Introdução à Programação
Orientada a Objetos com Java

Programação Imperativa
(e Ponteiros e Registros)

Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco

Na programação imperativa...

- Não temos classes nem objetos
- Não temos métodos nem atributos
- Só temos
  - Funções (correspondem aos métodos)
  - Variáveis (correspondem às variáveis locais e às variáveis estáticas)

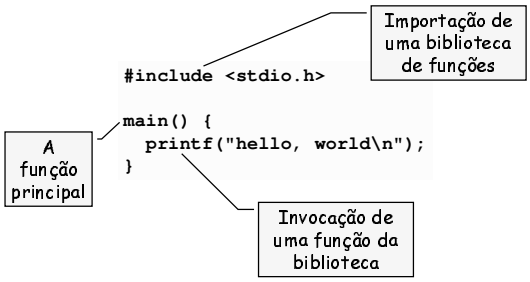
Não temos information hiding (mas C simula)

Um programa imperativo...

Contém
- Uma função principal, chamada **main**, por onde começa a execução do programa
- Várias funções auxiliares, para modularizar, dividir o código em partes
- Importação de bibliotecas de funções, que correspondem aos pacotes de Java

Mas sem controle de visibilidade e espaço de nomes (mas Modula 2 tem)

Um programa em C



Definindo e chamando funções auxiliares

```
#include <stdio.h>

void printString(char s[]) {
  printf(s);
}

main() {
  printString("hello,");
  printString("world");
  printString("\n");
}
```

Função auxiliar, recebendo um array de **char**, o que corresponde a **String** em C

Termina com '\0'

Nas funções podemos ter...

- Variáveis locais
  - usando tipos correspondentes aos tipos primitivos de Java, com diferenças mínimas
- Atribuições, como em Java
- Estruturas de controle
  - praticamente as mesmas de Java
- Chamadas de funções
- ...

A função para multiplicação

```
int multiply (int a, int b) {
    int count = abs(a);
    int n = abs(b);
    int mult = 0; int i;
    for (i = 0;i < n; ++i)
       mult = mult + count;
    if ((a > 0 &&  b < 0) ||
        (a < 0 && b > 0)) {
      mult = mult * (-1);
    }
    return mult;
 }
```

Função da biblioteca padrão, importada automaticamente, que devolve o valor absoluto

Nas funções **não** podemos ter...

- **new**
- **this**
- **super**
- **throw**
- **instanceof**
- **boolean**
- ...

Também não temos overloading, polimorfismo, dynamic binding, etc.

Ao invés de referências para objetos...

Nas funções podemos ter acesso a referências para variáveis

Ponteiros

- Toda variável tem um endereço e uma posição associados na memória
- Este endereço é visto como um ponteiro, uma referência, para o conteúdo da variável, da posição de memória
- Este endereço pode ser armazenado em uma variável do tipo ponteiro

Variáveis e endereços

- Memória abstrata:
  `{x→5, y→9, z→'a'}  (Id→Valor)`
- Memória concreta:
  - Associações:
    `{x→13, y→72, z→00}  (Id→Ref)`
  - Memória de fato:
    `{00→'a',...,13→5,  (Ref→Valor)`
    `72→9,...,99→`*undefined*`}`

Operadores para ponteiros

- **&x** (o endereço da variável **x**)
- **int *px;** (a definição da variável **px** que armazena endereços de variáveis inteiras)
- **px** (o endereço armazenado em **px**)
- ***px** (o conteúdo do endereço armazenado em **px**)

Ponteiros em C

```
int i, j;
int *ip;
i = 12;
ip = &i;
j = *ip;
*ip = 21;
```

A variável ip armazena um ponteiro para um inteiro

O endereço de i é armazenado em ip

O conteúdo da posição apontada por ip é armazenado em j

O conteúdo da posição apontada por ip passa a ser 21

A passagem de parâmetros em C é por valor...

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

```
int a,b;
a = 8;
b = 12;
swap(a,b);
```

A chamada da função não afeta os valores de a e b

Mas temos o equivalente à passagem por referência

```
void swap(int *px, int *py){
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

```
int a,b;
a = 8;
b = 12;
swap(&a,&b);
```

A chamada da função **afeta** os valores de a e b

Leitura de dados com **scanf**

```
int numero;
printf("Digite um decimal:\n")
scanf("%d",&numero);
```

Formato e tipo da informação a ser lida (`c`, `s`, `f`, etc.)

Endereço da variável aonde a informação lida deve ser armazenada depois de convertida

Escrita de dados com **printf**

```
printf("numero: %s, saldo: %.2f",
       n, v);
```

Informações a serem escritas

Tipos e formatos das informações a serem escritas (`%d`, `%6d`, `%f`, `%6.2f`, etc.)

No começo do arquivo podemos declarar constantes

```
#define MAXIMO 100
#define absoluto abs
#define max(A,B) ((A)>(B)?(A):(B))
```

Realiza a substituição de um texto por outro, considerando parâmetros

Variáveis externas

- Também chamadas de variáveis globais pois podem ser acessadas em qualquer função
- São definidas fora das funções e declaradas com o modificador **extern** em cada função que as utiliza
- Correspondem às variáveis estáticas públicas de Java

static

Significado diferente do de Java:

- As variáveis locais estáticas continuam existindo após a execução de uma função
- As variáveis externas estáticas só podem ser acessadas no arquivo aonde foram definidas

Também válido para funções

Equivalente a private

Arrays

```
int numeros[10];
numeros[0] = 5;
char numero[11];
```

O tamanho tem que ser especificado na declaração de uma variável, mas não na declaração de um parâmetro

```
void printStr(char s[]){
  printf(s);
}
```

Ponteiros e arrays

Arrays podem ser tratados como ponteiros em C!

```
int a[10];
int *pa;
pa = &a[0];
pa = a;
```

Equivalentes!

→ *pa == a[0] == pa[0]
→ *(pa+i) == a[i] == pa[i] == *(a+i)
→ a+i == &a[i]

~~a = pa~~
~~a++~~

Ao invés de classes, estruturas (registros)!

```
struct Conta {
  double saldo;
  char numero[11];
};
```

Define um tipo, sem information hiding, sem funções, sem construtor, sem herança, sem subtipo

Lista-se apenas os componentes de cada elemento do tipo

Criando estruturas estaticamente

Cria e inicializa

```
struct Conta c = {0,"1"};
```

Cria mas não inicializa

```
struct Conta d;
```

Acesso e atualização de partes da estrutura

```
d.saldo = c.saldo;
strcpy(d.numero,"2");
```

Função da biblioteca padrão de C: atualização de arrays é seletiva

Manipulando estruturas

```
struct Conta creditar(struct Conta x,double v){
  x.saldo = x.saldo + v;
  return x;
}
```

```
struct Conta c = {0,"1"};
struct Conta d;
d = c;
c = creditar(c,10);
```

A estrutura é copiada, não trabalha-se com referências

A passagem de parâmetros é por valor; é necessário retornar o resultado

Manipulando referências para estruturas

```
void creditar(struct Conta *c,double v){
    (*c).saldo = (*c).saldo + v;
}
```

```
struct Conta c = {0,"1"};
struct Conta d;
d = c;
creditar(&c,10);
```

A estrutura é copiada

Passa-se a referência; os efeitos da execução do método são refletidos em c

Notação especial para manipular referências para estruturas

```
void creditar(struct Conta *c,double v){
    c -> saldo = c -> saldo + v;
}
```

Estruturas complexas

```
struct Endereco {
  char rua[40];
  char complemento[10];
  char cep[10];
  char cidade[20];
  char estado[20];
};
struct Pessoa {
  char nome[35];
  struct Endereco *endereco;
  struct Pessoa *conjuge;
};
```

Permitindo compartilhamento de estruturas e recursão

Criando estruturas dinamicamente

```
struct Conta *pc;

pc = (struct Conta *)
     malloc(sizeof(struct Conta));
```

Não cria a estrutura mas sim uma variável ponteiro

Cast

Cria a estrutura: aloca memória dinamicamente

Indica o espaço necessário para armazenar um elemento do tipo

Simulando **new**

```
struct Conta *newConta(char *num,double v){
  struct Conta *retorno;
  retorno = (struct Conta *)
            malloc (sizeof(struct Conta));
  if (retorno == NULL) {
    fprintf("Erro na alocacao de memoria!");
  } else {
     strcpy(retorno->numero, num);
     retorno->saldo = v;
  }
  return retorno;
}
```

Gerando lixo

```
struct Conta *pc, *pd;
pc = (struct Conta *)
      malloc(sizeof(struct Conta));
pd = (struct Conta *)
      malloc(sizeof(struct Conta));
pc = pd;
```

A primeira estrutura criada não pode ser mais acessada, vira lixo! A memória não será liberada...

Eliminando lixo

```
struct Conta *pc, *pd;
pc = (struct Conta *)
      malloc(sizeof(struct Conta));
pd = (struct Conta *)
      malloc(sizeof(struct Conta));
free(pc);
pc = pd;
```

O programador é responsável pelo gerenciamento da memória: libera a memória da primeira estrutura criada

Ao invés de supertipos, unions

```
union ContaPoupanca {
  struct Conta c;
  struct Poupanca p;
};

union ContaPoupanca cp;
cp.c = conta;
cp.p = poupanca;
```

Os elementos deste tipo podem ser tanto contas quanto poupanças

Perde-se a conta armazenada em cp

Simulando `instanceof`

```
struct ContaGeral {
  int tipocontageral;
  union ContaPoupanca contageral;
};

struct ContaGeral cg;...
if (cg.tipocontageral == 0) {
  cg.contageral.c.saldo = 0;
} else {
  cg.contageral.p.juros = 0;
}
```

Indica o tipo da informação armazenada na union `contageral`

Uma conta armazenada?