Estruturas de Informação

Aspectos Essenciais C++

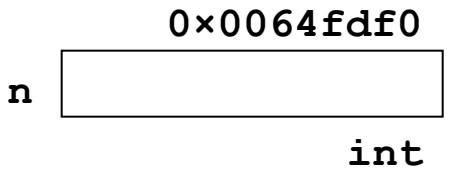
Departamento de Engenharia Informática (DEI/ISEP)
Fátima Rodrigues
mfc@isep.ipp.pt

Referências

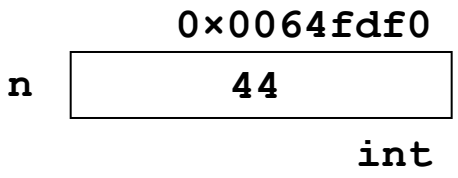
Operador de Referência &

- Uma declaração de variável associa três atributos fundamentais à variável: nome, tipo, endereço de variável

```
int n ;
```




```
int n = 44 ;
```



- Em C++ é possível obter o endereço de uma variável utilizando o operador referência &

```
void main()
{   int n = 44 ;
    cout << "n = " << n << endl ;     //imprime valor de n
    cout << "&n = " << &n << endl ;  //imprime endereço de n
}
```

```
> n = 44
> &n = 0×0064fdf0
```

Referências

- Uma referência é um nome alternativo (*alias*) para outra variável

```
Tipo& nome-ref = nome-var
```

```
void main ()
{
   int n = 44 ;
   int& rn = n ;   // rn é sinónimo de n

   cout << "n = " << n << ", rn = " << rn << endl ;
   n-- ;
   cout << "n = " << n << ", rn = " << rn << endl ;
   rn *= 2 ;
   cout << "n = " << n << ", rn = " << rn << endl ;
}
```

```
> n = 44, rn = 44
> n = 43, rn = 43
> n = 86, rn = 86
```

Os dois identificadores n e rn são nomes diferentes para a mesma variável

```
int& rn          //Erro: Inicialização obrigatória !
int& rn = 23 ;   //Erro: 23 não é uma variável

int& rn = n ;    //OK: uma referência deve ser inicializada com uma variável
```

Passagem de Parâmetros

Passagem de Parâmetros por Valor

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

void main()
{
    int a = 1, b = 2;
    troca(a, b);
    cout << "a : " << a << “, b : " << b << endl ;
}
```

```
> a: 1, b : 2
```

- Sempre que há uma **passagem de parâmetros por valor** é **executado o construtor cópia** do tipo de dados do parâmetro formal
- Quando a função termina destrutores dos tipos de dados dos parâmetros formais "destroem" os valores desses parâmetros

Passagem de Parâmetros por Referência

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

void main()
{
    int a = 1, b = 2;
    troca(a, b);
    cout << "a : " << a << " b : " << b << endl ;
}
```

```
> a: 2, b : 1
```

- Os parâmetros actuais são ligados aos parâmetros formais, **não existe cópia dos parâmetros**. O nome dos parâmetros formais substitui o nome dos parâmetros actuais. Poupa-se tempo e espaço !

- Qualquer alteração do parâmetro formal, altera o parâmetro actual. Deve-se usar quando se pretende que a função chamada **altere os valores dos argumentos**

Passagem de Parâmetros por Referência

Constante

```
void troca (const int& x, const int& y)
{
    int temp = x ;
    x = y ;         // ERRO: obj. const. não admite atribuições
    y = temp ;      // ERRO: idem
}
```

O parâmetro formal substitui o actual, mas como é constante, não admite alteração de valor

Caso **não se pretenda alterar** o valor do parâmetro actual usa-se:

- **Passagem por valor** no caso de tipos de dados primitivos
- **Passagem por referência constante** para tipos de dados não primitivos, tipos de dados que ocupem grande quantidade de espaço de armazenamento, pois a passagem por referência impede que o argumento seja duplicado

Valores de Retorno

Valores de Retorno

Objetos podem ser devolvidos:

- por valor
- por referência constante
- e ocasionalmente por referência

Quando é devolvido um **valor** (objeto), aquilo que é devolvido é copiado no ambiente de retorno, recorrendo-se ao **construtor cópia**

Por questões de eficiência, no caso da devolução de **tipos não primitivos**, deve-se usar o retorno por **referência constante**, para se evitar a invocação do construtor cópia

Um tipo de retorno de uma função pode ser uma referência desde que o valor devolvido seja um *lvalue* **não local à função**

Apontadores

Variáveis Dinâmicas

- Algumas Linguagens de Programação permitem ao programador explicitamente criar/destruir variáveis durante a execução do programa

- A área de armazenamento destas variáveis - **variáveis dinâmicas** - é a **Heap**

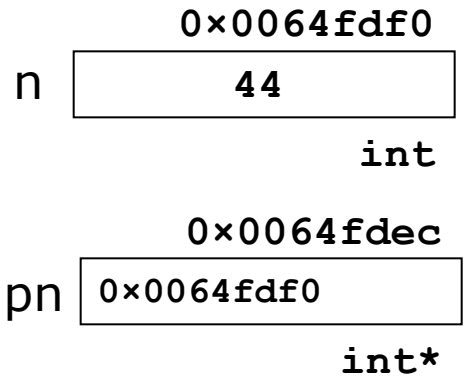
- Ao contrário das variáveis normais, as variáveis dinâmicas não podem ser acedidas directamente, de facto uma variável dinâmica não tem sequer um identificador

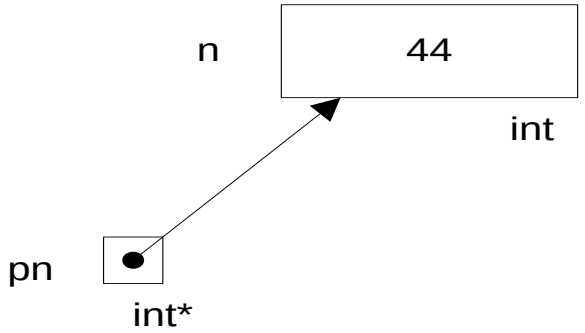
- Uma variável dinâmica é sempre acedida através de uma **variável apontador**

Apontadores e Endereços

- Um apontador é uma variável que contém o endereço de um objecto em memória (variável, elemento de array, etc.)
  Ex:

```
int*  pn = &n  ; //pn é um apontador que contém o endereço de um inteiro
```






```
void main()
{   int n = 44 ;
    cout << "n = " << n << "&n = " << &n << endl
    int* pn = &n
    cout <<  "pn = " << pn << endl ;
    cout <<  "&pn = " << &pn << endl ;
}
```

```
> n = 44   &n = 0×0064fdf0
> pn = 0×0064fdf0
> &pn = 0×0064fdec
```

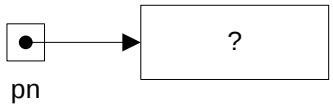
Operador new

```
int* pn ;
```

Declara um apontador para inteiro, ou seja, atribui um endereço a pn, mas a memória nesse endereço ainda não está alocada, logo pn não aponta para nenhuma memória alocada e portanto não pode ser acedido

Para criar uma variável dinâmica apontada por pn é necessário usar o operador **new**

```
pn = new int ;
```



Uma variável dinâmica do tipo int é criada, e o seu endereço é colocado no apontador pn

Atenção: `int* x, y, z;  // mesmo que: int* x; int y, z ;`

Alocação e libertação dinâmica de memória

- **new *tipo*** - **cria um objecto** do tipo indicado e retorna o seu endereço, ou 0 em caso de erro
  - objecto fica alojado numa zona de memória dinâmica chamada "heap"
  - pode-se inicializar objecto com: **new *tipo* (*valor-inicial*)**
  - endereço retornado deve ser atribuído a apontador do tipo: ***tipo****
  - destruir objecto (libertando memória) com: **delete *apontador* ;**

- **new *tipo* [*tamanho*]** - **cria um array de objectos** do tipo e tamanho indicados e retorna o endereço do 1º objecto do array
  - endereço retornado deve ser atribuído a apontador do tipo: ***tipo****
  - destruir array (libertando memória) com: **delete [ ] *apontador* ;**
  - Útil para criar arrays de tamanho conhecido apenas durante a execução do programa !
  - Quando as posições de memória inicialmente alocadas a um array são esgotadas é necessário re-alocar novo bloco de memória

Operador de Desreferência *

- Se **pn** apontar para **n** podemos aceder o valor de n através de **pn**
- A expressão ***pn** será avaliada como o valor de n, ou seja, o **conteúdo do apontado por pn**
- Aceder ou modificar a variável dinâmica criada é feito através do **apontador pn** usando o **operador ***

```
int* pn = new int ;
*pn = 7 ;          // coloca 7 na variável dinâmica
cin >> *pn ;       // lê do teclado o valor a armazenar na variável dinâmica
if (*pn == 0)      // testa se a var. dinâmica contém 0
```

```
void main()
{  int n = 44 ;
   cout << "n = " << n << "&n = " << &n << endl
   int* pn = &n
   cout <<  "pn = " << pn << endl ;
   cout <<  "&pn = " << &pn << endl ;
   cout <<  "*pn = " << *pn << endl ;
}
```

```
> n = 44   &n = 0×0064fdf0
> pn = 0×0064fdf0
> &pn = 0×0064fdec
> *pn = 44
```

Apontadores com zero e por inicializar

- A única constante que faz sentido atribuir a um apontador é **0** (zero), para significar que não está a apontar para nada (porque é garantido que 0 não é endereço válido para dados)
  - Podem-se comparar apontadores com a constante **0**

```
double* p = new double ;
if (p == 0) abort() ;
else *p = 3.14159 ;
```

- Se um apontador ainda não foi inicializado, não se pode aceder ao objecto apontado (com operador desreferência *)

```
float *p ;        // p é um apontador para float, não inicializado !
*p = 1.0 ;        // Erro: p não foi alocado

float x;
float* p = &x;    // OK: inicializa p com endereço de x
float* q;
q = new float;    // OK: aloca memória para 1 float
*q = 3.14159 ;    // OK: q foi alocado
```

Exemplo de manipulação de apontadores

*se **p** aponta para **x**, ***p** pode ocorrer em qualquer contexto em que **x** poderia ocorrer*

```
int x = 1, y = 2;

int* p ;

int* q ;          // p e q são apontadores para int

p = &x;           // coloca em p o endereço de x
                  // p agora aponta para x

q = p;            // copia para q o conteúdo de p
                  // q agora também aponta para x

y = *p;           // coloca em y o valor do objecto apontado por p
                  // y agora vale 1

*p = -1;          // coloca no objecto apontado por p o valor -1
                  // x agora vale -1

(*q)++;           // incrementa o valor do objecto apontado por q
                  // x agora vale 0
```

Valores válidos para um Apontador

Um apontador pode obter um valor válido por:

1. Cópia de outro apontador
   - Por atribuição
   - Por passagem de parâmetro
2. Atribuição do endereço de um objecto
   - Aplicação do operador unário & a um objecto
   - Valor retornado pelo operador new

```
int var1 = 8 ;
int var2 = 15 ;
int* ap1 ;
int* ap2 ;

ap1 = &var1 ;
ap2 = &var2 ;

cout << *ap1 << " " << *ap2 << endl ;
ap1 = ap2 ;
cout << *ap1 << " " << *ap2 << endl ;
cout << var1 << " " << var2 << endl ;
```

```
> 8      15
> 15     15
> 8      15
```

ap1 alterado !

```
int var1 = 8 ;
int var2 = 15 ;
int* ap1 ;
int* ap2 ;

ap1 = &var1 ;
ap2 = &var2 ;

cout << *ap1 << " " << *ap2 << endl ;
*ap1 = *ap2 ;
cout << *ap1 << " " << *ap2 << endl ;
cout << var1 << " " << var2 << endl ;
```

```
> 8      15
> 15     15
> 15     15
```

var1 alterada !

Apontadores para Objectos

```
class Ponto
{
  public:
         double x;
         double y;
       …
}

Ponto*  p = new Ponto ;

(*p).x = 22 ;     // equivalente a p->x = 22 ;
```

Como p é um apontador para um objecto Ponto, *p é um objecto Ponto

(*p).x acede ao seu membro x

São necessários parêntesis na expressão (*p).x pois o operador directo de seleção de membro "." tem precedência sobre o operador desreferência "*****"

As duas notações (*p).x e p->x são equivalentes

Classes: Pessoa / Aluno

```
class Pessoa {
    private:
       string nome ;
       Data dtnasc ;
       int alt ;

    public:
       Pessoa();
       Pessoa(string n, const Data& d, int al);
       Pessoa(const Pessoa& p);
       virtual ~Pessoa();

       virtual Pessoa* clone() const ;

       string getNome() const;
       int getAlt() const;
       const Data& getDtnasc() const;

       void setDtnasc(const Data& dt);
       void setNome(string n);
       void setAlt(int a);

       int Idade() const ;

       virtual void listar() const;

};
```

```
class Aluno: public Pessoa {
  private:
     int numero;
     string curso;

  public:
    Aluno();
    Aluno(const Pessoa& p, int num,
                                string c);
    Aluno(const Aluno &a);
    ~Aluno () ;

    Pessoa* clone() const ;

    int getNum() const ;
    void setNum(int n) ;

    string getCurso() const ;
    void setCurso(string c) ;

    void listar() const;
};
```

Exemplo com classe derivada

```
void main()
{
    Pessoa maria ("Maria",1980,1,15,165);
    cout << maria.nome << '\n';                    // ERRO, membro privado
    cout << maria.Idade() << '\n';                 // OK, método público

    Aluno jose ("Jose",Data(1955,5,20),190,1245);
    cout << jose.getNome() << '\n';                // OK, método público herdado

    Pessoa& refpess = jose  ;                      // OK, Pessoa é superclasse
    refpess.listar();                              // OK, imprime nome,dt-nasc,alt,num

    Aluno& refaluno = maria;                       // ERRO

    Pessoa* ptrpess = &jose;                       // OK, Pessoa é superclasse
    ptrpess->listar();                             // OK, imprime nome,dt-nasc,alt,num

    Aluno* ptraluno = &maria;                      // ERRO

    Pessoa p1 = maria ;                            //OK, invoc. explícita do construtor cópia

    Pessoa p2 = jose ;                             // OK, Pessoa é superclasse
    p2.listar();                                   // OK, imprime só  nome,dt-nasc,altura

    Aluno aluno = maria;                           // ERRO, não converte
}
```

Sobrecarga de Operadores

Sobrecarga de Operadores

- Em C++ é possível reescrever código para operadores de modo a manipularem operandos de classe diferente da que estão predefinidos

- Esta característica confere uma maior legibilidade aos programas, particularmente quando se trata de expressões aritméticas envolvendo tipos definidos pelo programador

- Métodos ou funções globais declaradas com a palavra chave *operator* associado ao símbolo do operador:

  operator @ ( )

  promove a sobrecarga desse operador, sendo **@** qualquer dos símbolos susceptíveis de sobrecarga

Operadores que podem ser redefinidos

- Quase todos os operadores podem ser redefinidos:
  - aritméticos: + - *(unário ou binário)* * / %
  - bit-a-bit: ^ & | ~ << >>
  - lógicos: ! && ||
  - de comparação: == != > < <= >=
  - de incremento e decremento: ++ -- *(pósfixos ou préfixos)*
  - de atribuição: = += -= *= /= %= |= &= ^= ~= <<= >>=
  - de alocação e libertação de memória: new new[] delete delete[]
  - de sequenciação: ,
  - de acesso a elemento de array: [ ]
  - de acesso a membro de objecto apontado: →
  - de chamada de função: ( )

Só podem ser definidos como métodos

- **Operadores que não podem ser redefinidos**:
  - de resolução de âmbito: ::
  - de acesso a membro de objecto: . .* →*

Funções-operador

Uma função com nome **operator** seguido do símbolo de um operador define o significado desse operador para objectos de uma classe (ou combinação de classes)

```
class Data {
   private:
     int ano, mês, dia ;

   public:
….
     const Data& operator++();
     const Data& operator++(int);
     const Data& operator = (const Data& d);
     bool operator > (const Data& d) const;
     bool operator == (const Data& d) const;

     void escreve(ostream& ostr) const;
     void leitura(istream & in) ;
};
```

Operadores unários

- Um operador unário prefixo (- ! ++ --) pode ser definido por:

  1) um membro-função não estático sem argumentos, ou

  2) uma função não membro com um argumento

- Para qualquer **operador unário prefixo** @, @x pode ser interpretado como:

  1) x.operator@(), ou

  2) operator@(x)

- Os **operadores unários posfixos** (++ --) são definidos com um **argumento adicional do tipo int** que nunca é usado (serve apenas para distinguir do caso prefixo):

  Para qualquer operador unário posfixo @, x@ pode ser interpretado como:

  1) x.operator@(int), ou

  2) operator@(x,int)

Exemplo com operador unário

```
const Data& Data::operator++() {
   if (dia < diasPorMes[mes])
     dia++;
   else if (mes==2 && dia == 28 && anoBissexto(ano))
      dia = 29 ;
   else {
      dia = 1;
      ano = (mes == 12? (ano+1) : ano);
      mes = (mes == 12? 1 : mes+1); }

   return *this;
}

const Data& Data::operator++(int) {
   //Data aux = *this ;
   Data aux(*this) ;
   operator++() ;
   return aux ;
}
```

Exemplo com operador unário

```
int main()
{
   Data d1 (2003,12,31);
   d1.listar() ;
   ++d1 ;
   d1.listar() ;
   Data d2 = d1++ ;
   cout << " d1 " ; d1.listar() ;
   cout << " d2 " ; d2.listar() ;
   return 0 ;
}
```

```
2003/12/31
2004/1/1
2004/1/2
2004/1/1
```

Operadores binários

- Um operador binário pode ser definido como:
  1) um **método da classe**, ou
  2) uma **função global**

- Para qualquer operador binário @, x@y pode ser interpretado como:
  1) x.operator@(y), ou
  2) operator@(x,y)

- Os operadores = [] () e -> só podem ser definidos da 1ª forma (através membro-função não estático), para garantir que do lado esquerdo está um *lvalue*

Operadores binários (método da classe)

```
class Complex {
  private:
          double re;
          double im;

  public:
          ...
          const Complex& operator + (const Complex& c) ;
} ;

const Complex& Complex::operator + (const Complex& c)  {
    re += c.re ; im += c.im ;

    return *this ; }

void main ()
{
  Complex c1(4,-6), c2(3,0), c3 ;

  c3 = c1 + c2 ;
  c3 = c1.operator+(c2) ;

  cout << c3 << endl << c1 << endl << c2 << endl ;

  c3 = 3 + c1 ;              // erro !
  c3 = Complex(3,0) + c1     // Ok ! Coerção explícita
  c3 = c1 + 3 ;              // erro !
}
```

```
7 - 6i
4 - 6i
3 + 0i
```

Operadores binários (função global)

```
class Complex {
  ...
  public:
  ...
  const Complex& operator += (const Complex& other) ;
  ...
} ;

const Complex& Complex::operator += (const Complex& other) {
   re += other.re ;
   im += other.im ;

   return *this ; }

// Operador definido como global
Complex operator + (const Complex& c1, const Complex& c2) {
   Complex aux(c1) ;
   aux += c2 ;

   return aux ;  }
```

Método ou Função Global

Operadores	Expressões	Método	Função Global
Unário prefixo	@ a	a.operator @ ( )	operator @ (a)
Unário sufixo	a @	a.operator @ (int)	operator @ (a, int)
Binário	a @ b	a.operator @ (b)	operator @ (a,b)
Afectação	a = b	a.operator = (b)	
Indexação	a [b]	a.operator [ ] (b)	
Desreferência	a →	a.operator→( )	
Chamada função	a (...)	a.operator( )(...)	

Entrada e saída de dados com *streams*

- **`cout << exp1 << exp2 << ...`**
  - **cout** é o *standard output stream* (normalmente conectado ao ecrã)
  - escreve (insere) no *stream* de saída os valores das expressões indicadas
  - "**<<**" está definido para tipos de dados *built-in* e pode ser definido para tipos de dados definidos pelo utilizador
- **cin >> *var1* >> *var2* >> ...**
  - **cin** é o *standard input stream* (normalmente conectado ao teclado)
  - lê (extrai) do *stream* de entrada valores para as variáveis da direita
  - ">>" está definido para tipos de dados *built-in* e pode ser definido para tipos de dados definidos pelo utilizador
  - salta caracteres "brancos" (espaço, *tab*, *newline, carriage return, vertical tab* e *formfeed*), que servem para separar os valores de entrada

Sobrecarga dos Operadores << e >>

```
class Data {
    ...
     void escreve(ostream& ostr) const;
     void leitura(istream& in)  ;
};

void Data::escreve(ostream& out) const {
  out << ano << "/" << mes << "/" << dia; }

ostream& operator << (ostream& out, const Data& d) {
     d.escreve(out) ;
     return out  ; }

void Data::leitura(istream& in) {
   char b ;
   in >> dia >> b >> mes >> b >> ano;  }

istream& operator >> (istream& in, Data& d) {
     d.leitura(in) ;
     return in;          }
```

Os operadores <<, >> são definidos como uma função que invoca o método escreve e leitura da respectiva classe

É uma solução standard, igual para todas as classes !

Funções e Classes *Template*

Funções Template

Existem algoritmos que são independentes do tipo, e por isso faz todo o sentido escrever código para estes algoritmos uma só vez, **independentemente do tipo** o para o qual serão instanciados

```
int minimo (int a, int b)
{ return (a < b ? a : b) ; }

float minimo (float a, float b)
{ return (a < b ? a : b) ; }
```

Funções Template

- Usadas para implementar uma única função que executa operações idênticas para diferentes tipos de dados
- Função genérica definida em função de parâmetros que representam o tipo de dados a operar
- O tipo de dados é especificado (instanciado) quando a função é chamada
- Precede-se o cabeçalho da função com a palavra-chave **template** seguida de uma lista de parâmetros entre **< >,** que representam o tipo de dados a instanciar mais tarde, e o seu nome.
  - cada parâmetro é precedido da palavra-chave **class**.

```
template <class T>
T minimo (const T& a, const T& b)
{ return (a < b ? a : b) ; }
```

Outro Exemplo de Função Template

```
template <class T>
void troca (T& a, T& b){
    T temp = a ;
    a = b ;
    b = temp ;  }

main() {
   int     a=3, b=10;
   double  x=3.99, y=5.72;
   Circulo c1(Ponto(10,75),34,"azul");
   Circulo c2(Ponto(5,88),99,"branco");

   troca(a,b);          // OK
   troca(x,y);          // OK
   troca(c1,c2);        // ?   }
```

Quando o compilador encontra a invocação da função **template** `troca()` gera o seu respetivo código. São criadas 3 versões, uma para manipular **int**, outra para **double** e outra para **Circulo**

A função troca() só executará instanciada com a classe Circulo se esta classe tiver definido a **sobrecarga do operador de atribuição** exigido pela função

Classes Template

Através de classes Template consegue-se implementar implicitamente, um comportamento comum para Classes garantindo-se que diferentes "objectos" suportam operações segundo a mesma sintaxe (mesmo nome, mesmos parâmetros)

Classes Template são também conhecidas como **classes genéricas** ou um gerador de classes - a classe template possibilita estabelecer um padrão para as definições de classes

Classes concretas são definidas independentemente

Classes Template

- Permite a generalização de classes. Classes similares que possuem diferentes tipos de dados para os mesmos membros, não necessitam ser definidas mais do que uma vez

- "Classe genérica" (também chamada classe parametrizada) definida em função de parâmetros a instanciar para se ter uma "classe ordinária"

- Precede-se a definição da "classe genérica" com:

  template <*class nome-de-parâmetro*, ... >

- Para se obter uma classe ordinária, tem que se instanciar os parâmetros, numa lista entre <> a seguir ao nome da classe genérica

Parâmetros Classes Template

Uma classe Template define então uma família de classes **parametrizáveis**, ou seja, que tomam como parâmetros:

- parâmetros tipo *(type template parameters*)
- parâmetros valor *(non type template parameters*)

Exemplo:

```
Template <typename T, class W, int X, char Y>
class Exemplo
{ … } ;
```

Declara um template de classes Exemplo:

- 2 parâmetros tipo (T e W)
- 2 parâmetros valor X e Y

Este template pode ser instanciado:

```
Exemplo <int, string, 3, ’a’>
Exemplo <Data, Pessoa, 75, ’p’>
```

Exemplo de Classe Template

```
template <class T>
class Grafica {
  private:
           int ind ;
           int cap ;
            T *vect ;
  public:
          Grafica();
          ~Grafica();

          void inserir (const T& elem) ;
          void escrever(ostream& out) const ;  };

template <class T>
Grafica<T>::Grafica {
   cap = dim ;
   vect = new T [cap] ;  //vector dinâmico classe template
   ind = 0 ;  }

template <class T>
Grafica<T>::~Grafica() {
   delete [] vect ; }
```

Exemplo de Classe Template

```
template <class T>
void Grafica<T>::inserir(const T& elem) {
  if (ind == cap) {     //vector cheio ! Dobrar espaço
    cap*=2;
    T *newvect = new T [cap] ;

    for (int i = 0 ; i < ind ; i++)
       newvect[i] = vect[i] ;

    delete [] vect;     //liberta a memória correspondente ao vector inicial

    vect = newvect ; }   //renomeia novo vector
    vect[ind] = elem ;
    ind++;
}

template <class T>
void Grafica<T>::escrever(ostream& out) const {
   for (int i=0; i< ind; i++)
      vect[i].desenhar() ;

   out << endl ;  }

template <class T>
ostream& operator << (ostream& out, const Grafica<T>& g) {
   g.escrever(out) ;
   return out ;      }
```

Instanciação Classe Template

```
void main() {
    Grafica<Circulo> grafCirc ;
    Circulo c1(Ponto(10,75),3);
    Circulo c2(Ponto(5,88),5);
    grafCirc.inserir(c1) ;
    grafCirc.inserir(c2) ;
    cout << grafCirc ;

   Grafica<Quadrado> grafQuadrad ;
   Quadrado q1(Ponto(1,1),4) ;
   Quadrado q2(Ponto(0,0),2) ;
   Quadrado q3(Ponto(3,3),5) ;
   grafQuadrad.inserir(q1) ;
   grafQuadrad.inserir(q2) ;
   grafQuadrad.inserir(q3) ;
   cout << grafQuadrad ;       }
```

```
Circulo Centro (10,75) Raio 3
Circulo Centro (5,88) Raio 5
Quadrado lado 4
Quadrado lado 2
Quadrado lado 5
```

- O tipo para qual será instanciada a classe template é conhecido em tempo de compilação: **polimorfismo estático**
- **Não podem ser construídas coleções heterogéneas de objetos**

Breve Introdução à Biblioteca STL

Standard Template Library - STL

É uma biblioteca que contém classes de *coleções* (estruturas de dados) e de *algoritmos de coleções*, organizados na forma de *templates*

A STL oferece os seguintes benefícios:

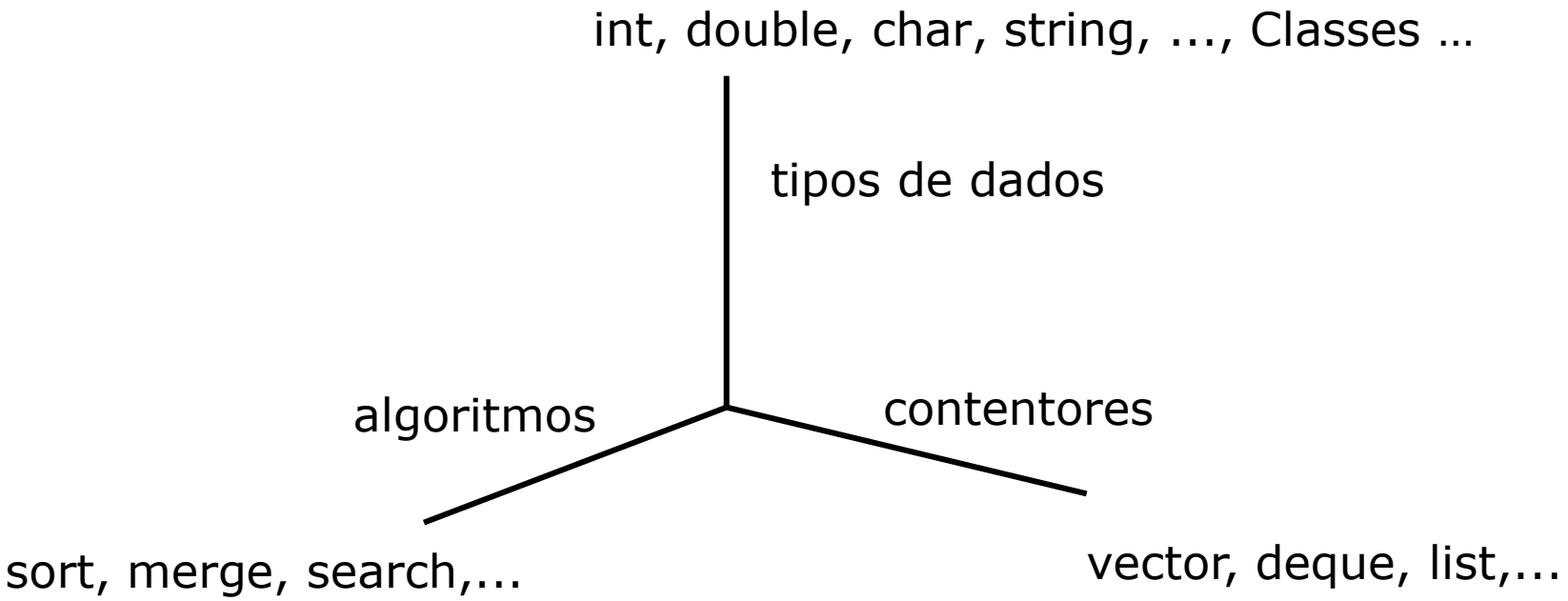
- **reutilização de código** – como é baseado em templates, as classes STL podem ser adaptadas a tipos distintos sem mudança de funcionalidade
- **portabilidade e facilidade de uso**

Filosofia subjacente à STL

O mesmo algoritmo pode ser aplicado a diferentes contentores de diferentes tipos de dados:

Algoritmo Sort $\rightarrow$ vector int,

Algoritmo Sort $\rightarrow$ list string, ...



Categorias de classes na STL

Três principais componentes da Biblioteca STL:

- **Classes *containers* ≈ Contentores**

  Usados para armazenar coleções de objetos

- ***Iterators* ≈ Iteradores**

  Permitem percorrer os elementos das coleções, independentemente do seu tipo, **estabelecem a ligação entre os algoritmos genéricos e os contentores**, cada contentor tem o seu próprio iterador que sabe como percorrê-lo

- **Algoritmos genéricos**

  Oferecem funcionalidades genéricas para manipular os dados dos contentores

Contentores

Correspondem às coleções de elementos de um determinado tipo, na forma de classes template

Os contentores definidos pela STL são:

- **vector**: elementos organizados na forma de um *array* que pode crescer dinamicamente
- **list**: elementos organizados na forma de uma lista duplamente ligada
- **deque**: elementos organizados em sequência, permitindo inserção ou remoção no início ou no fim sem necessidade de movimentação de outros elementos
- **map**: cada elemento é um par <*chave, elemento*> sendo que a *chave* é usada para ordenação da coleção
- **set**: coleção ordenada na qual os próprios elementos são utilizados como chaves para ordenação da coleção
- **multimap**
- **multiset**

Vector

- Um array é uma coleção de elementos todos do mesmo tipo diretamente acedidos através de um índice inteiro
- A classe vector é uma alternativa à estrutura de dados standard array
- Um vector é um array redimensionável
- Os vetores são mais poderosos do que os arrays devido ao número de funções disponíveis na classe e pelo facto de disponibilizarem gestão automática de memória

Exemplo Class Vector

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
   int elem;
   vector<int> v;

   cout << "Tamanho do contentor " << v.size() << endl;

   //inseção no fim do contentor
   for (int i = 0; i < 10; i++) {
      cout << "Próximo elemento: "; cin >> elem;
      v.push_back(elem); }

   cout << "Elementos do contentor em ordem inversa: " << endl;
   for (vector<int>::reverse_iterator itrv = v.rbegin(); itrv < v.rend(); itrv++){
      cout << *itrv << " " ; }
   cout << endl;
```

Exemplo Class Vector

```
    //Iteradores na direção original
    vector<int>::iterator inicio = v.begin();
    vector<int>::iterator fim = v.end();

    //Ordenação dos elementos do vector
    sort(inicio,fim);

    cout << "Elementos ordenados: " << endl;
    for (vector<int>::const_iterator it = v.begin(); it < v.end(); it++) {
        cout << *it << " ";
    }

    return 0;
}
```

Ficheiros

FICHEIRO

É uma coleção externa de dados que tem associada uma estrutura de dados designada por "stream"

- **Acesso direto** - ficheiros em disco
- **Acesso sequencial** - ficheiros em "*tape*"

A operação de leitura retira dados de um "stream" de entrada

A operação de escrita junta novos dados ao fim de um "stream" de saída

Os ficheiros são usados para guardar grandes quantidades de dados em disco

Manipulação de ficheiros com *streams*

```
#include <iostream>
#include <fstream>
using namespace std ;

void main() {
  ifstream origem ;
  origem.open("f1");   //define variável e abre ficheiro para leitura (ifstream)
  if (!origem)
  { cerr << "Erro a abrir ficheiro f1\n"; return -1; }

  ofstream  destino ;
  destino.open ("f2") ;  // idem, para escrita  (ofstream)
  if (!destino)
  { cerr << "Erro a abrir ficheiro f2\n"; return -1; }

  char c;
  origem >> c ;
  while (!origem.eof()){
     destino << c ;
     origem >> c;   }
  origem.close() ;
  destino.close() ;
}
```

Manipulação de ficheiros com *streams*

Se o ficheiro (f1) de texto tiver o seguinte conteúdo:

"Dennis Ritchie*LF*Brian Kernighan*LF*Bjarne Stroustrup*EOF*"

```
> DennisRitchieBrianKernighanBjarneStroustrup
```

Para ler linha a linha ter-se-á que usar a função get()

```
....
char c;
origem.get(c) ;
while ( !origem.eof() )
{
     destino << c ;
     origem.get(c) ;
}
....
```

```
Dennis Ritchie
Brian Kernighan
Bjarne Stroustrup
```

Manipulação de ficheiros com *streams*

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std ;

const int maxbook = 20 ;
int main() {
    struct Livro {
      string titulo ;
      string autor ;
      float preco ; } ;
  Livro book ;
  Livro bibliot[maxbook] ;
  int ind = 0, inic = 0;
  string linha ;

  ifstream fx ;              //define variav. e abre fx. para leitura
  fx.open("Livros.txt") ;
  if (!fx)
  { cout << "Fx. nao existe !" << endl ; return -1 ; }

  while (!fx.eof()) {
     getline(fx,linha,'\n');
     if (linha.size()  > 0) {
       int pos = linha.find(',', inic);
       string livro(linha.substr(inic,pos-inic));
       pos++ ;
```

Livros.txt

```
livro1,autor1,250.50
livro2,autor2,300
livro3,autor3,195.30
…
```

Manipulação de ficheiros com *streams*

```
      inic = pos ;
      pos = linha.find(',', inic);
      string autor(linha.substr(inic,pos-inic));
      pos++ ;

      inic = pos ;
      pos = linha.find(',', inic);  //pos = -1 já não encontra vírgula
      string preco_liv(linha.substr(inic,pos-inic));  //devolve a substring até ao fim
      char* aux = &preco_liv[0];
      float price  = atof(aux);
      pos++ ;

      book.titulo = livro ;
      book.autor = autor ;
      book.preco = price ;

      if (ind < maxbook)
         bibliot[ind++] = book ;
      }
   }
   fx.close() ;
   for (int i=0; i < ind; i++)
      cout << bibliot[i].titulo << " " << bibliot[i].autor << " " ;
      cout << bibliot[i].preco << endl ;

   return 0 ;
}
```