Definição de tipos pelo usuário

Gonzalo Travieso

2018

1 Tipos definidos pelo usuário

Até agora, estivemos usando tipos de dados pré-definidos seja na própria linguagem, seja em alguma de suas bibliotecas padrão.

Tipos de dados são extremamente importantes em C++. Já vimos que o compilador exige que cada variável tenha seu tipo declarado na sua definição (explicitamente ou por dedução de tipos quando usamos `auto`). Também vimos que o compilador não permite mistura de tipos diferentes, a menos que haja alguma forma de conversão entre eles.

Além dos tipos pré-definidos na linguagem, o usuário pode definir novos tipos. A definição de novos tipos pode contribuir para a compreensão do código e para sua simplicidade, dependendo da situação. Veremos aqui algumas das formas que podem ser usadas para definir novos tipos.

2 Novos nomes para tipos existentes

Uma forma simples de definir um novo tipo é dar um novo nome a um tipo pré-existente. A razão principal para se fazer isso é facilitar a compreensão do código. Como dito frequentemente, códigos são textos para a transmissão de informação entre programadores (com o efeito colateral de poderem ser executados por um computador), e eles são mais frequentemente lidos do que escritos ou alterados. Desta forma, é apropriado gastar um pouco mais de tempo durante a escrita do código para facilitar a sua leitura.

Considere por exemplo o caso de um código que possui uma função para calcular a posição depois de decorrido um certo intervalo de tempo de um corpo sujeito a movimento uniformemente acelerado. Essa função poderia ser definida como abaixo:

```
double posicao_MUA(double x0, double v0,
                   double a, double dt)
{
    return x0 + v0 * dt + 0.5 * a * dt * dt;
}
```

O código está correto, mas note como é usado o tipo `double` para unidades de significado físico distinto. Podemos facilitar a compreensão do código definindo tipos de nome distinto para cada uma dessas unidades. Para isso, usamos a palavra-chave `using`:

```
using Posicao = double;
using Velocidade = double;
using Aceleracao = double;
using Tempo = double;
Posicao posicao_MUA(Posicao x0, Velocidade v0,
                    Aceleracao a, Tempo dt)
{
    return x0 + v0 * dt + 0.5 * a * dt * dt;
}
```

Nesta versão, fica mais claro o significado de cada parâmetro e do valor de retorno. Note que os tipos definido provavelmente serão usados em diversos locais no restante do código, ajudando também a compreensão nesses pontos.

Também se ganha um pouco de flexibilidade. Por exemplo, se agora queremos alterar o código para lidar com movimento em três dimensões, o código fica:

```
constexpr int NDIM = 3;
using Posicao = std::array<double, NDIM>;
using Velocidade = std::array<double, NDIM>;
using Aceleracao = std::array<double, NDIM>;
using Tempo = double;
Posicao posicao_MUA(Posicao x0, Velocidade v0,
                    Aceleracao a, Tempo dt)
{
    Posicao x;
    for (int i = 0; i < NDIM; ++i) {
        x[i] = x0[i] + v0[i] * dt + 0.5 * a[i] * dt * dt;
    }
    return x;
}
```

Todas as variável dos tipos `Posicao`, `Velocidade` e `Aceleracao` e os códigos associados deverão ser readequadas para carregar os valores das três coordenadas, mas o compilador indicará isso (se você esquecer de alterar algum local); isso não aconteceria se estivéssemos usando `double` antes.

3 Tipos enumerados

Em alguma situações, iremos utilizar um tipo de dados que:

- Tem apenas um número limitado (e pequeno) de valores possíveis.
- Queremos dar nomes a cada um dos valores.

Nestas situações, devemos declarar um novo tipo usando `enum class`. Por exemplo, se um programa vai lidar com dias da semana, podemos declarar:

```
enum class Semana {
    domingo, segunda, terca, quarta,
    quinta, sexta, sabado
};
```

Isto declara um novo tipo de dados denominado `Semana`; esse tipo de dados tem 7 valores possíveis, que são os apresentados entre chaves, separados por vírgula. Note o uso de ponto-e-vírgula depois da declaração.

Para usar esse novo tipo, basta declarar uma variável desse tipo e colocar valores nela, conforme desejado. Os valores são acessados com o uso do operador de escopo `::` para indicar que eles são parte do `enum Semana`:

```
Semana hoje = Semana::quarta;

if (hoje == Semana::quarta) {
    std::cout << "Tem aula\n";
}
```

Múltiplos tipos `enum class` podem ter identificadores com o mesmo nome, sem gerar problema (porisso precisamos usar o operador de escopo). Por exemplo, o mesmo programa que tem o tipo `Semana` pode também definir um tipo `Ordem` da seguinte forma:

```
enum class Ordem {
    primeira, segunda, terceira,
    quarta, quinta
};
```

Considerando essas definições, algumas operações são permitidas, outras não (veja comentários)

```
Semana dia;
Ordem marcha;

dia = Semana::quarta; // OK
marcha = Ordem::quarta; // OK
dia = quarta; // ERRO: que quarta?
dia = Ordem::quarta; // ERRO: tipo errado
marcha = Ordem::sexta; // ERRO: inexistente
```

4 Tipos estruturados

Um caso comum é quando temos diversas informações, não necessariamente do mesmo tipo, que são em princípio relacionadas entre si. Se armazenamos essas informações em variáveis distintas, a relação entre elas fica perdida no código (e deve ser indicada explicitamente em comentários). Vamos supor que um programa lida com partículas, e para uma partícula deve guardar um identificador (um número inteiro), sua posiçao e sua velocidade atuais (3D). Se temos N partículas, poderíamos fazer isso da seguinte forma:

```
// id[i] e' identificador da particula i
std::vector<int> id(N);
// pos[i] e' posicao da particula i
std::vector<Posicao> pos(N);
// vel[i] e' velocidade da particula i
std::vector<Velocidade> vel(N);
```

Note como as informações sobre cada partícula estão dispersas em elementos de três vetores distintos. Ao invés disso, podemos definir um tipo para representar partículas que guarda todas as informações de uma delas:

```
struct Particula {
    int id;
    Posicao pos;
    Velocidade vel;
};
```

Isto define um novo tipo denominado `Particula` que tem as informações indicadas (um tipo `int` denominado `id`, um tipo `Posicao` denominado `pos` e um tipo `Velocidade` denominado `vel`). Feito isso, se queremos lidar com uma partícula e suas informações, declaramos uma variável desse tipo e acessamos suas partes, denominadas **membros**, através do *operador de acesso a membro* `.` (ponto); por exemplo (supondo a definição de posição e velocidade através de `array<double, 3>` apresentada acima):

```
Particula p;
p.id = 137;
p.pos = Posicao{1.0, 0.0, 3.0};
p.vel = Velocidade{0.0, 0.0, 0.0};
```

5 Exemplo final

Como um último exemplo, usando os três tipos discutidos, considere o código abaixo que lê informações sobre funcionários de uma empresa (código de identificação, nome, nome do cargo e ano de ingresso) de um arquivo onde esses dados estão apresentados por linha para cada funcionário, cada campo separado por vírgula do seguinte, guarda os dados em um vetor e mostra esses dados ordenados por diversos campos. Veja se você entende todos os detalhes.

```
#include <string>
#include <vector>
#include <functional>
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iomanip>

struct Pessoa {
    int id;
    std::string nome;
    std::string funcao;
    int ano_ingresso;
};

enum class Ordena {
    por_id, por_nome, por_funcao, por_ingresso
};
```

```
using FunComp = std::function<bool (Pessoa const &,
                                    Pessoa const &)>;

FunComp funcao_comparacao(Ordena ordem)
{
    FunComp f;
    switch (ordem) {
    case Ordena::por_id:
        f = [] (Pessoa const &a,
                Pessoa const &b)
            {
                return a.id < b.id;
            };
        break;
    case Ordena::por_nome:
        f = [] (Pessoa const &a,
                Pessoa const &b)
            {
                return a.nome < b.nome;
            };
        break;
    case Ordena::por_funcao:
        f = [] (Pessoa const &a,
                Pessoa const &b)
            {
                return a.funcao < b.funcao;
            };
        break;
    case Ordena::por_ingresso:
        f = [] (Pessoa const &a,
                Pessoa const &b)
            {
                return a.ano_ingresso < b.ano_ingresso;
            };
    }
    return f;
}

Pessoa le_pessoa(std::istream &is)
{
    Pessoa quem;
    char virgula;

    is >> quem.id;
    is >> virgula;
    std::getline(is, quem.nome, ',');
    std::getline(is, quem.funcao, ',');
    is >> quem.ano_ingresso;

    return quem;
```

```
}


void mostra_colaboradores(std::vector<Pessoa> const &pessoas, Ordena ordem)
{
    auto ordenado{pessoas};
    std::sort(begin(ordenado), end(ordenado),
              funcao_comparacao(ordem));
    for (auto pessoa: ordenado) {
        std::cout << std::setw(4) << pessoa.id
                  << std::setw(30) << pessoa.nome
                  << std::setw(20) << pessoa.funcao
                  << std::setw(8) << pessoa.ano_ingresso
                  << std::endl;
    }
}

int main(int, char *[])
{
    std::vector<Pessoa> todos;

    std::ifstream dados("funcionarios.dat");
    while (dados.good() && !dados.eof()) {
        auto nova = le_pessoa(dados);
        if (dados.good()) {
            todos.push_back(nova);
        }
    }

    std::cout << "Por id:\n";
    mostra_colaboradores(todos, Ordena::por_id);
    std::cout << "Por nome:\n";
    mostra_colaboradores(todos, Ordena::por_nome);
    std::cout << "Por funcao:\n";
    mostra_colaboradores(todos, Ordena::por_funcao);
    std::cout << "Por ingresso:\n";
    mostra_colaboradores(todos, Ordena::por_ingresso);

    return 0;
}
```