Callable

Il concetto callable

Molti algoritmi generici resi disponibili dalla libreria standard sono forniti in due differenti versioni, la seconda delle quali è parametrizzata rispetto a una "policy".

Ad esempio, per l'algoritmo `std::adjacent_find`, che intuitivamente ricerca all'interno di una sequenza la prima occorrenza di due elementi adiacenti ed equivalenti, abbiamo le seguenti dichiarazioni (in overloading):

```
template
FwdIter adjacent_find(FwdIter first, FwdIter last);

template
FwdIter adjacent_find(FwdIter first, FwdIter last, BinPred pred);
```

Nella prima versione, il predicato binario utilizzato per il controllo di equivalenza degli elementi è `operator==`.

> Si noti che istanze diverse del template possono usare definizioni diverse (in overloading) di `operator==`, ma il nome della funzione usata per il controllo di equivalenza è fissato (*"hard-wired"*).

La seconda versione consente invece di utilizzare un qualunque tipo di dato fornito dall'utente, a condizione che questo si comporti come predicato binario definito sugli elementi della sequenza. Tenendo a mente il *"duck typing"*, ci dovremmo quindi chiedere quali sono i modi legittimi di istanziare il parametro template `BinPred`.

In altre parole, ci chiediamo quali siano i tipi di dato concreti ammessi per il parametro funzione `pred`, cioè quelli che consentono di compilare correttamente il test

```
if (pred(*first, *next)) //...
```

dove `first` e `next` sono due iteratori dello stesso tipo.

Considerando un caso un po' meno specifico, ci chiediamo quali siano i tipi di dato `Fun` che consentono (ai propri valori `fun`) di essere intuitivamente utilizzati come nomi di funzione in una chiamata:

```
fun(arg1, ..., argN);
```

L'insieme di questi tipi di dato forma il concetto *"callable"* (i tipi "chiamabili", cioè "invocabili" come le funzioni):

- puntatori a funzione
- oggetti funzione
- expressioni lambda (dal C++11)

Torna all'indice

I puntatori a funzione

Nei pochi esempi concreti che abbiamo visto fino ad ora, abbiamo sempre istanziato i parametri "callable" usando un opportuno puntatore a funzione.

Per esempio, quando usiamo il nome della funzione

```
bool pari(int i);
```

per istanziare il predicato unario della `std::find_if`, il parametro typename `UnaryPred` viene legato al tipo concreto `bool (*)(int)` (puntatore ad una funzione che prende un argomento intero per valore e restituisce un bool).

> Da un punto di vista tecnico, sarebbe pure possibile passare le funzioni per riferimento (invece che per valore), evitando il type decay ed ottenendo quindi un riferimento invece che un puntatore. Siccome questa

alternativa NON porta alcun beneficio concreto (anzi, complica solo la comprensione del codice), è considerato pessimo stile.

Torna all'indice

Gli "oggetti funzione"

Oltre alle vere e proprie funzioni, vi sono altri tipi di dato i cui valori possono essere invocati come le funzioni e che quindi, in base al "duck typing", soddisfano i requisiti del concetto callable.

In particolare, una classe che fornisca una definizione (o anche più definizioni, in overloading) del metodo `operator()` consente ai suoi oggetti di essere utilizzati al posto delle vere funzioni nella sintassi della chiamata di funzione.

Esempio:

```
struct Pari {
 bool operator()(int i) const {
     return i % 2 == 0;
 }
};

int foo() {
 Pari pari;
 if (pari(12345)) ...
 ...
}
```

L'oggetto `pari` (di tipo `struct Pari`) non è una funzione ma, essendo fornito di un metodo `operator()`, può essere invocato come una funzione.

> Si noti che la dichiarazione di `operator()`, detto *operatore parentesi tonde* o anche *operatore di chiamata di funzione*, presenta due coppie di parentesi tonde: la prima fa parte del *nome* dell'operatore, la seconda fornisce la lista dei parametri per l'operatore.

Spesso l'operatore è marcato `const` perché gli oggetti funzione sono spesso *"stateless"* (non hanno stato, cioè non hanno dei dati membro) e quindi non sono modificati dalle invocazioni dei loro `operator()`.

Torna all'indice

Osservazioni

A prima vista, gli oggetti funzione potrebbero sembrare un modo complicato di risolvere un problema semplice: perché definire una classe con un metodo `operator()` quando posso, più semplicemente, passare direttamente il nome di una semplice funzione?

Da un punto di vista tecnico, gli oggetti funzione possono essere usati per ottenere un vantaggio in termini di efficienza rispetto alle normali funzioni: in particolare, l'uso degli oggetti funzione fornisce al compilatore più opportunità per l'ottimizzazione del codice.

Si consideri un programma che lavori su un `vector` di interi e istanzia più volte la funzione generica `std::find_if` per effettuare ricerche nel `vector` usando criteri di ricerca (cioè, predicati unari) diversi.

Consideriamo i seguenti predicati espressi mediante funzioni:

```
bool pari(int i);
bool dispari(int i);
bool positivo(int i);
bool negativo(int i);
bool maggiore_di_1000(int i);
bool numero_primo(int i);
```

Queste sei funzioni hanno tutte lo stesso tipo `bool(int)`, ovvero funzione che prende un argomento `int` per valore e restituisce un `bool`. Di conseguenza, quando si istanzia l'algoritmo `std::find_if` sul vettore usando i sei predicati, si

ottiene ogni volta la stessa identica istanza del template di funzione.

Se, per comodità, usiamo i seguenti alias di tipo

```
using Iter = std::vector::iterator
using Ptr = bool (*)(int);
```

la specifica istanza ottenuta sarà la seguente:

```
Iter std::find_if(Iter first, Iter last, Ptr pred);
```

Il codice generato, quindi, è unico e deve gestire correttamente tutte le sei possibili invocazioni: di conseguenza, la chiamata al predicato è implementata come chiamata di funzione (attraverso il puntatore a funzione).

Supponiamo ora che i sei predicati siano stati invece implementati mediante oggetti funzione, ovvero definendo sei classi `Pari`, `Dispari`, `Positivo`, ecc...

> Nota: l'uso dell'iniziale maiuscola è solo una convenzione, utile per evitare di fare confusione.

In questo caso, quando si istanzia l'algoritmo `std::find_if` sul vettore usando i sei oggetti funzione, siccome i tipi degli oggetti funzione sono distinti si otterranno sei diverse istanze del template di funzione:

```
Iter std::find_if(Iter first, Iter last, Pari pred);
Iter std::find_if(Iter first, Iter last, Dispari pred);
Iter std::find_if(Iter first, Iter last, Positivo pred);
...
```

Quando genera il codice per una delle sei istanze, il compilatore vede l'invocazione di uno solo dei sei metodi `operator()` e quindi può ottimizzare il codice per quella specifica invocazione (per esempio, facendo l'espansione in linea della chiamata). Si ottiene quindi un codice eseguibile più grande (sei istanze invece di una sola), ma meglio ottimizzabile e quindi potenzialmente più efficiente.

Torna all'indice

Le espressioni lambda

Capita frequentemente che una determinata funzione (o un oggetto funzione) debba essere fornita come callable ad una invocazione di un algoritmo generico. In alternativa, che una funzione (o oggetto funzione) debba essere definita a fronte di un unico punto del codice che la invoca.

In questi casi, fornire la definizione della funzione (o della classe che implementa un oggetto funzione equivalente) presenta alcuni svantaggi:

- occorre inventare un nome appropriato;
- occorre fornire la definizione in un punto diverso del codice rispetto all'unico punto di uso, potenzialmente distante.

Le espressioni lambda (dette anche funzioni lambda) forniscono una comoda sintassi abbreviata per potere definire un oggetto funzione "anonimo" e immediatamente utilizzabile.

> Nota: le espressioni lambda sono state introdotte con lo standard C++11 e sono state oggetto di estensioni negli standard C++14 e C++17.

Esempio: istanziazione di `std::find_if` con una lambda expression che implementa il predicato `pari` sul tipo T.

```
void foo(const std::vector& v) {
 auto iter = std::find_if(v.begin(), v.end(),
                          [](const long& i) {
                         return i % 2 == 0;
                        });
  // ... usa iter
}
```

L'espressione lambda è data dalla sintassi

```
  [](const long& i) { return i % 2 == 0; }
```

dove si distiguono i seguenti elementi:

```
[]                        // capture list (lista delle catture)
(const long& i)           // lista dei parametri (opzionale)
{ return i % 2 == 0; }    // corpo della funzione
```

In questo esempio, la lista delle catture è vuota; inoltre, il tipo di ritorno è omesso, in quanto viene dedotto dall'istruzione di return contenuta nel corpo della funzione. Volendo (ma di solito non si fa), è possibile specificarlo con la sintassi del *"trailing return type"*, che usa l'operatore freccia:

```
  [](const long& i) -> bool { return i % 2 == 0; }
```

Questo uso della espressione lambra all'interno della invocazione della `std::find_if` corrisponde intuitivamente alle seguenti operazioni:

1. Definizione di una classe "anonima" per oggetti funzione, cioè una classe dotata di un nome univoco scelto dal sistema.
2. Definizione all'interno della classe di un metodo `operator()` che ha i parametri, il corpo e il tipo di ritorno specificati (o dedotti) dalla lambda expression.
3. Creazione di un oggetto funzione *"anonimo"*, avente il tipo della classe anonima suddetta, da passare alla `std::find_if`.

In pratica, è come se il programmatore avesse scritto:

```
struct Nome_Univoco {
 bool operator()(const long& i) const { return i % 2 == 0; }
};

auto iter = std::find_if(v.begin(), v.end(), Nome_Univoco());
```

La lista delle catture può essere usata quando l'espressione lambda deve potere accedere a variabili locali visibili nel punto in cui viene creata (che è diverso dal punto in cui verrà invocata). Supponiamo per esempio di volere trovare il primo valore della sequenza che sia maggiore del parametro "soglia":

```
void foo(const std::vector& v, long soglia) {
 auto iter = std::find_if(v.begin(), v.end(),
                          [soglia](const long& i) {
                            return i > soglia;
                          });
  // ... usa iter
}
```

In questo caso, la definizione della lambda è equivalente ad una classe nella quale le variabili catturate sono memorizzate in dati membro, inizializzati in fase di costruzione dell'oggetto funzione:

```
struct Nome_Univoco {
 long soglia;
    Nome_Univoco(long s) : soglia(s) { }
    bool operator()(const long& i) const { return i > soglia; }
};

auto iter = std::find_if(v.begin(), v.end(), Nome_Univoco(soglia));
```

Si possono catturare più variabili, separate da virgole. La notazione `[soglia]` è equivalente alla notazione `[=soglia]` e indica una cattura per valore; se invece si utilizza la notazione `[&soglia]` si effettua una cattura per riferimento (utile quando si vogliono evitare copie costose). Nella lista delle catture è possibile indicare `this`, catturando così (per valore) il puntatore implicito all'oggetto corrente; la sintassi è ammessa se la lambda è definita all'interno di un metodo (non-statico) di una classe, dove è effettivamente disponibile il puntatore this.

Esistono notazioni abbreviate per le *"catture implicite"*:

- `[=]` --> cattura (implicitamente) ogni variabile locale usata nel corpo per valore.
- `[&]` --> cattura (implicitamente) ogni variabile locale usata nel corpo per riferimento.

- `[=, &pippo, &pluto]` --> cattura (implicitamente) per valore, tranne pippo e pluto che sono catturate per riferimento.
- `[&, =pippo, =pluto]` --> cattura (implicitamente) per riferimento, tranne pippo e pluto che sono catturate per valore.

Il consiglio è di effettuare sempre catture esplicite, per maggiore leggibilità del codice.

> Si noti che il metodo `operator()` definito nella classe è qualificato `const`; di conseguenza, le variabili catturate possono essere accedute in sola lettura. Se si vuole consentirne la modifica, occorre aggiungere alla lambda il modificatore `mutable`.

Torna all'indice

Esempio

Modifichiamo la lambda dell'esempio per tenere traccia del numero di sue invocazioni.

```
void foo(const std::vector& v) {
 long num_chiamate = 0;
 auto iter = std::find_if(v.begin(), v.end(),
                          [&num_chiamate](const long& i) mutable {
                            ++num_chiamate;
                            return i % 2 == 0;
                          });
std::cout << "Funzione lambda invocata " << num_chiamate << " volte\n";
}
```

Il modificatore `mutable` viene associato a tutti i dati membro catturati: esso consente anche ad un metodo marcato const di accedere in scrittura al dato membro.

> Si noti che abbiamo catturato per riferimento, perché vogliamo che venga modificata proprio la variabile locale della funzione `foo` (non una sua copia).

La classe che viene generata implicitamente è quindi simile alla seguente:

```
struct Nome_Univoco {
 mutable long& num_chiamate;
    Nome_Univoco(long& nc) : num_chiamate(nc) { }
    bool operator()(const long& i) const {
     ++num_chiamate;
     return i % 2 == 0;
    }
};
```

Nel caso vengano effettuate catture per riferimento, occorre prestare attenzione a NON usare la funzione lambda dopo che il tempo di vita della variabile catturata è terminato (si incorrerebbe in undefined behavior).

> NOTA: come detto, l'espressione lambda crea un oggetto funzione anonimo di tipo anonimo. E' comunque possibile dare un nome all'oggetto lambda, anche se non se ne conosce il tipo, sfruttando `auto` per effettuare la deduzione del tipo.

Nell'esempio seguente, diamo un nome alla lambda per poterla usare più volte:

```
void copia_corte(const std::vector& v,
                 const std::list& l,
                 unsigned max_size) {
 auto corta = [max_size](const std::string& s) {
                  return s.size() <= max_size;
              };
 std::ostream_iterator out(std::cout, "\n");
 out = std::copy_if(v.begin(), v.end(), out, corta);
 out = std::copy_if(l.begin(), l.end(), out, corta);
}
```

Torna all'indice