Tipi Derivati e Spazi dei Nomi

Programmazione Avanzata e Parallela
2022/2023

Alberto Casagrande

Tipi Semplici e Tipi Strutturati

In taluni casi, i tipi semplici non sono sufficienti

Per esempio, per rappresentare:

- i razionali dobbiamo usare due interi
- gli array necessitiamo di un puntatore e della loro lunghezza
- i cambia-monete abbiamo bisogno di due array (uno per i tagli e uno per il serbatorio delle monete)

Vorremo un modo per costruire dei **tipi strutturati**

Strutture (C++)

Le **strutture** definiscono tipi di dato derivati

I valori del tipo di dato derivato si chiamano **oggetti**

- **membri** (**variabili d'istanza**): sono variabili associate ad ogni istanza della struttura e i cui valori definiscono il valore dell'oggetto
- **metodi** (**funzioni di membro**): sono funzioni che operano sugli oggetti e hanno accesso ai membri

[Vedi Cap 9.4 di PPP e Cap 8.2 di CPL]

Una Struttura d'Esempio: `IntArray`

```
struct IntArray {                   // IntArray è il nome della struttura
  size_t len;                       // questi sono i membri: proprietà delle
  int* values;                      // singole istanze della struttura

  int& at(const size_t i) {         // questo è un metodo che opera sui dati
    return values[i];               // di ogni istanza
  }

  size_t size();                    // questa è la dichiarazione del metodo
};

size_t IntArray::size() {           // ecco la definizione del metodo
  return size;                      // size() per IntArray
}
```

I Parametri dei Metodi

Il metodo `IntArray::at(const size_t i)` *sembra* avere un parametro

In realtà ne ha **due**:

- **quello esplicito**, i.e., `i`
- **quello implicito**, i.e., l'istanza della struttura su cui opera

```
int& at(const size_t i) {
  return values[i];       // values di quale istanza?
                          // del parametro implicito
}
```

Strutture e Variabili

Le strutture sono dei tipi e possiamo dicharare delle variabili

```
IntArray a1;                          // a ogni variabile è associato un oggetto
IntArray a2{3, new int[3]};           // inizializzare i membri dell'oggetto
IntArray aa[10];                      // possiamo dichiarare array di oggetti

std::cout << a2.len << std::endl; // i membri possono essere valutati...

a1.values = new int[10];              // ...assegnati e...
a1.len = 10;

a1.at(i) = 5;                         // ...i membri di ogni istanza invocati
```

Metodi "Statici" o "di Classe"

Se il metodo non opera su un'instanza della struttura è un **metodo statico** o **di classe**

Li etichettiamo con la parola riservata `static`

```
struct Array {
  ...

  static const char* type_name() {          // questo metodo è puramente
    return "Array";                         // dimostrativo
  }
};
```

Invocazione di Metodi Statici

Possono essere invocati come i quelli non statici

```
IntArray a1;

std::cout << a1.type_name() << std::endl();
```

ma anche senza fare riferimento a un'istanza specifica

```
std::cout << IntArray::type_name() << std::endl();
```

Costruttori e Distruttori

Le strutture *possono* essere dotate di:

- uno o più metodi **costruttori** per inizializzare un oggetto
- un metodo **distruttore** invocato per al cancellamento di un'istanza

```
struct IntArray {
  ...

  IntArray();                          // i costruttori hanno lo stesso
  IntArray(const size_t length);       // nome della struttura

  ~IntArray() { delete[] values; }     // questo è il distruttore
};
```

Costruttori e Distruttori (Cont'd)

```
IntArray::IntArray(const size_t len)
 : len{len}, values{new int[len]}   // l'inizializzazione dei membri
{}                                  // definizione del costruttore
```

Durante l'inizializzazione, è possibile invocare altri costruttori

```
IntArray::IntArray(): IntArray(0) {}

IntArray::IntArray(const size_t len, const int init_value)
  : IntArray(len)
{
   for (size_t i=0; i<len; ++i) values[i] = init_value;
}
```

Usare i Costruttori e i Distruttori

```
{
  IntArray a;               // viene invocato IntArray()
  IntArray aa[10];          // viene invocato IntArray() 10 volte

  IntArray b(10);           // viene invocato IntArray(const size_t)
  IntArray c{12};           // viene invocato IntArray(const size_t)
  auto p = new int(15);     // invocato IntArray(const size_t)

  IntArray d(10, 5);        // viene invocato IntArray(const size_t, const int)

  delete p;     // prima di deallocare lo spazio viene invocato ~IntArray()

}  // il distruttore viene invocato su ogni oggetto che smette di esistere
```

Metodi `const`

Se un metodo non modifica l'oggetto, può essere definito `const`

```
struct IntArray {
  ...
  const size_t& size() const { return len; }  // questi metodi sono `const`
  const int& at(const size_t i) const;        // perché non modificano
  void println() const;                       // il valore dell'instanza
};
```

`at(const size_t i)` è sovraccaricato: il parametro implicito è `const`

Metodi `const` (Cont'd)

```
void IntArray::println() const {
  std::cout << "[";
  for (size_t i=0; i<len; ++i) {
    if ()
    std::cout << values[i];
  }
  std::cout << "]" << std::endl;
}
```

Un Puntatore all'Oggetto Corrente

`this` è un puntatore all'oggetto corrente

Possiamo accedere ai membri e ai metodi dell'oggetto con la sintassi dei puntatori

```
// ridimensiona l'array, ma perdi i dati
void IntArray::resize(const size_t len) {
  this->len = len;   // assegno al membro dell'oggetto `len`
                     // il valore del parametro formale `len`
  delete[] values;
  values = tmp;
}
```

Template di Strutture

```
template<typename T> struct Array { // Esistono template di strutture
  size_t len;
  T* values;   // il tipo del puntatore dipende dal parametro del template

  Array(const size_t len, const T& init_value);
  Array(const size_t len): len{len}, values{new T[len]};
  Array(): Array(0) {}

  T& at(const size_t i) { return values[i]; }
  const T& at(const size_t i) const { return values[i]; }
  const size_t& size() const { return len; }
  void println() const;   // dichiarazione del metodo
  ~Array() { delete[] value; }
};
```

Template di Strutture (Cont'd)

```
template<typename T>
Array<T>::Array<T>(const size_t len, const T& init_value)
  : Array(len)
{
  for (size_t i=0; i<len; ++i)
    values[i] = init_value;
}

template<typename T> void Array<T>::println() const {
  std::cout << "[";
  for (size_t i=0; i<len; ++i) {
    if (i>0) std::cout << ",";
    std::cout << values[i];
  }
  std::cout << "]" << std::endl;
}
```

Dichiarare i Template di Strutture

... come usare le strutture, ma con il tipo parametrico

```
Array<int> a{5}, b{7, 3};
Array<int> A[10];

Array<int>* c = new Array<int>(6);
```

Abbiamo già visto questa sintassi nelle dichiarazioni?

```
std::unique_ptr<int> sp{new int(7)};
std::shared_ptr<int> up;
```

Strutture vs Tipi

Non tutto è permesso nei tipi... e nelle strutture?

```
Array<int> a(2);

a.len = 10000;    // modifico il membro che rappresenta la lunghezza
a.println();      // cosa stiamo stampando?
```

Il codice sopra è sicuramente errato!!!

Vorremmo non consentire l'accesso ad alcuni membri o metodi

`public` and `private`

Limitano l'accesso ai membri e ai metodi

```
template<typename T> struct Array { // Esistono template di strutture
private:      // quanto segue non è visibile all'esterno della struttura
  size_t len;
  T* values;   // il tipo del puntatore dipende dal parametro del template

public:       // quanto segue è visibile a tutti
  Array(const size_t len);
  ...
};
```

`public` è implicito in `struct`

`public` and `private` (Cont'd)

Adesso...

```
Array<int> a(2);

a.len = 10000;   // ERRORE: len è un membro privato
```

Ma...

```
const int& at(const size_t i) const {
  return values[i];   // OK: perché 'at(const size_t)' è un metodo di
}                     // Array<T>
```

Funzioni e Strutture Amiche

`friend` concede l'accesso a `private` ad alcune funzioni/strutture

```
template<typename T> struct Array {
  ...
  friend void reset_to(Array<T>& A, const T val) {    // questa è una funzione
    for (size_t i=0; i<A.len; ++i)                    // non un metodo, ma
       A.values[i] = val;                             // len e values possono
  }                                                   // essere usati perché
};                                                    // è 'friend' di 'Array'
```

```
Array<int> A(10);

reset_to(A, 3);
```

La Parola Riservata `class`

Può essere usata al posto di `struct`

```
template<typename T> class Array {
private:
  size_t len;
  T* values;

public:
  Array(const size_t len);
  ...
};
```

L'unica differenza è che `private` è implicito in `class`

Overload degli Operatori come Metodi

Gli operatori `+`, `-`, `*`, `/`, `++`, `->`, `+=`, `-=`, `=`, `==`, `&&`, `!`, `[]`, `()`,... possono essere sovraccaricati come metodi...

```
template<typename T> class Array {
  ...
  Array<T> operator+(const Array<T>& a) const;    // il parametro mancante è
                                                  // l'oggetto corrente
  T& operator[](const size_t i) {
    return values[i];
  }

  const T& operator[](const size_t i) const {
    return values[i];
  }
};
```

Overload degli Operatori come Metodi (Cont'd)

```
template<typename T>
Array<T> Array<T>::operator+(const Array<T>& a) const {
  Array<T> res(this->len+a.len);  // creo un array `res` la cui dimensione è
                                  // la somma delle dimensioni di `a` e `b`

  // copio `a` all'inizio di `res`
  std::copy(values, values+len, res.values);  // #include <algorithm>

  // copio `b` all'inizio di `res`
  std::copy(a.values, a.values+a.len, res.values+len);

  return res;
}
```

Overload degli Operatori come Funzioni

Possiamo sovraccaricare `+`, `-`, `*`, `/`, `++`, `->`, `+=`, `-=`, `=`, `==`, `&&`, `!`, ... (ma non `[]` o `()`) anche come funzioni

```
template<typename T>
Array<T> operator+(const Array<T>& a, const Array<T>& b) {
  // Array<T> res(a.len+b.len);         // ERRORE: len è privato!!!
  Array<T> res(a.size()+b.size());

  // std::copy(a.values, ..);           // ERRORE: values è privato!!!
  for (size_t i=0; i<a.size(); ++i) res.at(i) = a.at(i);
  for (size_t i=0; i<b.size(); ++i) res.at(i+a.size()) = b.at(i);
  return res;
}
```

Usare gli Operatori Sovraccaricati

Indipendentemente da come sono implementati

```
Array<int> a(10, 5), b(5, 7);

std::cout << a[5] << std::endl;      // viene invocato Array<int>::operator[]

Array<int> c = a+b;                  // viene invocato Array<int>::operator+

a.println();
b.println();
c.println();
```

Altri Operatori Sovraccaricabiliti

Anche `<<` e `>>` sono degli operatori che:

- hanno come primo parametro uno stream in output/input
- possono essere applicati in sequenza

```
std::cout << "[" << 3.2 << a << 3 << std::endl;
```

L'ultima linea è come...

```
((((std::cout << "[") << 3.2) << a) << 3) << std::endl;
```

Sovraccaricare `<<`

```
template<typename T>
std::ostream& operator<<(std::ostream& os,     // è una "generalizzazione" di
                         const Array<T>& A) { // std::cout
  os << "[";
  for (size_t i=0; i<A.size(); ++i) {
    if (i>0) os << ",";

    os << A.at(i);
  }
  os << "]";

  return os;
}
```

Problemi Con `Array`

Cosa succede se provo a copiare un valore del tipo `Array`?

Copio i valori dei suoi membri `len` e `values`

Ma `values` è un puntatore...

```
Array<int> a;
Array<int> b;

b=a;      // PROBLEMA: cosa succede quando distruggo i valori di 'a' e 'b'?

Array<Array<int>> c(10, Array<int>(7,10));  // PROBLEMA: cosa succede quando
                                            // c viene de-allocato?
```

Overload dell'Assegnamento

```
template<typename T> class Array {
  ...
  Array<T>& operator=(const Array<T>& orig) {
    if (this == &orig) {    // se sono lo stesso oggetto, lascia perdere
      return *this;
    }

    delete[] values;        // cancello il vecchio array

    len = orig.len;         // aggiorno la lunghezza
    values = new T[len];    // alloco un nuovo array
    std::copy(orig.values, orig.values+len, values);  // copio i valori

    return *this;
  }
};
```

Con Overload dell'Assegnamento

Adesso

```
Array<int> a;
Array<int> b;

b=a;      // OK: a.values e b.values puntano a due aree di memoria diverse

Array<Array<int>> c(10, Array<int>(7,10));  // OK: c[i].values sono tutti
                                            // diversi
```

Ora possiamo anche scrivere

```
std::cout << "c: " << c << std::endl;
```

Una Classe per le Stringhe

`std::string` rappresenta le stringhe di caratteri

Tra le altre cose, possiamo:

- leggere e scrivere una stringa con gli operatori `<<` e `>>`
- estrarre una sottostringa con il metodo `substr`
- cercare una sottostringa con il metodo `find`
- concatenare due stringhe con la funzione `operator+`
- tentare di convertire la stringa in un valore con le funzioni `stoi`, `stoul`, `stof`

Esempio di Utilizzo di `std::string`

```
#include <iostream>
#include <string>

int main() {
  std::string s1, s2("test test");

  std::cin >> s1;
  int value_s1 = stoi(s1);

  std:cout << s2.find(" test") << std::endl
           << (s1+s2) << std::endl
           << "stoi(" << s1 << ")=" << value_si << std::endl;
  return 0;
}
```

Enumerazioni (Come il C)

Possono essere usati per definire un insieme di costanti

Es. I colori **bianco**, **grigio** e **nero** delle visite in profondità

```
enum color {white, grey, black};

void print_color(const color& c) {
  switch(c) {
    case white:
    ...
  }
}
```

Enumerazioni (Cont'd)

- possono essere usati come delle costanti *built-in*

```
print_color(grey);
```

- sono implicitamente convertibili agli interi

```
int a = white;
```

- non possono essere ri-assegnati

```
white = 0;                // errore
```

Scoped Enum (C++11)

Possono essere usati per definire un insieme di costanti

```
enum class color {white, grey, black};

void print_color(const color& c) {
  switch(c) {
    case color::white:     // i valori vanno preceduti dallo scope
    ...
  }
}
```

Scoped Enum (Cont'd)

non sono implicitamente convertibili agli interi

```
int a = color::white;        // errore color::white è un color
```

[Vedi Cap 9.5 di PPP e Cap 8.5 di CPL]

Gli Spazi dei Nomi (C++)

Consentono di raggruppare variabili, funzioni e tipi derivati

```
namespace Geometry {
  static const double pi = 3.1428...;

  class Triangle {
    ...
  };
  ...
}

namespace Statistic {
  double average(double a, double b);
  ...
}
```

Gli Spazi dei Nomi (Cont'd)

Ciò che è contenuto in un `namespace` può essere usato direttamente

```
auto a = Statistic::average(2.3, 1);    // indicando il namespace
```

o dopo aver specificato quale che stiamo usando quel `namespace`

```
using namespace Statistic;

auto a = average(2.3, 1);
```

[Vedi Cap 8.7 di PPP e Cap 14 di CPL]

Lo Spazio dei Nome `std`

Abbiamo già visto uno spazio dei nomi: `std`

```
Array<int> A(20, 1);

std::cout << A << std::endl;

std::unique_ptr<double> up{new double(2.2)};

std::copy(values, values+len, res.values);
```

Come e Dove Usare `using namespace`

Andrebbe usato all'interno di un blocco di una funzione/metodo

```
int main() {
  using namespace std;
  ...
}
```

Mai usarlo prima della dichiarazione di una classe o di una funzione

```
// molteplici using namespace

template<typename T>
Array<T> genera(const size_t i);   // qual'è il namespace di Array?
```