Apêndice C

Curiosidades e fenómenos estranhos

Apresentam-se aqui algumas curiosidades acerca da linguagem C++ e da programação em geral. Note-se que este apêndice contém código que vai do completamente inútil até ao interessante ou mesmo, imagine-se, potencialmente útil. Não o leia se não se sentir perfeitamente à vontade com a matéria que consta nos capítulos que a este se referem. A ordem das curiosidades é razoavelmente arbitrária.

C.1 Inicialização

As duas formas de inicialização do C++ diferem em alguns aspectos. Compare-se o programa

```
#include <iostream>

using namespace std;

int main()
{
    int i = 1;
    {
        int i = i;
        cout << i << endl;
    }
}
```

com este outro

```
#include <iostream>

using namespace std;

int main()
```

```
{
    int i = 1;
    {
        int i(i);
        cout << i << endl;
    }
}
```

O segundo escreve 1 no ecrã. O primeiro escreve lixo (só por azar, ou sorte, será 1). Porquê?

Porque no primeiro a variável mais interior já está definida (embora contendo lixo) quando se escreve `= i`, pelo que a variável é inicializada com o seu próprio valor (tinha lixo... com lixo fica). No segundo caso a variável mais interior não está ainda definida quando se escreve `(i)`, pois estes parênteses fazem parte da definição. Assim, a variável interior é inicializada com o valor da variável mais exterior...

C.1.1 Inicialização de membros

Uma consequência interessante do que se disse atrás é que também se aplica nas listas de construtores das classes. Por exemplo:

```
class A {
  public:
    A(int i, int j, int k)
        : i(k), // i membro inicializada com k parâmetro!
          j(j), // j membro inicializada com j parâmetro!
          k(i)  // k membro inicializada com i parâmetro!
    {
    }
  private:
    int i;
    int j;
    int k;
};
```

C.2 Rotinas locais

A linguagem C++ proíbe a definição de rotinas locais: todas as funções e procedimentos são globais. Esta afirmação é verdadeira, mas apenas para funções e procedimentos não membro. Uma vez que existe o conceito de classe local, podem-se usar métodos de classe em classes locais para simular rotinas locais. O método é simples:

```
// Define-se esta macro para tornar a sintaxe de definição mais simpática:
#define local_definitions struct _
```

```
// Define-se esta macro para tornar a sintaxe de invocação mais simpática:
#define local _::

int main()
{
    // As rotinas locais definem-se sempre dentro deste falso ambiente:
    local_definitions {
        // As rotinas locais são sempre métodos de classe (static):
        static int soma(int a, int b)
        {
            return a + b;
        }
    };

    // A invocação de rotinas locais faz-se através da falsa palavra-chave local:
    int n = local soma(10, 45);
}
```

C.3 Membros acessíveis "só para leitura"

A linguagem C++ proporciona três diferentes categorias de acesso para os membros de uma classe: há membros públicos, protegidos e privados. O valor de variável membro pública (a evitar porque viola o princípio do encapsulamento) pode ser usado ou alterado directamente sem quaisquer restrições. Uma variável membro privada não é acessível de todo do exterior. Não era simpático que existissem variáveis membro cujo valor estivesse disponível para ser usado directamente mas não pudesse ser alterado excepto pela própria classe? Note-se que uma constante membro pública não resolve o problema, pois nem a classe a pode alterar!

A solução passa por usar uma referência constante pública para aceder a uma variável membro privada! Por exemplo, se a variável for do tipo `int` e se chamar `valor`:

```
class A {
  public:
    // O construtor faz valor referenciar valor_:
    A(int v)
        : valor(valor_), valor_(v) {
    }

    // Quando se usa este truque é fundamental fornecer um construtor por cópia
    // apropriado. Porquê? Porque senão a referência da cópia refere-se à variável
    // do original!
    A(A const& a)
        : valor(valor_), valor_(a.valor_) {
    }
```

```
    // Da mesma forma tem de se fornecer o operador de atribuição por cópia. Caso
    // contrário não se poderiam fazer atribuições entre instâncias de classes com o
    // truque. É que o C++ não pode fornecer o operador de atribuição por cópia
    // automaticamente a classes com membros que sejam referências!
    A& operator = (A const& a) {
        valor_ = a.valor_;
        return *this;
    }

    // A referência valor para a variável valor_ é constante para evitar alterações
    // por entidades externas à classe:
    int const& valor;

    // Um procedimento para atribuir um novo valor à variável. Só para efeitos de tes-
te:
    void poeValor(int v) {
        valor_ = v;
    }

  private:
    // A variável propriamente dita é privada...
    int valor_;
};
```

Com esta classe é possível escrever o seguinte código:

```
#include <iostream>

using namespace std;

int main()
{
    A a(10);
    cout << a.valor << endl; // mostra 10.
    a.valor = 20;            // dá erro!
    a.poeValor(30);
    cout << a.valor << endl; // mostra 30.
    A b = a;
    cout << b.valor << endl; // mostra 30.
    A c(40);
    c = a;
    cout << c.valor << endl; // mostra 30.
}
```

Quem disse que não havia a categoria “só para leitura” em C++?

C.4 Variáveis virtuais

O princípio do encapsulamento obriga a esconder as variáveis membro de uma classe (fazê-las privadas) e fornecer funções de inspecção para o seu valor se necessário. Isto é fundamental por um conjunto de razões:

1. A implementação da classe pode precisar de mudar. As variáveis membro fazem naturalmente parte da implementação de uma classe. Uma variável membro pública faz também parte da interface da classe, pelo que uma alteração na implementação pode ter impactos sérios em código que use a classe.

2. Se o valor contido pela variável tiver de ser calculado de forma diferente em classes derivadas, só com funções de inspecção virtuais se pode especializar essa forma de cálculo.

Suponha-se, por exemplo, uma hierarquia de classes representando contas. A classe base da hierarquia representa uma abstracção de conta. Para simplificar considera-se que uma conta permite apenas saber o seu saldo (e possui um destrutor virtual, como todas as classes polimórficas):

```
class Conta {
  public:
    virtual ~Conta() {}
    virtual double saldo() const = 0;
};
```

Definem-se agora duas concretizações do conceito. A primeira é uma conta simples. Uma conta simples tem um saldo como atributo. Possui um construtor que inicializa o saldo e sobrepõe uma função de inspecção especializada que se limita a devolver esse saldo (ou seja, fornece um método para a operação abstracta `saldo()`). A segunda é um *portfolio* de contas, i.e., uma "conta de contas". O portfolio de contas tem um procedimento `acrescentaConta()` que permite acrescentar uma conta ao portfolio, um destrutor para destruir as suas contas e fornece também um método para a operação virtual `saldo()`:

```
#include <list>

class ContaSimples : public Conta {
  public:
    ContaSimples(double saldo)
        : saldo_(saldo) {
    }
    virtual double saldo() const {
        return saldo_;
    }
  private:
    double saldo_;
```

```
};

class Portfolio : public Conta {
  public:
    ~Portfolio() {
        // Destrói contas do portfolio:
        for(std::list<Conta*>::iterator i = contas.begin();
            i != contas.end(); ++i)
            delete *i;
    }
    void acrescentaConta(Conta* nova_conta) {
        contas.push_back(nova_conta);
    }
    virtual double saldo() const {
        // É a soma dos saldos das contas no portfolio:
        double saldo = 0.0;
        for(std::list<Conta*>::const_iterator i = contas.begin();
            i != contas.end(); ++i)
            saldo += (*i)->saldo();
        return saldo;
    }
  private:
    std::list<Conta*> contas;
};
```

Estas classes podem ser usadas como se segue:

```
#include <iostream>

using namespace std;

int main()
{
    // Constrói duas contas simples (ponteiros para Conta!):
    Conta* c1 = new ContaSimples(10);
    Conta* c2 = new ContaSimples(30);

    // Mostra saldos das contas simples:
    cout << c1->saldo() << endl; // mostra 10.

    cout << c2->saldo() << endl; // mostra 30.


    // Constrói uma conta portfolio e acrescenta-lhe as duas contas:
    Portfolio* p = new Portfolio;
    p->acrescentaConta(c1);
```

```
    p->acrescentaConta(c2);

    // Guarda endereço da conta portfolio no ponteiro c para Conta:
    Conta* c = p;

    // Mostra saldo da conta portfolio:
    cout << c->saldo() << endl; // mostra 40.
}
```

Note-se que o polimorfismo é fundamental para que a invocação da operação `saldo()` leve à execução do método `saldo()` apropriado à classe do objecto apontado e não à classe do ponteiro! Isto seria muito difícil de reproduzir se se tivesse colocado uma variável membro para guardar o saldo na classe base, mesmo que fosse privada. Porquê? Porque nesse caso a operação `saldo()` não seria abstracta, estando definida na classe base como devolvendo a valor dessa variável membro. Por isso o valor dessa variável teria de ser mantido coerente com o saldo da conta. Isso é fácil de garantir para uma conta simples mas muito mais difícil num portfolio, pois sempre que uma classe num portfolio tiver uma mudança de saldo terá de avisar a conta portfolio desse facto para que esta tenha a oportunidade de actualizar o valor da variável.

Se essa variável fosse pública ter-se-ia a desvantagem adicional de futuras alterações na implementação terem impacto na interface da classe, o que é indesejável.

Logo, não se pode usar uma variável membro pública neste caso para guardar o saldo.

Porquê este discurso todo no apêndice de curiosidades e fenómenos estranhos do C++ e numa secção chamada **Variáveis virtuais**? Porque... é possível ter variáveis membro públicas, sem nenhum dos problemas apontados, e ainda apenas com permissão para leitura (restrição fácil de eliminar e que fica como exercício para o leitor).

O truque passa por definir essa variável membro como sendo de uma classe extra `DoubleVirtual` (amiga da classe `Conta`) e que possui:

1. Um construtor que recebe a instância da conta.
2. Um operador de conversão implícita para `double` que invoca o operador virtual de cálculo do saldo.

Adicionalmente, pode-se colocar o operador de cálculo do saldo na parte privada das classes, uma vez que deixa de ser útil directamente para o consumidor das classes. Finalmente, para evitar conflitos de nomes, este operador passa a ter um nome diferente:

```
class Conta {
  public:
    class DoubleVirtual {
    public:
        DoubleVirtual(Conta& base)
        : base(base) {
```

```
        }
        operator double () const {
            return base.calculaSaldo();
        }
    private:
        // Guarda-se uma referência para a conta a que o saldo diz respeito:
        Conta& base;
    };
    friend DoubleVirtual;
    Conta()
        : saldo(*this) {
    }
    Conta(Const const&)
        : saldo(*this) {
    }
    Conta& operator = (Conta const&) {
        return *this;
    }
    virtual ~Conta() {}
    DoubleVirtual saldo;
  private:
    virtual double calculaSaldo() const = 0;
};

#include <list>

class ContaSimples : public Conta {
  public:
    ContaSimples(double saldo)
        : saldo_(saldo) {
    }
private:
    double saldo_;
    virtual double calculaSaldo() const {
        return saldo_;
    }
};

class Portfolio : public Conta {
  public:
    ~Portfolio() {
        for(std::list<Conta*>::iterator i = contas.begin();
            i != contas.end(); ++i)
            delete *i;
    }
    void acrescentaConta(Conta* nova_conta) {
```

```
            contas.push_back(nova_conta);
        }
      private:
        std::list<Conta*> contas;
        virtual double calculaSaldo() const {
            double saldo = 0.0;
            for(std::list<Conta*>::const_iterator i = contas.begin();
                i != contas.end(); ++i)
                saldo += (*i)->saldo;
            return saldo;
        }
};
```

O programa de teste é agora mais simples, pois aparenta aceder directamente a um atributo para obter o saldo das contas:

```
#include <iostream>

using namespace std;

int main()
{
    // Constrói duas contas simples (ponteiros para Conta!):
    Conta* c1 = new ContaSimples(10);
    Conta* c2 = new ContaSimples(30);

    // Mostra saldos das contas simples:
    cout << c1->saldo << endl;
    cout << c2->saldo << endl;

    // Constrói uma conta portfolio e acrescenta-lhe as duas contas:
    Portfolio* p = new Portfolio;
    p->acrescentaConta(c1);
    p->acrescentaConta(c2);

    // Guarda endereço da conta portfolio no ponteiro c para Conta:
    Conta* c = p;

    // Mostra saldo da conta portfolio:
    cout << c->saldo << endl;
}
```

Este truque pode ser refinado definindo uma classe C++ genérica `TipoVirtual` que simplifique a sua utilização mais genérica:

```
template <typename B, typename T>
```

```
class TipoVirtual {
  public:
    TipoVirtual(B& base, T (B::*calcula)() const)
        : base(base), calcula(calcula) {
    }
    operator T () const {
        return (base.*calcula)();
    }
  private:
    // Guarda-se uma referência para a classe a que variável diz respeito:
    B& base;
    // Guarda-se um ponteiro para a função membro que devolve o valor da variável:
     T (B::*calcula)() const;
};

class Conta {
  public:
    Conta()
        : saldo(*this, &calculaSaldo) {
    }
    Conta(Const const&)
        : saldo(*this, &calculaSaldo) {
    }
    Conta& operator = (Conta const&) {
        return *this;
    }
    virtual ~Conta() {}
    friend TipoVirtual<Conta, double>;
    TipoVirtual<Conta, double> saldo;
  private:
    virtual double calculaSaldo() const = 0;
};

// O resto tudo igual.
```

Com esta classe C++ genérica definida, a utilização de "variáveis virtuais" exige apenas pequenas alterações na hierarquia de classes. Note-se que:

1. Nada se perdeu em encapsulamento. A variável `saldo` é parte da interface mas não é parte da implementação!

2. Nada se perdeu em polimorfismo. Continua a existir uma função para devolução do saldo. Esta função pode (e deve) ser especializada em classes derivadas.

3. Mas que sucede quando se copiam instâncias de classes com variáveis virtuais? Classes com variáveis virtuais têm de definir o construtor por cópia! Têm também de definir o operator de atribuição por cópia!

4. É possível construir outra classe C++ genérica para permitir acesso completo, incluindo escritas. Essa classe C++ genérica fica como exercício!

C.5 Persistência simplificada

!!Texto a completar! Verificar se ainda existe versão com registo automático.

```
/** Este módulo contém uma única classe C++ genérica muito simples, Serializador,
    que serve para simplificar a serialização polimórfica de hierarquias de classes.

    Esta técnica foi desenvolvida por mim independentemente, embora tenha
    benificiado com algumas ideias de:

    Jim Hyslop e Herb Sutter, "Conversations: Abstract Factory, Template Style",
    CUJ Experts, Junho de 2001.

    Jim Hyslop and Herb Sutter, "Conversations: How to Persist an Object",
    CUJ Experts, Julho de 2001.

    @see Serializador.

    Copyright © Manuel Menezes de Sequeira, ISCTE, 2001. */

#ifndef SERIALIZADOR_H
#define SERIALIZADOR_H

#include <map>
#include <string>
#include <typeinfo>

/** Uma classe C++ genérica que serializa polimorficamente objectos referenciados por
    ponteiros. Os únicos requisitos feitos à hierarquia de classes são que dois
    métodos têm de estar presentes:

    Construtor por desserialização:
    Classe(istream& entrada);

    Operação polimórfica de serialização:
    virtual void serializa(istream& saida) const;

    Conceitos (novo): Serializável? */
template <class Base>
class Serializador {
  public:
```

```
  /** Usado para serializar polimorficamente uma instância da hierarquia
      encimada pela classe Base. Um identificador da classe é colocado no
      canal antes dos dados da classe propriamente ditos. */
  static void serializa(std::ostream& saida, Base const* a);

  /** Usado para construir uma nova instância da hierarquia de classes através
      da sua desserialização a partir de um canal de entrada. Assume-se que o
      canal contém um identificador de classe, tal como inserido pelo método
      serializa() acima. */
  static Base* constroi(std::istream& entrada);

  /** Uma classe usada para simplificar o registo de classes na hierarquia. */
  template <class Derivada>
  class Registador {
    public:
      Registador() {
          Serializador<Base>::
              regista_(typeid(Derivada).name(),
                       &Serializador<Base>::template
                           constroi_<Derivada>);
      }
  };

private:

  typedef Base* Criador(std::istream&);

  static Serializador& instancia();

  Serializador() {}

  Serializador(Serializador const&);

  Serializador& operator = (Serializador const&);

  std::map<std::string, Criador*> registo;

  template <class Derivada>
  static Base* constroi_(std::istream& entrada);

public:
  // Devia ser privado e Registador devia ser amiga de Serializador,
  // mas o GCC estoirou...
  static void regista_(std::string const& nome_da_classe,
                       Criador* criador);
```

```
};

template <class Base>
template <class Derivada>
inline Base* Serializador<Base>::constroi_(std::istream& entrada)
{
    return new Derivada(entrada);
}

template <class Base>
inline void Serializador<Base>::
    regista_(std::string const& nome_da_classe, Criador* criador)
{
    // Uma asserção é apropriada aqui, uma vez que o registo de classes deve
    // ocorrer antes de a função main() começar...
    assert(instancia().registo.count(nome_da_classe) == 0);

    instancia().registo[nome_da_classe] = criador;
}

template <class Base>
Base* Serializador<Base>::constroi(std::istream& entrada)
{
    // Isto deveria ser uma asserção lançadora de excepções (pré-condição):
    if(not entrada)
        throw string("canal de entrada errado em "
                     "Serializador<Base>::constroi");

    std::string nome_da_classe;
    getline(entrada, nome_da_classe);

    // Isto deveria ser uma excepção para erros extraíndo de um canal de entrada:
    if(not entrada)
        throw string("erro extraíndo identificador de classe");

    // Isto deveria ser uma excepção para classe por registar:
    if(instancia().registo.count(nome_da_classe) == 0)
        throw std::string("Classe '") + nome_da_classe +
              "' por registar!";

    return instancia().registo[nome_da_classe](entrada);
}

template <class Base>
inline void Serializador<Base>::serializa(std::ostream& saida,
                                          Base const* a)
```

```
{
    // Isto deveria ser uma asserção lançadora de excepções (pré-condição):
    if(not saida)
        throw string("canal de saída errado em "
                     "Serializador<Base>::serializa");

    // Desnecessário em rigor, mas útil (traz alguma simetria)...
    // Isto deveria ser uma excepção para classe por registar:
    if(instancia().registo.count(typeid(*a).name()) == 0)
        throw std::string("Classe '") + typeid(*a).name() +
              "' por registar!";

    saida << typeid(*a).name() << std::endl;

    // Isto deveria ser uma excepção para erros inserindo num canal de saída:
    if(not saida)
        throw string("erro inserindo identificador de classe");

    a->serializa(saida);
}

template <class Base>
Serializador<Base>& Serializador<Base>::instancia()
{
    static Serializador instancia;

    return instancia;
}

#endif // SERIALIZADOR_H
```

!!Seguem-se os ficheiros de teste: a.H, a.C, b.H, b.C, c.H, c.C, d.H, d.C, teste.C

`a.H`

```
#ifndef A_H
#define A_H
#include <iostream>
#include <string>

class A {
  public:
    A();
    A(std::istream& entrada);

    virtual void serializa(std::ostream& saida) const;
```

```
};

inline A::A()
{
    std::clog << "A criado" << std::endl;
}

inline A::A(std::istream& entrada)
{
    if(not entrada)
        throw string("canal de entrada erra-
do em A::A(std::istream&)");

    std::string texto;
    std::getline(entrada, texto);

    if(not entrada or texto != "A")
        throw string("erro desserializando A");

    std::clog << "A criado de canal" << std::endl;
}

inline void A::serializa(std::ostream& saida) const
{
    if(not saida)
        throw string("canal de saída errado em A::serializa");

    saida << "A" << std::endl;

    if(not saida)
        throw string("erro serializando A");

    std::clog << "A serializado" << std::endl;
}

#endif // A_H
```

a.C

```
#include "a.H"

#include "serializador.H"

namespace {
    Serializador<A>::Registador<A> faz_registo;
}
```

b.H

```
#ifndef B_H
#define B_H

#include <iostream>
#include <string>

#include "a.H"

class B : public A {
  public:
    B();
    B(std::istream& entrada);

    virtual void serializa(std::ostream& saida) const;
};

inline B::B()
{
    std::clog << "B criado" << endl;
}

inline B::B(std::istream& entrada)
    : A(entrada)
{
    std::string texto;
    std::getline(entrada, texto);

    if(not entrada or texto != "B")
        throw string("erro desserializando B");

    std::clog << "B criado de canal" << std::endl;
}

inline void B::serializa(ostream& saida) const
{
    if(not saida)
        throw string("canal de saída errado em B::serializa");

    A::serializa(saida);

    saida << "B" << std::endl;

    if(not saida)
        throw string("erro serializando B");
```

```
    std::clog << "B serializado" << std::endl;
}

#endif // B_H
```

b.C

```
#include "b.H"

#include "serializador.H"

namespace {
    Serializador<A>::Registador<B> faz_registo;
}
```

c.H

```
#ifndef C_H
#define C_H

#include <iostream>
#include <string>

#include "a.H"
class C : public A {
  public:
    C();
    C(std::istream& entrada);

    virtual void serializa(std::ostream& saida) const;
};

inline C::C()
{
    std::clog << "C criado" << endl;
}

inline C::C(std::istream& entrada)
    : A(entrada)
{
    std::string texto;
    std::getline(entrada, texto);

    if(not entrada or texto != "C")
        throw string("erro desserializando C");
```

```
    std::clog << "C criado de canal" << std::endl;
}

inline void C::serializa(ostream& saida) const
{
    if(not saida)
        throw string("canal de saída errado em C::serializa");

    A::serializa(saida);

    saida << "C" << std::endl;

    if(not saida)
        throw string("erro serializando C");

    std::clog << "C serializado" << std::endl;
}

#endif // C_H
```

c.C

```
#include "c.H"

#include "serializador.H"

namespace {
    Serializador<A>::Registador<C> faz_registo1;
    Serializador<C>::Registador<C> faz_registo2;
}
```

d.H

```
#ifndef D_H
#define D_H

#include <iostream>
#include <string>

#include "c.H"

class D : public C {
  public:
    D();
    D(std::istream& entrada);

    virtual void serializa(std::ostream& saida) const;
```

```
};

inline D::D()
{
    std::clog << "D criado" << endl;
}

inline D::D(std::istream& entrada)
    : C(entrada)
{
    std::string texto;
    std::getline(entrada, texto);

    if(not entrada or texto != "D")
        throw string("erro desserializando D");

    std::clog << "D criado de canal" << std::endl;
}

inline void D::serializa(ostream& saida) const
{
    if(not saida)
        throw string("canal de saída errado em D::serializa");

    C::serializa(saida);

    saida << "D" << std::endl;

    if(not saida)
        throw string("erro serializando D");

    std::clog << "D serializado" << std::endl;
}

#endif // D_H
```

d.C

```
#include "d.H"

#include "serializador.H"

namespace {
    Serializador<A>::Registador<D> faz_registo1;
    Serializador<C>::Registador<D> faz_registo2;
```

```
    /* Mensagens de erro não são difícieis de entender:

       class X {
         public:
           X(std::istream&) {}
       };
       Serializador<C>::Registador<X> faz_registo3;
       Serializador<D>::Registador<C> faz_registo4;
    */
}
```

teste.C

```
#include <iostream>
#include <fstream>

using namespace std;

#include "a.H"
#include "b.H"
#include "c.H"
#include "d.H"

#include "serializador.H"

class E : public D {
  public:
    E() {}
    E(istream&) {}
};

int main()
{
    A* a = new A;
    A* b = new B;
    A* c = new C;
    A* d = new D;

    try {
        ofstream saida("teste1.txt");

        Serializador<A>::serializa(saida, a);
        Serializador<A>::serializa(saida, b);
        Serializador<A>::serializa(saida, c);
        Serializador<A>::serializa(saida, d);
```

```
        saida.flush();

        ifstream entrada("teste1.txt");

        A* a = Serializador<A>::constroi(entrada);
        A* b = Serializador<A>::constroi(entrada);
        A* c = Serializador<A>::constroi(entrada);
        A* d = Serializador<A>::constroi(entrada);
    } catch(string excepcao) {
        cerr << "Algo de errado... " << excepcao << endl;
    }

    C* x = new C;
    C* y = new D;

    try {
        ofstream saida("teste2.txt");

        Serializador<C>::serializa(saida, x);
        Serializador<C>::serializa(saida, y);

        saida.flush();

        ifstream entrada("teste2.txt");

        C* x = Serializador<C>::constroi(entrada);
        C* y = Serializador<C>::constroi(entrada);
    } catch(string excepcao) {
        cerr << "Algo de errado... " << excepcao << endl;
    }

    // Falha...
    try {
        A* outro = new E;
        Serializador<A>::serializa(cout, outro);
    } catch(string excepcao) {
        cerr << "Algo de errado... " << excepcao << endl;
    }
}
```