C++ Orientado a Objetos
Ponteiros

Prof. Flávio de Oliveira Silva, M.Sc.

flavio@facom.ufu.br

flaviosilva@computer.org

Ponteiros
Declaração

- As variáveis em C++ podem ser de duas diferentes naturezas
- Uma variável pode conter um **VALOR** ou um **ENDEREÇO**
- VALOR
  - A declaração de uma variável que pode conter um valor é feita da seguinte forma:

    ```
    Tipo nomeVariavel;
    ```

    - Tipo – Qualquer tipo básico da linguagem (int, float, char, double etc.) ou o nome de uma classe
    - nomeVariavel – Nome válido de uma variável
- ENDEREÇO
  - A declaração de uma variável que pode conter um endereço é feita da seguinte forma:

    ```
    Tipo *nomeVariavel;
    ```

    - Tipo – Qualquer tipo básico da linguagem (int, float, char, double etc.) ou o nome de uma classe
    - nomeVariavel – Nome válido de uma variável, precido pelo asterisco (*)

Ponteiros
Declaração

- Uma variável que contém um endereço de mémoria é chamada Ponteiro
- Esta variável contém um endereço que neste caso “aponta” para uma outra variável ou objeto
- O uso de ponteiros é um recurso poderoso da linguagem C++
- Ponteiros para variáveis podem ser alocados de forma dinâmica, ou seja, durante a execução do programa.
- Como o ponteiro representa o endereço do objeto é mais eficiente passar para um método um ponteiro para aquele objeto do passar uma cópia do objeto, que utiliza maior quantidade de memória.
- Declaração de ponteiros - Exemplos

```
int *pValor; //contém o endereço de um int
//variável pValor conterá um endereço de um inteiro
int *pInt, i //Declara um ponteiro para inteiro e um inteiro
double* pdValue1, pdValue1; //dois ponteiros para double
char* pString;
```

Ponteiros
Atribuição

- Um ponteiro pode receber apenas um endereço.
- Como então obter um endereço de uma variável?
- Para isto é utilizado um operador da linguagem - &
- O operador &(endereço de) retorna o endereço de uma variável
- O endereço retornado pode então ser atribuído ao ponteiro

```
int iV (34);  //declara um inteiro
int *pV;      //declara um ponteiro
pV = &iV;
//A linha acima pode ser lida como:
//o ponteiro pV recebe o (endereço de) iV
             pV    =          &        iV;
```

Ponteiros
Inspeção do endereço

- O ponteiro contém o endereço de uma outra variável
- Será que é possível, a partir do ponteiro, saber qual o conteúdo desta variável?
- Isto pode ser feito com o operador * (asterisco) que significa “o conteúdo do endereço”

```
int iV(34), i;        //declara dois inteiros
int *pV;              //declara um ponteiro
pV = &iV;
//A linha acima pode ser lida como:
//o ponteiro pV recebe o (endereço de) iV
//           pV    =          &        iV;
i = *pV;
//A linha acima pode ser lida como:
//a variável i recebe o (conteudo do endereço em) pV
//           i  =                *                pV;
```

Ponteiros
Atribuição e Inspeção - Exemplo

```
int main(int argc, char *argv[]){
    int iV (34);  //declara um inteiro
    int *pV;      //declara um ponteiro
    pV = &iV;     //ponteiro recebe o endereço da variável
    cout << "&iV: " << &iV << " - iV: " << iV << endl;
    cout << "&pV: " << &pV << " - pV: " << pV << endl;
    cout << "*pV: " << *pV << endl;
    system("PAUSE");
    return 0;
}
```

MEMÓRIA

	ENDEREÇO	VALOR
pV	0x22FF34	0x22FF38
iV	0x22FF38	34

- **RESULTADO:**

```
&iV: 0x22ff38 - iV: 34
&pV: 0x22ff34 - pV: 0x22ff38
*pV: 34
```

- A impressão do ponteiro retorna um endereço em hexadecimal

Ponteiros
Operadores: * e & - Resumo

- Declaração
  - Operador * - Indica que a variável contém um endereço de uma outra variável do mesmo tipo
    - Variável e dita um **Ponteiro**
  - Operador & - Indica que a variável é uma cópia de outra variável
    - Variável é dita uma **Referência**
- Uso – Para uma variável do tipo ponteiro
    - `int v1,`
    - `int *pInt1;`
  - Operador * - Conteúdo do endereço contido em
    - `v1 = *pInt1; //v1 recebe conteúdo do endereço contido em pInt1`
  - Operador & - Endereço desta variável
    - `pInt1 = &v1; //pInt1 recebe "endereço" de v1`

Ponteiros
Inicialização

- Um ponteiro sempre é inicializado com um endereço qualquer.
- Para atribuir um endereço pode ser utilizado o operador “&” (endereço de) ou uma simples atribuição

```
int v1,
int *pInt1, *pInt2;
pInt1 = &v1;      //pInt1 recebe "o endereço de" v1
pInt2 = pInt1;    //Ambos são ponteiros para o mesmo tipo
```

- É possível atribuir o valor NULL a um ponteiro desta forma ele não contém nenhum endereço válido

```
char* pString = NULL;
```

- Uma condição muito comum é comparar um ponteiro com o valor NULL e caso seja verdadeiro então um endereço pode ser atribuído ao mesmo

```
if (pString ==NULL){
    //atribui um endereço ao ponteiro ;
}
```

Ponteiros - Tipos

- O ponteiro pode apontar para qualquer tipo básico da linguagem e também para objetos de qualquer classe. Sendo assim um ponteiro está sempre associado a algum tipo.

```
double *pDouble;
```

- C+ + suporta um tipo especial de ponteiro, chamado ponteiro void, que pode apontar para objetos de qualquer tipo.

```
void* pQualquer;
void* pVetorVariosTipos[32];
```

- É possível a criação de vetores de ponteiros. Um vetor de ponteiros pode ser manipulado da mesma forma que qualquer outro vetor. Porém o que este vetor contém é endereços de outros objetos
- Para acessar métodos de um ponteiro para um objeto deve ser utilizado operador ponteiro (->), conforme mostrado abaixo:

```
Circle c(3);
Circle *pC = &C;
pC->area();
*pC.area();
```

Ponteiros
Acesso Métodos e ou Valores

- Caso exista um ponteiro para um objeto, para acesso aos métodos do mesmo deve ser utilizado o operador ponteiro (->) e não o operador ponto (.)

```
Circle c(3);
Circle *pC = &c;
pC->area();
double d;
d = c.area();
```

- O mesmo operador pode ser utilizado para acessar campos de uma estrutura

```
struct Pessoa* pPessoa;
...
pPessoa->idade = 35;
double salario = pPessoa->salario;
```

Ponteiros
Tamanho de um Ponteiro

- O tamanho de um ponteiro equivale ao tamanho em bytes do tipo contido naquele ponteiro
- O tamanho de um tipo (básico / objeto) qualquer pode ser obtido com o operador **sizeof**

```
int i;
int *pI;
sizeof(i) //4
sizeof(pI) //4
double d;
double *pD;
sizeof(d) //8
sizeof(pD) //4
```


sizeof(pD) //8

Tamanho de um Ponteiro Exemplo

```
int main(int argc, char *argv[]){
        int iValor(34);   //declara um inteiro
        int *pValor;                    //declara um ponteiro
        pValor = &iValor;               //ponteiro recebe o endereço
        double *pD1, *pD2;
        double d=2.1;
        pD1 = pD2 = &d;
        cout << "&pD1: " << &pD1 << " - cpD1: " << pD1 << " - *pD1: " << *pD1 << endl;
        cout << "&pD2: " << &pD2 << " - cpD1: " << pD2 << " - *pD2: " << *pD2 << endl;
        cout << "size pValor: " << sizeof(pValor) << " | size pD2: " << sizeof(pD2);
        cout << " | size iValor: " << sizeof(iValor) << " | size d: " << sizeof(d) << endl;
}
```

- Saída
  - &pD1: 0x22ff30 - cpD1: 0x22ff20 - *pD1: 2.1
  - &pD2: 0x22ff2c - cpD2: 0x22ff20 - *pD2: 2.1
  - size pValor: 4 | size pD2: 4 | size iValor: 4 | size d: 8

Ponteiros e Arrays

- Uma variável que contém um array, na realidade é um ponteiro
- Ao declarar um array a variável conterá o endereço da primeira posição
- Em um array todas as posições são alocadas de forma contínua e isto reflete no endereço das posições seguintes a partir da primeira

```
int array[10];
int* pArray;
pArray = array //não é necessário utilizar o operador &
```

- Caso uma função (método) receba como parâmetro um array, a mesma receberá apenas o endereço da primeira posição.

Ponteiros
Aritmética

- Dependendo da natureza da variável os operadores (+ - *) terão diferentes comportamentos
- VALOR
  - Operadores se comportam como previstos na álgebra considerando que as variáveis são tipos básicos

```
int a(4), b(5);
a = a + b;
```

- ENDEREÇO
  - Operadores comportam como previsto na álgebra, porém a operação é aplicada a endereços e não aos valores

```
int *pA, *pB;
pA = pA + 1;      //ao endereço contido em pA + 1 * 4 (bytes)
pB = pB + 2;      //ao endereço contido em pB + 2 * 4 (bytes)
```

  - Operação acima adiciona 1 ao valor mas 1 vez o tamanho (sizeof) do tipo contido no ponteiro

Ponteiros
Aritmética

- Caso um conjunto de variáveis do tipo ponteiro sejam definidas a fim de utilizar um segmento contiguo de memória é possível o uso operadores e se movimentar entre os vários endereços;

```
int* pIntArray[32];
pIntArray+1;    //o próximo endereço de memória
pIntArray-1;    //endereço de memória anterior
pIntArray++;    //o próximo endereço de memória
pIntArray--;    //endereço de memória anterior
```

- Neste caso 100+1 não é igual a 101!
- Mas ao próximo endereço de memória, que no caso de um inteiro é igual a 104 (4 bytes)

pIntArray-1;

Ponteiros
Aritmética

- A aritmética de Ponteiros é bastante utilizada no manipulação de vetores

	ENDEREÇO	CONTEÚDO
array[0]	0x22fef0	0
array[1]	0x22fef4	1
array[2]	0x22fef8	4
array[3]	0x22fefc	9
array[4]	0x22ff00	16
array[5]	0x22ff04	25
array[6]	0x22ff08	36
array[7]	0x22ff0c	49
array[8]	0x22ff10	64
array[9]	0x22ff14	81
pEndArray		0x22fef0
	0x22ff10	

```
&array: 0x22fef0
array[0]: 0 - &array[0]: 0x22fef0
array[1]: 1 - &array[1]: 0x22fef4
array[2]: 4 - &array[2]: 0x22fef8
array[3]: 9 - &array[3]: 0x22fefc
array[4]: 16 - &array[4]: 0x22ff00
array[5]: 25 - &array[5]: 0x22ff04
array[6]: 36 - &array[6]: 0x22ff08
array[7]: 49 - &array[7]: 0x22ff0c
array[8]: 64 - &array[8]: 0x22ff10
array[9]: 81 - &array[9]: 0x22ff14
pEndArray: 0x22fef0
*pEndArray: 0
pEndArray+4: 0x22ff00
pEndArray+8: 0x22ff10
pEndArray+1: 0x22fef4
```

Ponteiros
Alocação dinâmica de memória

- A alocação dinâmica de memória é um meio eficiente, porém perigoso, de se trabalhar em C+ + .
- Ao alocar somente a memória necessária o programa utiliza somente a memória que vai efetivamente necessitar.
- A alocação estática, por sua vez, pode trazer problemas, por exemplo: Ao alocar 256 posições de memória para uma string pode acontecer de nunca se utilizar mais que 20 posições ou então pode ocorrer um estouro. Exemplo:

```
char sNom[256];
```

- Para alocar a memória utiliza-se o operado **new**, conforme mostrado abaixo:

```
int *pValor = new int;
Circle* pCircle = new Circle(3);
char* sNome = new char[256];
```

- Ao utilizar o operador **new**, o método construtor do objeto é automaticamente chamado

Ponteiros
Liberação dinâmica de memória

- Toda memória alocada pelo programa dever ser necessariamente liberada pelo mesmo!
- Para liberar a memória alocada utiliza-se o operador delete
  - `int *pValor = new int;`
  - `Circle* pCircle = new Circle(3);`
  - `char* sNome = new char[256];`
  - `...`
  - `delete pValor;`
  - `delete pCircle;`
  - `delete sNome;`
- As variáveis estáticas, alocadas pelo programa, são liberadas automaticamente, não sendo necessária sua remoção
- Ao utilizar o operador delete, o método destruidor do objeto é chamado

Arrays como ponteiros
Alocação Dinâmica

```
int main(int argc, char *argv[]){
        int *vetor = new int[7];
        int* iTemp;
        iTemp = vetor;
        cout << "&vetor: " << &vetor  << " - vetor: " << vetor  << endl;
        cout << "vetor: " << vetor << endl;
        for(int i=0;i < 7; i++){
          *iTemp = 10+i;
          iTemp++;
        }
        iTemp = vetor;
        cout << "&iTemp: " << &iTemp  << " - iTemp: " << iTemp  << endl;
        for(int i=0;i < 7; i++){
          cout << "iTemp: " << iTemp << " - *iTemp: " << *iTemp << endl;
          iTemp++;
        }
        delete [] vetor;
}
```

	ENDEREÇO	VALOR
i		0
vetor	endV	endXXX
	endXXX	10
		11
		12
		13
		14
		15
		16
iTemp		endXXX

Arrays como ponteiros
Alocação Dinâmica - Endereços

- Saida
  - &vetor: 0x22ff44 - vetor: 0x3729d8
  - &iTemp: 0x22ff40 - iTemp: 0x3729d8
  - iTemp: 0x3729d8 - *iTemp: 10
  - iTemp: 0x3729dc - *iTemp: 11
  - iTemp: 0x3729e0 - *iTemp: 12
  - iTemp: 0x3729e4 - *iTemp: 13
  - iTemp: 0x3729e8 - *iTemp: 14
  - iTemp: 0x3729ec - *iTemp: 15
  - iTemp: 0x3729f0 - *iTemp: 16

	ENDEREÇO	VALOR
vetor	0x22ff44	0x3729d8
	0x3729d8	10
	0x3729dc	11
	0x3729e0	12
	0x3729e4	13
	0x3729e8	14
	0x3729ec	15
	0x3729f0	16
iTemp	0x22ff40	0x3729d8

Referências

- Uma referência pode ser entendida com um “apelido” para um objeto
- Após inicializar uma referência, qualquer alteração no objeto e na referência ao mesmo serão percebidas em ambos.
- Abaixo é mostrado como criar uma referência:

```
int iNumber(43);
int &iNumberRef = iNumber; //declarando e inicializando
iNumber = 2;
cout << "iNumberRef: " << iNumberRef << endl; //imprime 2
iNumberRef = 3;
cout << "iNumber: " << iNumber << endl; //imprime 3
```

- A referência deve ser sempre inicializada isto pode ser feito somente uma vez.

Ponteiros
Operadores: * e &

- Declaração
  - Operador * - Indica que a variável contém um endereço de uma outra variável do mesmo tipo
    - Variável e dita um **Ponteiro**
  - Operador & - Indica que a variável é uma cópia de outra variável
    - Variável é dita uma **Referência**
- Uso – Para uma variável do tipo ponteiro
    - `int v1,`
    - `int *pInt1;`
  - Operador * - Conteúdo do endereço contido em
    - `v1 = *pInt1; //v1 recebe conteúdo do endereço contido em pInt1`
  - Operador & - Endereço desta variável
    - `pInt1 = &v1; //pInt1 recebe "endereço" de v1`
- Uso – Para uma variável do tipo referencia
  - Os operadores * e & não são utilizados

Ponteiros como Parâmetros

- O uso de ponteiros como parâmetros é muito eficiente quando o objeto a ser passado representa uma grande quantidade de memória
- Outro interessante uso do ponteiro ocorre quanto é necessário a modificação do objeto dentro do método.
- Quando um ponteiro é passado para o método, na realidade é passado apenas o endereço de um outro objeto.
- Sendo assim qualquer modificação dentro do método será refletida no objeto.

Referências como Parâmetros

- Um importante uso de referências é a passagem de parâmetros para métodos.
- O uso de referência permite que o método utilize a variável diretamente e não sua cópia, aumentando a eficiência e economizando memória.
- Sempre que um argumento é passado por referência sua modificação dentro do método será percebida ao sair do mesmo, visto que não há uma cópia e sim uma manipulação direta do objeto.
- Para evitar que uma referência seja modificada dentro de um método pode ser utilizada a palavra reservada **`const`** dessa forma mantém-se os benefícios e evita-se a modificação da variável.
  - Neste caso não é permitida a alteração da variável dentro do método, qualquer tentativa será rejeitada pelo compilador

Passagem de Parâmetros Resumo

- A seguir são mostrados exemplos de protótipos de método que utilizam as várias formas de passagem de parâmetros
- Passagem por Valor
  - Outra cópia é feita; Menos eficiente
  - `void ClassName::metodo(int);`
- Passagem por Referência
  - O próprio objeto é passado e logo Alteração dentro do método é refletida fora do mesmo; Mais eficiente
  - `void ClassName::metod(int&);`
- Passagem por Referência Constante
  - O próprio objeto é passado; alteração dentro do método não é refletida fora do mesmo; Mais eficiente
  - `void ClassName::metod(const int&);`
- Passagem por Ponteiro
  - O endereço do objeto é passado; alterações afetam o objeto original
  - `void ClassName::metod(int*);`

Passagem de Parâmetros Exemplo – Definição Métodos

```
//Passagem por valor
void ClassName::metodoV(int i){
  i = 10;
}
//Passagem por referência
void ClassName::metodoR(int& i){
  i = 20;
}
//Passagem por referência constante
void ClassName::metodoRC(const int& i){
  //i = 30; compilador não aceita modificação!
}
//Passagem por ponteiro
void ClassName::metodoP(int* i){
  *i = 40;
}
```

Passagem de Parâmetros Exemplo – Uso Métodos

```
int i(0);
int &iRef = i;
const int &iConstRef = i;
int* pI = &i;
ClassName p;
p.metodoV(i);
cout << "i: " << i << endl; //imprime 0
//Passagem por referência
p.metodoR(iRef);
cout << "i: " << iRef << endl; //imprime 20
//Passagem por referência constante
p.metodoRC(iConstRef );
cout << "i: " << iConstRef << endl; //imprime 20
//Passagem por ponteiro
p.metodoP(pI);
cout << "pI: " << pI <<endl; //imprime o endereço de i
cout << "*pI: " << *pI << endl; //imprime 40
```

Aplicação
Vetor Dinâmico

- O uso da alocação de memória é de utilidade para criar um vetor
- Vantagem do vetor estático que é o acesso através de um índice
  - Tempo constante para acesso (consulta ou alteração)
- Vantagem do número variável de elementos
  - A partir da alocação dinâmica da memória em tempo de execução
  - Tamanho do vetor pode aumentar ou diminuir durante seu uso
- Porém o mesmo possuirá a característica de um vetor onde a área de memória alocada é contigua na memória
- Como representar este vetor?

Vetor Dinâmico
Operação resize

- Aumenta, ou diminui, o número de elementos no vetor

	ENDEREÇO	CONTEÚDO
vetor[0]	0x22ff10	7
vetor[1]	0x22ff14	10
vetor[2]	0x22ff18	4
vetor[3]	0x22ff1c	1
	0x22ff20	12
pArray	0x32ff10	0x22ff10

MEMÓRIA
(Antes da Operação **resize**)

	ENDEREÇO	CONTEÚDO
vetor[0]	0x22ff10	7
vetor[1]	0x22ff14	10
vetor[2]	0x22ff18	4
vetor[3]	0x22ff1c	1
vetor[4]	0x22ff20	12
vetorNew[0]	0x22ff2c	7
vetorNew[1]	0x22ff30	10
vetorNew[2]	0x22ff34	2342234234
pArray	0x32ff10	0x22f10

MEMÓRIA
(Durante a Operação **resize**)

	ENDEREÇO	CONTEÚDO
vetorNew[0]	0x22ff2c	7
vetorNew[1]	0x22ff30	10
vetorNew[2]	0x22ff34	4
pArray	0x32ff10	0x22f2c

MEMÓRIA
(Após a Operação **resize**)

Vetor Dinâmico
Operação resize

- Antes
  - Vetor está alocado em área contigua da memória com um numero **n** de elementos
- Durante
  - Vetor está alocado em área contigua da memória com um número **n** de elementos
  - Nova área de memória (vetorNew) é alocada com **m** elementos, sendo que **m** pode ser maior ou menor que **n**
- Após
  - Área inicial de memória foi liberada
  - Vetor agora está alocado em área contigua (vetorNew) da memória com um numero **m** de elementos

Vetor Dinâmico
Representação Memória

- Representação na memória

	ENDEREÇO	CONTEÚDO
vetorNew[0]	0x22ff2c	7
vetorNew[1]	0x22ff30	10
vetorNew[2]	0x22ff34	1
pArray	0x32ff10	0x22f2c

maxSize

Vetor Dinâmico Representação C++

- Classe
  - DArray
- Possuirá os seguintes atributos:
  - Ponteiro para primeira posição
    - T* pArray
  - Número máximo de elementos no vetor
    - int maxSize

```
1. typedef int T;
2. class DArray {
3.   private:
4.       T* pArray;
5.       int maxSize;
6.   public:
7.    //operações
8. };
```

Vetor Dinâmico
Operações

- Construir um novo vetor (int vetor[10])
  - Vazio
  - Com n elementos
  - A partir de outro vetor
- Recuperar elemento na i-ésima posição
  - a = vetor[i]
- Alterar um elemento na i-ésima posição
  - vetor[i] = a
- Resize
  - Alterar o tamanho do vetor em tempo de execução
  - int resize(int n)

Vetor Dinâmico - Operações Construção

- Construtor padrão
  - Cria um vetor dinâmico vazio, ou seja, maxSize = 0

```
DArray();
```

- Construtor que aloca previamente vetor com n posições
  - Inicializa o valor de maxSize e aloca a memória necessária

```
DArray(int n);
```

- Construtor que cria um novo vetor dinâmico a partir de outro
  - Vetor dinâmico construído será uma cópia do objeto recebido

```
DArray(DArray&);
```

- Destrutor, responsável por liberar a memória

```
~DArray();
```

Vetor Dinâmico - Operações Acesso Informação

- Operações que permitem acessar os dados contidos em um vetor dinâmico
- Obtém a informação que está na i-ésima (iPos) posição do vetor
  - A informação é devolvida na variável data
  - retorno indica sucesso(0) ou falha(1)

```
int getData(int iPos, T& data);
```

- Obtém o tamanho máximo da memória alocada para o vetor

```
int getSize();
```

- Obtém um ponteiro para o endereço da primeira posição do vetor

```
T* getArray();
```

Vetor Dinâmico – Operações Modificação da Informação

- Operações que permitem modificar os dados contidos em um vetor dinâmico
- Altera a informação que está na i-ésima (iPos) posição do vetor
  - Retorno indica sucesso(0) ou falha(1)
  - O valor alterado no vetor (data) é passado por referência

```
int setData(int iPos, const T& data);
```

- Altera o tamanho máximo de um vetor, sendo possível aumentar ou diminuir o vetor em tempo de execução
  - Retorno indica sucesso(0) ou falha(1)

```
int resize(int iNewSize);
```

Vetor Dinâmico – Operações Auxiliares

- Permite imprimir o conteúdo do vetor dinâmico

```
void print();
```

- Permite copiar o conteúdo de um outro vetor. Ao final ambos serão iguais
  - Retorno indica sucesso(0) ou falha(1)

```
int copy(const DArray& darray);
```

Vetor Dinâmico
Definição da Classe (Darray.h)

```
1.  typedef int T;
2.  class DArray {
3.    private:
4.       T* pArray;
5.       int maxSize;
6.    public:
7.      DArray();
8.      DArray(int n);
9.      DArray(DArray&);
10.     ~DArray();
11.     int getSize();
12.     T* getArray();
13.     int getData(int iPos, T& data);
14.     int resize(int iNewSize);
15.     int setData(int iPos, const T& data);
16.     void print();
17.     int copy(DArray& darray);
18. };
```

Templates

- Durante a definição de uma estrutura de dados é necessário definir tipos de dados para os elementos
- No vetor dinâmico é necessário armazenar o endereço da primeira posição alocada (pArray)
- A definição deste atributo poderia ser realizada da seguinte forma:
  - `int*pArray;`
    - Vetor conterá somente endereços de números inteiros
  - `double*pArray;`
    - Vetor conterá somente endereços de números de precisão dupla
- A fim de deixar a definição genérica foi utilizado um recurso da linguagem C (Typedef)
  - `typedef int T;`
  - `T*pArray;`
    - Vetor conterá somente endereços de números inteiros, porém a alteração em um único ponto do código (a definição do tipo T) pode altera o vetor para outro tipo
- Apesar de ser flexível a estratégia acima possui uma deficiência
- Como em um mesmo código possui um vetor dinâmico de inteiros e outro de elementos do tipo float, por exemplo?

Templates

```
#include "DArray.h"
int main(int argc, char *argv[]){
  DArray vetorInteiros(3);
  DArray vetorFloat(4);
  ...
}
```

- Como o typedef é feito no arquivo “Darray.h” os objetos acima representam um vetor dinâmico que conterá ou números inteiros (float) ou números de ponto flutuante (float)
- O typedef não é a solução mais indicada
- A linguagem C++ oferece o recurso de templates

Templates
Exemplo de Uso – Visão Geral

```
#include "Circle.h"
int main(int argc, char *argv[]){
  DArray<int> vetorInteiros(3);
  DArray<float> vetorFloat(4);
  DArray<Circle> vetorCircle(4);
  ...
}
```

Vetor Dinâmico
Definição da Classe (DArray.h)

```
1.  //typedef int T;
2.  template <class T> class DArray {
3.    private:
4.       T* pArray;
5.       int maxSize;
6.    public:
7.      DArray();
8.      DArray(int n);
9.      DArray(DArray&);
10.     ~DArray();
11.     int getSize();
12.     T* getArray();
13.     int getData(int iPos, T& data);
14.     int resize(int iNewSize);
15.     int setData(int iPos, const T& data);
16.     void print();
17.     int copy(DArray& darray);
18. };
```

Template

Modificação Implementação (DArray.cpp)

```
1.  #include "DArray.h"

2.  //construtor padrao
3.  template <class T> DArray<T>::DArray(){
4.    pArray = NULL;
5.    maxSize = 0;
6.  }
```

Templates
Uso Classe – Modificação

```
#include "DArray.cpp"
#include "Ponto.h"
int main(int argc, char *argv[]){
  DArray<int> vetorInteiros(3);
  DArray<float> vetorFloat(4);
  DArray<Ponto> vetorPontos(4);
  ...
}
```

Templates
Restrições dos Operadores

- O Código abaixo, utilizando template, apresenta problemas:

```
1.  //imprime o conteudo do vetor
2.  template <class T> void DArray<T>::print(){
3.      T* pArrayTemp = pArray;
4.      for(int ii=0;ii<maxSize;ii++){
5.          cout << "DArray[" << ii << "]:" << *pArrayTemp
    << endl;
6.          pArrayTemp++;
7.      }
8.  }
```

- A operação << pode não estar definida para a classe <T>

Templates
Restrições dos Operadores

- Considere o código abaixo:

```
1.  int a(3), b(7)
2.  if (a == b){
3.    cout << "a: " << a << endl;
4.    return true;
5.  }
6.  else {
7.    cout << "b: " << b << endl;
8.    return false;
9.  }
```

- O mesmo funciona corretamente. A comparação (==) entre dois inteiros pode ser calculada, assim como a impressão (>>) do mesmo
- Caso a primeira linha de código fosse alterada, conforme abaixo

```
1.  Ponto a(3), b(7)
```

- O código não funcionará! Como comparar dois objetos da classe Ponto? Como deve ser a impressão de um objeto da classe Ponto?
- É necessário que este comportamento seja construído em uma classe

Templates
Restrições dos Operadores - Resolução

- Para resolver a restrição dos operadores a linguagem C++ permite a definição de operadores dentro de uma classe
- Desta forma é possível construir (codificar) o comportamento desejado para um referido operador sobre os objetos daquela classe
- Este recurso é conhecido na linguagem C++ como sobrecarga de operadores
- Vamos supor a classe Ponto definida da seguinte forma:

```
1. class Ponto {
2.   // As variáveis membro são privadas (encapsulamento)
3.   private:
4.     double dX, dY;
5.   // Os métodos sao públicos
6.   public:
7.     ...
8. };
```

- A modificação consiste em alterar a definição da classe ponto (Ponto.h) e a implementação (Ponto.cpp) onde será adicionado o comportamento dos novos operadores

Templates
Sobrecarga Operadores - Definição

- No arquivo que contém a definição da classe serão definidos os novos operadores

```
1.  class Ponto {
2.    // As variáveis membro são privadas (encapsulamento)
3.    private:
4.      double dX, dY;
5.    // Os métodos sao públicos
6.    public:
7.      ...
8.      //OPERADORES
9.      bool operator == (Ponto c) const;
10. };
```

- Neste caso o arquivo contém a indicação de que o operador de igualdade (==) será definido para a classe
- Utiliza-se a palavra reservada “operator” na definição

Templates

Sobrecarga Operadores - Comportamento

- No arquivo que contém a implementação dos métodos da classe (Ponto.cpp) o comportamento do operador será definido

```
1. // Operador relacional "igualdade"
2. bool Ponto::operator== (Ponto p) const{
3.   if ( (dX == p.getX()) && (dY == p.getY()) )
4.     return true;
5.   else
6.     return false;
7. }
```

- A partir de agora é possível comprar dois objetos da classe Ponto
- O método acima descreve o comportamento do operador. O objeto Ponto recebido (p) é comparado com o objeto que está a esquerda do operador, ou seja o próprio objeto que é submetido à operação

Templates
Sobrecarga – Operadores << e >>

- No caso dos operadores “<<” e “>>” existem algumas particularidades

```
1.  #include <iostream>
2.  using namespace std;
3.  class Ponto{
4.    // As variáveis membro são privadas (encapsulamento)
5.    private: double dX, dY;
6.    // Os métodos sao públicos
7.    public:
8.     //OPERADORES
9.     bool operator== (Ponto) const;
10.    friend istream& operator>> (istream& cin, Ponto&);
11.    friend ostream& operator << (ostream& cout, const Ponto&);
12. };
```

- Estes operadores são definidos em outras classes (istream e ostream) e o comportamento será adicionado na classe Ponto
- A palavra reservada **friend** indica que há uma relação de confiança entre o método acima e a classe Ponto
- Assim será possível que estas classes realizem um acesso direto às variáveis privadas

Templates
Sobrecarga – Operador <<

```
1.  //Sobrecarga do operador de inserção -  insere os dados na stream
    de saída cout
2.  ostream& operator << (ostream& cout, const Ponto& p){
3.   //Como friend, o operador "<<" acessa variáveis privadas da classe
4.   //O ponto será impresso no formato (dX , dY)
5.    cout << "(" << p.dX << " , " << p.dY << ")";
6.    return cout;
7.  }
```

- Desta forma a linguagem conhece agora a forma que a extração dos dados da classe Ponto (<<) é realizada sendo possível “imprimir” o conteúdo de um objeto da classe Ponto
- Deve ser notado que “operator << “ não é uma operação da classe, mas uma função externa que como possui relação de confiança (amizade) com a classe acessa seus dados protegidos diretamente
- Desta forma esta operação poderia estar em qualquer parte do código

Templates
Sobrecarga – Operador >>

```
1. //Sobrecarga do operador de extração - extrai dados da stream
2. istream& operator >> (istream& istr, Ponto& p){
3.  //Como friend, o operador ">>" acessa variáveis privadas da classe
4.  //O Ponto deverá ser digitado no formato:(dX, dY) obrigatoriamente
5.  char parenteseEsq, parenteseDir, virgula1;
6.  //Caracteres acima serão lidos e  desprezados.
7.   istr >> parenteseEsq >> p.dX >> virgula1 >> p.dY >> parenteseDir;
8.   return istr;
9. }
```

- Desta forma a linguagem conhece agora a forma que a extração dos dados da stream de entrada (>>) é realizada sendo possível “ler” o conteúdo de um objeto da classe Ponto
- Deve ser notado que “operator >> “ não é uma operação da classe, mas uma função externa que como possui relação de confiança (amizade) com a classe acessa seus dados protegidos diretamente
- Desta forma esta operação poderia estar em qualquer parte do código