  y = 1;
}
```

Invocação:

```
int total;
fun(& total, & total);
```

Qual o resultado final da variável `total`?

Modelos de Implementação P. P.

- II. Colisão de elementos de Array:

Invocação:

```
sub1(a[i], a[j]); // se (i==j) => alias
```

- III. Colisão entres parâmetros formais e variáveis globais (em Pascal):

```
procedure grande
  var global : integer;
   procedure pequeno(var local:integer)
   begin
       local := 10; // altera variável global
   end;             // são sinónimos (alias)
 begin
       pequeno(global);
 end.
```

Modelos de Implementação P. P.

5. Passagem por nome (modo múltiplo)
   - Parâmetro formal é substituindo textualmente pelo parâmetro real;
   - Parâmetros Formais são vinculados a um método de acesso no momento de chamada, mas a vinculação real a um valor ou endereço é retardada até que uma referência ou atribuição ao parâmetro formal seja realizada;
     - Propósito: flexibilidade de vinculação tardia (só quando é realmente necessário).

Modelos de Implementação P. P.

- Semântica Resultante:
  - Se o parâmetro actual for uma variável escalar o efeito é o de transmissão por referência;
  - Se for uma constante é por valor;

Passagem de Parâmetro

- Exemplo do C:
  - Transmissão somente por valor. Porém transmitindo-se o endereço e o efeito é "semanticamente semelhante" à passagem por referência.

```
void swap(int *x, *y)
{ int temp = *x;
  *x = *y;
  *y = temp;
}
swap(&a,&b);
```

Passagem de Parâmetro

- Exemplo do Pascal e Modula-2:
  - Passagem por valor e por referência através do atributo ***var*** no parâmetro formal.

```
procedure swap(var x: integer; var y: integer);
var temp : integer;
begin
    temp := x;
    x := y;
    y:= temp;
end;
```

Invocação:

```
swap(a,b);
```

Passagem de Parâmetro

- Exemplo do C++:
  - Passagem de valor como em C , contudo, também permite passagem por referência explicita através do operador '&' no parâmetro formal.

```
void swap(int & x, int & y)
{ int temp = x;
   x = y;
   y = temp;
}
```

Invocação:

```
swap(a,b); // para o valor de 2 variáveis
```

Passagem de Parâmetro

- Exemplo do Ada:
  - Todos os três modos estão disponíveis:
    - Se **out**, não pode ser referenciado;
    - Se **in**, não pode ter um valor atribuído;
    - Se **in out**, pode ser referênciado e atribuido;

```
procedure soma(     a : in out integer;
                    b : in integer;
                    c : out float)
```

- Exemplo do Java:
  - Como em C/C++, a passagem efectua-se por valor. Como objectos só podem ser passados por referência, efectivamente estes são passados a referência.

Passagem de Parâmetro

- Verificação do Tipo dos Parâmetros:
  - Actualmente é considerado muito importante a verificação do tipo dos parâmetros por questões de confiabilidade.
  - Exemplos de Linguagens:
    - FORTRAN 77 e C original: nenhuma verificação.
    - Pascal, Modula-2, FORTRAN 90, Java e Ada: está sempre presente.
    - ANSI C/ C++: em geral presente mas pode ser omitida pelo programador;

Passagem de Parâmetro

- Implementação de Métodos de Passagem de Parâmetros:

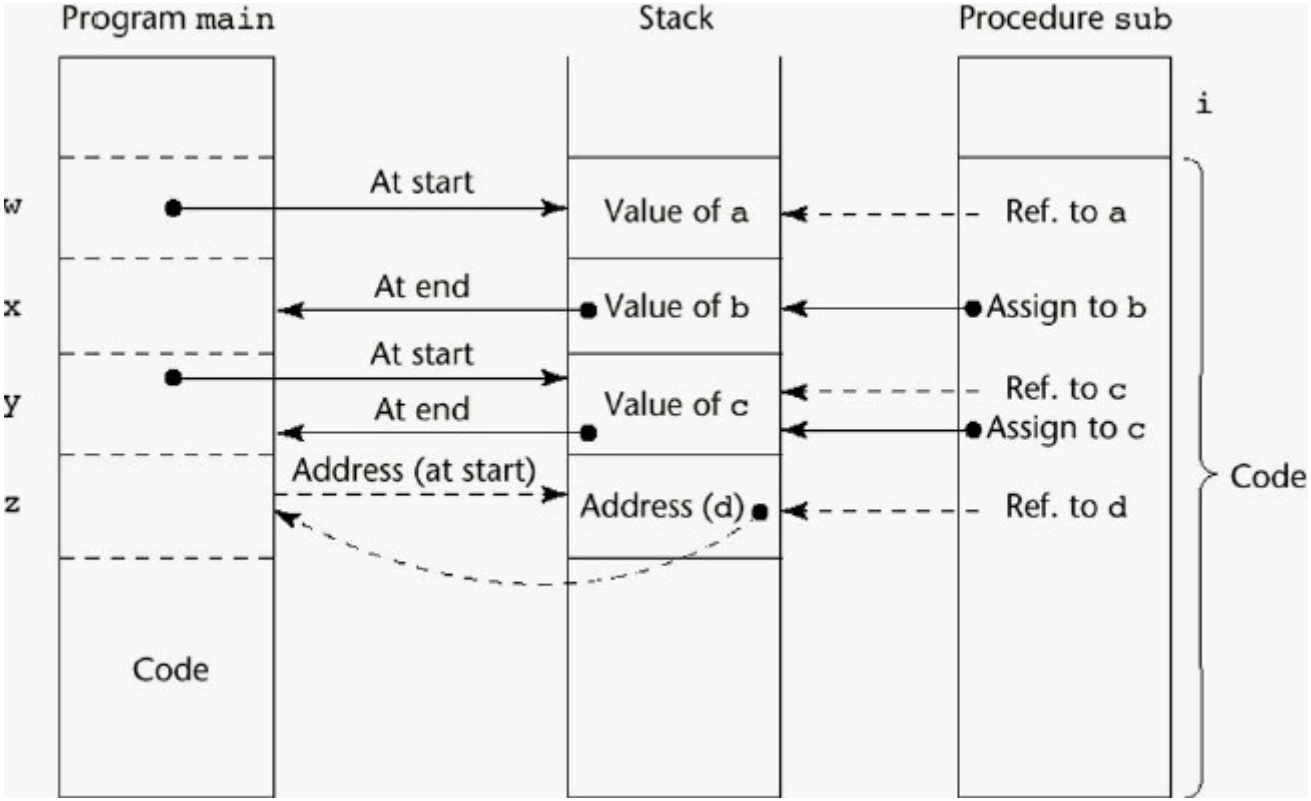
  ALGOL 60 e quase todos dos seus descendentes usam a pilha dinâmica de execução.

  - Passagem por Valor e
  - Passagem por Resultado:
    - Cópia dos valores para a pilha;
    - Referencias são indirectas na pilha;
  - Passagem por Referência:
    - Cópia do endereço para a pilha;

Passagem de Parâmetro

- Passagem por Nome:
  - Segmento de código (chamado de ***thunks***) residente em tempo de execução que avalia o endereço do parâmetro actual, e referencia cada referência ao parâmetro formal.
  - Extremamente caro comparado com referência ou valor-resultado.

Pilha Dinâmica de Execução



Passagem de Parâmetro

- Array Multidimensional como Parâmetro:
  - Quando um array multidimensional é passado como parâmetro, e o subprograma é compilado separadamente, o compilador necessita conhecer o tamanho do array para construir a função de mapeamento.
  - Exemplo do C/C++:
    - Programador necessita declarar os tamanhos das dimensões, excepto da primeira. Este facto geral menor flexibilidade na programação.

Array multidimensional como Parâmetro

- 1º Ex. em C (Primeira dimensão não incluída):

```
void soma(int matriz[][10])
{int i, j, sum = 0;
  for (i = 0 ; i < 10; i++)...
}
 void main()
{ int  m[5][10];
    ...
  Soma (a); // Função de mapeamento:
} // Seja S = sizeof(int), L = linha, C = coluna
m[L, C] <=> *(&m[0][0] + (10 * S * L) + S * C)
```

É necessário um novo subprograma para uma matriz com um nº diferente de colunas

Array multidimensional como Parâmetro

- 2º Ex. em C (Solução para o problema da dimensão fixa de colunas):
  - Passar um apontador para o array e o tamanho das dimensões como parâmetros.

```
void fun(float *mat, int linhas, int cols);
```

  - e incluir a função de mapeamento em termos do tamanho dos parâmetros. Função de mapeamento (mat[i][j]):

```
s = sizeof(float)
mat[i][j] = *(mat+(s*i*cols)+s*j);
```

Array multidimensional como Parâmetro

- Ex. do Pascal:
  - Tamanho declarado é parte integrante do tipo array. O tipo array deve ser conhecido na declaração do subprograma.

```
type mat = array[1..10,1..20] of integer;
procedure proc( mm: mat);
begin ... end;
```

- Ex. do Java:
  - Semelhante a ADA. Em JAVA arrays são objectos, onde cada array herda uma constante (*lenght*) com o tamanho do array quando o objecto é criado.
  - O parâmetro formal é a variável com colchetes vazios.
  - Cada dimensão pode ser obtida no subprograma usando-se o atributo *length*.

Subprogramas Sobrecarregados

- Sobrecarga de subprograma – é um subprograma que possui o mesmo nome doutro subprograma num mesmo ambiente de referência (escopo), contudo têm de ter protocolos diferentes.
  - C++ e Ada possuem sobrecarga de subprogramas predefinidos e programadores podem implementar novos subprogramas sobrecarregados.
  - Ex. em C++:

```
int maximo(int x, int y)
{ return x>y ? x : y; }
double maximo(double x, double y)
{ return x>y ? x : y; }
int maximo(int x, int y, int w, int z)
{ return maximo(maximo(x,y),maximo(y,z));}
```

Subprogramas Genéricos

- Um subprograma genérico ou polimórfico é aquele que recebe parâmetros de tipos diferentes em diferentes activações.
- Sobrecarga de subprograma é um mecanismo de polimorfismo *ad hoc* (para um fim especifico).
- Polimorfismo paramétrico - subprogramas utilizados em expressões, os quais recebem parâmetros de diferentes tipos especificados pelos operandos da expressão.
  - Ex. ADA e C++ utilizam polimorfismo paramétrico.

Ex. de Função Genérica em C++

- Ex. de Função genérica em C++:

```
template <class Type>
Type swap(Type & first, Type & second)
{ Type aux = first;
  first = second;
  second = aux;
}
```

Tipo Genérico

- Funções C++ genéricas (*template*) são instanciadas implicitamente quando seu nome é utilizado numa invocação ou quando o seu endereço é obtido com o operador '&'.

Ex. Ordenação Genérica em C++

```
template <class Type>
void generic_sort(vector<Type> & v)
{
  for(int iter=0;  iter < v.size()-1; iter++)
  {
    for(int col=iter+1; col < v.size()-1; col++)
    {  if(v[iter] > v[col])
         swap(v[iter], v[col]);
    }
  }
}
```

Tipo Genérico

Combinação entre Bubble Sort e Selection Sort.

Exemplo de invocação:

```
vector<int> idades;
...
generic_sort(idades);
```

Compilação Separada e independente

- Compilação Independente – Unidades de compilação que podem ser compiladas separadamente, sem qualquer informação sobre as outras unidades (interdependências). A coerência de tipos das interfaces não é verificada entre as unidades compiladas.
- Compilação Separada – Unidades de compilação que podem ser compiladas separadamente, utilizando informações de interface para verificar a interdependência entre as partes.
- Exemplos de linguagens:
  - C, FORTRAN 77: Compilação independente;
  - FORTRAN 90, Ada, Modula-2, C++: Compilação separada;
  - Pascal: Programa tem que ser todo compilado de uma única vez.

Questões de Projeto

- Considerações de projecto para Funções:
  - Efeitos colaterais permitidos?
    - Parâmetros de dupla via? - Ada não permite;
    - Referência não local? - todas L.P. permitem;
  - Que tipo de valores podem ser devolvido?
    - FORTRAN, Pascal e Modula-2 – só tipos simples;
    - C – qualquer tipo excepto funções e arrays;
    - Ada – qualquer tipo (subprogramas não são considerados tipos);
    - C++ e Java – como em C, mas classes podem ser devolvidas (retornadas).

Acesso a Ambientes não locais

- Variáveis não locais de um subprograma são aquelas que são visíveis (podem ser utilizadas) mas não estão declaradas no subprograma.
- Variáveis Globais são aquelas que podem ser visíveis em todos os subprogramas.
  - Ex. em FORTRAN (Blocos *COMMON*):
    - Antes do FORTRAN 90, os blocos COMMON eram a única forma de aceder a variáveis não locais.
    - Estes blocos podem ser utilizado para partilhar dados ou partilhar memória.

Acesso a Ambientes não locais

- Ex. em C (declaração *extern*):
  - Variáveis globais são criadas pela declaração *extern* (definidas fora de qualquer função);
  - Declarações da variáveis noutros módulos, definem o tipo da variável externa, mas não indica onde está definida.

```
//File: menu.c
// Criar variável global
char * MenuID = "File";
...
printf("MenuID=%s",MenuID);
```

```
//File: applic.c
//declaração de variável externa
extern char * MenuID;
...
printf("MenuID=%s",MenuID);
```

Sobrecarga de Operadores

- Operadores sobrecarregados – operadores que têm múltiplos significados, isto é, estão definidos para diversos tipos de operandos.
  - Ex.: 25 + 40 (int + int) e 25.0 + 40.0 (real + real)
- Os operados sobrecarregados devem ter protocolos diferêntes.
- Quase todas as linguagem de programação possuem operadores sobrecarregados.
- Programadores podem criar novos significados para operadores em C++ e ADA (esta facilidade não existe em Java).

Sobrecarga de Operadores

- Exemplo em Ada:
  - Assumir que VECTOR_TYPE já foi definido como um tipo array com elementos INTEGER.

```
function "*"(A,B : in VECTOR_TYPE)return INTEGER is
   SUM : INTEGER := 0;
   begin
   for INDEX in A'range loop
     SUM := SUM + A(INDEX) * B(INDEX);
   end loop;
   return SUM;
end "*";
```

A sobrecarga é boa se a legibilidade não for afectada.

Sobrecarga definida pelo programador é bom ou não?

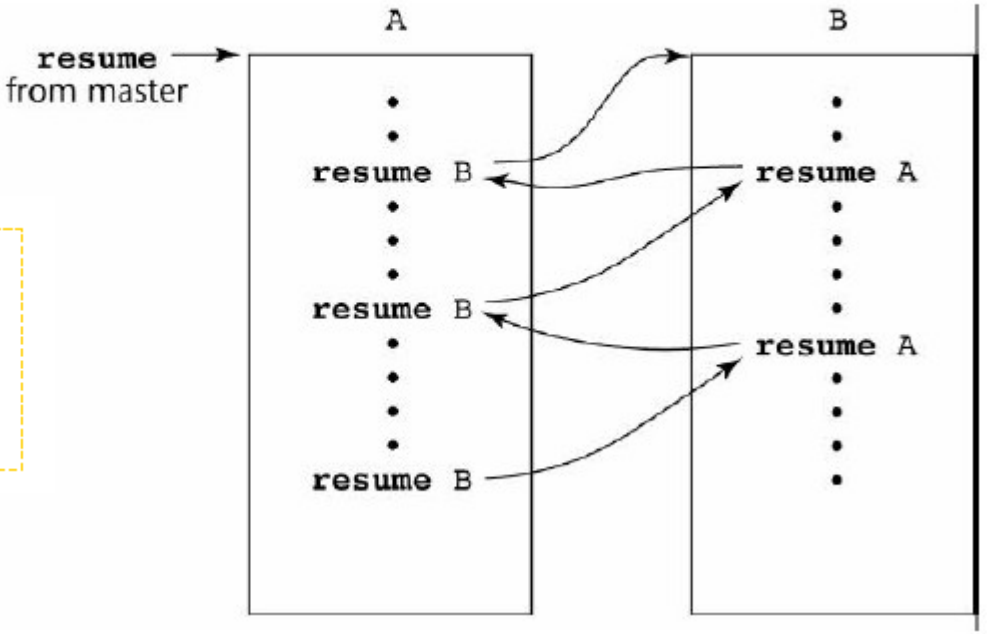
Co-rotinas

- Uma co-rotina é um subprograma que possui múltiplas entradas.
- Quanto uma co-rotina é activada, através de outro subprograma, esta é executa parcialmente sendo suspensa quando retorna o controle, podendo ser reactivada de onde parou (se invocada novamente).
- A invocação duma co-rotina é designado de retomada (*resume*).
- A primeira retomada de uma co-rotina é para seu início mas, invocações subsequentes iniciam imediatamente após o último comando executado.

Co-rotinas

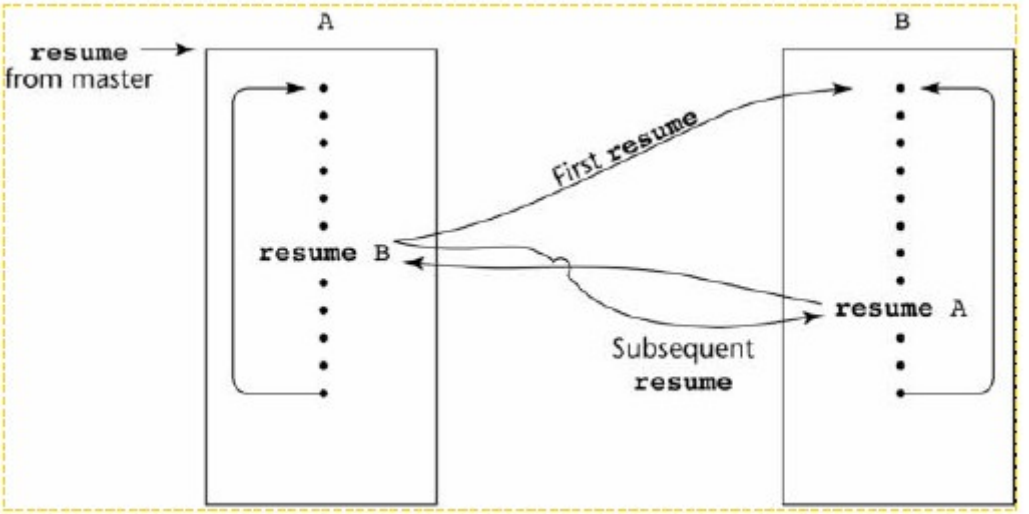
- Tipicamente, co-rotinas repetidamente retomam a execução entre si, possivelmente em laço infinito.

Execução de duas co-rotinas sem laços.



Co-rotinas

Execução de duas co-rotinas com laços.



- Co-rotinas fornecem um mecanismo de execução de unidade de programas quase concorrentes.
- A execução de co-rotinas é intercalada e não sobreposta.