Aula 9: Templates

Paulo Cortez

2004/2005

Técnicas de Programação I
Licenciatura em Engenharia Electrónica e Computadores

http://www.dsi.uminho.pt/disciplinas/EITPI

Objectivos da Aula

- Manipular *templates* em C++;

Templates

- Considere a função que calcula o somatório de um array:

```
int sumVector(int *array, int n)
    {
        int sum(0); // mais eficiente que: int sum=0;
        for (int idx = 0; idx < n; ++idx)
            sum += array[idx];
        return (sum);
    }
```

- Se se quisesse efectuar a mesma função com um array de outro tipo (exemplo: `double`), teria de repetir todo o código;
- Para contornar situações semelhantes, em C++ existe um mecanismo especial chamado de **template**;

Definição de templates

```
// Definem-se da mesma forma que uma função normal
// a excepção é que podem receber e devolver tipos genéricos
template <typename T> // um tipo genérico
    T sumVector(T *array, int n)
    {
        T sum=0;
        for (int idx = 0; idx < n; ++idx)
            sum += array[idx];
        return (sum);
    }
template <typename T1, typename T2> // dois tipos genéricos
double prodVector(T1 *a1, T2 *a2, int n) // produto interno
    {
            double res=0;
            for (int i = 0; i < n; i++)
                res += a1[i]*a2[i];
            return (res);
    }
```

Exemplo de utilização

```
int main()
{ int n[]={1,2};
  double m[]={1.1,2.2};

// soma de vectores de inteiros
  cout << "soma:" << sumVector(n,2) << "\n"; // 3
// soma de vectores de reais
  cout << "soma:" << sumVector(m,2) << "\n"; // 3.3
// produto interno vector de inteiros * vector de reais
  cout << "prod:" << prodVector(n,m,2) << "\n"; // 5.5
// produto interno vector de reais * vector de inteiros
  cout << "prod:" << prodVector(m,n,2) << "\n"; // 5.5
  return 0;
}
```

Declaração explícita

- Quando se sabe à partida que tipos se vão utilizar, é possível declarar explicitamente esses tipos:

```
template <typename T> // um tipo genérico
    T sumVector(T *array, int n)
    { ... }
template int sumVector<int>(int *, int);
template double sumVector<double>(double *, int);

int main()
{ ... }
```

- As declarações explícitas permitem que o compilador crie versões normais das funções, reduzindo o tempo de compilação;

Classes Template

```
// Também é possível construir classes template:
// ficheiro: vector.h
#ifndef _Vector_h_
#define _Vector_h_
#include <iostream>
using namespace std;
template <typename T>
class Vector
{ int d_size; T *d_v;
  public:
   Vector(int size):d_size(size){d_v=new T[d_size];}
   ~Vector(){delete d_v;}
   // acessors
   T operator[](int index) const{return d_v[index];}
   int size() const { return d_size;}
```

Classes Template

```
// continuação do vector.h
// manipulators
T &operator[](int index){ return d_v[index];}
void enlarge(int newsize)
    { T *aux=new T[newsize];
      for(int i=0;i<d_size;i++) aux[i]=d_v[i];
      delete []d_v; d_v=aux; d_size=newsize;
    }
friend ostream &operator<<(ostream &o, Vector<T> const &t)
    { for(int i=0;i<t.d_size;i++) o << t.d_v[i] << " ";
      return o << "\n";
    }
};
#endif
```

Classes Template

```
// ficheiro: main.cc
#include "vector.h"
int main()
{ Vector<int> vi(0);    // vector de inteiros
  vi.enlarge(2);        // alargar vector
  vi[0]=1; vi[1]=2;
  cout << vi;           // 1 2
  Vector<double>vd(2); // vector de reais
  vd[0]=1.1; vd[1]=2.2;// 1.1 2.2
  cout << vd;
  return 0;
}
```

Classes Descendentes Template

```
template<typename T>
class Base
    { T d_t;
        public:
            Base(T const &t):d_t(t);
    };
template<typename T>
class Derived: public Base<T>
    { public:
            Derived(T const &t) : Base(t){}
    };
class Ordinary: public Base<int> //
    {
        public:
            Ordinary(int x) : Base(x) {}
    };
```

Classes Descendentes Template

```
int main()
{
   Base <int> b(3);
   Derived<double> d(3.3);
   Ordinary o(5);
   return 0;
}
```