24 - Callable

Concetto

Molti algoritmi generici disponibili dalla libreria standard sono forniti in due differenti versioni: la seconda è parametrizzata rispetto a una *policy*.
es. `std::adjacent_find` che ricerca all'interno di una sequenza la prima occorrenza di due elementi adiacenti ed equivalenti
due dichiarazioni:

```
template <typename FwdIter>
FwdIter adjacent_find(FwdIter first, FwdIter last) {
    if ( first == last )
        return last;
    FwdIter next = first;
    ++ next;
    while ( next != last ) {
        if (* first == * next ) // test di equivalenza
            return first;
        first = next;
        ++ next;
    }
    return last;
}
```

il predicato binario utilizzato per il controllo di equivalenza degli elementi è l' `operator==`

```
template <typename FwdIter, typename BinPred>
FwdIter adjacent_find(FwdIter first, FwdIter last, BinPred pred) {
    if ( first == last )
        return last;
    FwdIter next = first;
    ++ next;
    while ( next != last ) {
        if ( pred (* first, * next )) // test di equivalenza
            return first;
        first = next;
        ++ next;
    }
    return last;
}
```

consente di utilizzare un qualunque tipo di dato, a condizione che questo si comporti come predicato binario definito sugli elementi della sequenza.

Un **callable** è qualsiasi cosa che può essere invocata con la sintassi di una funzione:
`fun(arg1, arg2, ...);`
In particolare, include:

- **puntatori a funzione**: un puntatore a funzione è il modo più semplice e diretto per definire un callable.

```
bool pari(int i) {
    return i % 2 == 0;
}

int main() {
    bool (*fun_ptr)(int) = &pari;  // Puntatore alla funzione "pari"
    std::cout << fun_ptr(4);       // Invocazione attraverso il puntatore
}
```

Il nome della funzione stessa può essere usato direttamente come parametro dove è richiesto un callable:

```
std::find_if(v.begin(), v.end(), pari);
```

- **oggetti funzione**: sono oggetti che possono essere usati come callable grazie alla definizione dell'operatore `operator()` all'interno della classe.

```
struct Pari {
    bool operator()(int i) const {  // L'oggetto Pari è "callable"
        return i % 2 == 0;
    }
};

int main() {
    Pari pari;
    if (pari(4)) {  // Usato come una funzione
        std::cout << "Numero pari!";
    }
}
```

Vantaggi:

1. *ottimizzazione*: il compilatore può ottimizzare meglio le chiamate agli oggetti funzione rispetto ai puntatori a funzione. Per esempio, può espandere il codice in modo "inline".
2. *personalizzazione*: gli oggetti funzione possono avere stato (dati membro) e comportamento personalizzato, mentre una funzione normale no.
   Esempio di stato in un oggetto funzione:

```
struct MaggioreDi {
    int soglia;
    MaggioreDi(int s) : soglia(s) {}

    bool operator()(int i) const {
        return i > soglia;
    }
};

int main() {
    MaggioreDi maggiore_di_10(10);
```

```
    std::cout << maggiore_di_10(15);  // Restituisce true
}
```

- espressioni lambda (c++11):

Espressioni Lambda

Capita spesso che una funzione debba essere fornita come callable ad una singola invocazione di un algoritmo generico, in questi casi fornire la definizione presenta svantaggi come: inventare un nome appropriato e fornire la definizione in un punto diverso del codice rispetto all'unico punto di uso, perciò ricorriamo alle funzioni lambda.
Le **lambda expressions** (o funzioni lambda) sono una sintassi comoda per definire oggetti funzione *anonimi* direttamente nel punto in cui sono necessari.

```
auto lambda = [](int i) { return i % 2 == 0; };
std::cout << lambda(4);  // Stampa true
```

Struttura

`[capture](parametri) -> tipo_di_ritorno { corpo };`

- `[]` : capture list, definisce quali variabili locali possono essere "catturate" dalla lambda;
- `(parametri)` : lista dei parametri della funzione (opzionale);
- `-> tipo_di_ritorno` : specifica il tipo di ritorno (opzionale, può essere dedotto automaticamente);
- `{ corpo };` : il corpo della funzione

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

la lista delle catture è vuota, il tipo di ritorno è omesso in quanto dedotto dal return.
E' possibile specificarlo con il **trailing return type**:
`[]( const long & i ) -> bool { return i % 2 == 0; };`
L'uso dell'espressione lambda all'interno di `std::find_if` corrisponde all'esecuzione di queste operazioni:

1. definizione di una classe anonima per oggetti funzione
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 anonimo, avente il tipo della classe anonima, da passare alla invocazione. Ovvero:

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

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

Lista delle Catture

Può essere usata quando l'espressione deve poter accedere a variabili locali visibili nel punto in cui viene creata (diverso dal punto in cui verrà invocata):

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

è equivalente ad una classe nella quale le variabili catturate sono memorizzate in dati membro.

Catture di variabili

Quando la lambda usa variabili locali definite al di fuori di essa, queste devono essere catturate nella **capture list** (`[]`):

- *cattura per valore* (`=`): le variabili sono copiate all'interno della lambda;
- *cattura per riferimento* (`&`): la lambda accede direttamente alle variabili originali.

```
int soglia = 10;
auto lambda = [soglia](int i) { return i > soglia; }; //cattura per valore
auto lambda_ref = [&soglia](int i) { return i > soglia; }; //cattura per riferimento
```

Catture implicite

- `[=]` : cattura per valore ogni variabile locale usata nel corpo
- `[&]` : cattura per riferimento **ogni variabile locale usata nel corpo**
- `[=, &pippo]` : cattura per valore, tranne pippo catturato per riferimento
  preferire le catture esplicite.

```
int soglia = 10;
int moltiplicatore = 2;
auto lambda = [=](int i) { return i > soglia * moltiplicatore; };
```

Funzioni Lambda con nome

E' possibile dare un nome alle funzioni lambda con `auto` . Esempio: diamo un nome all'oggetto lambda per poterlo utilizzare più volte.

```
void copia_corte ( const std::vector<std::string>& v,
                   const std::list<std::string>& l,
                   unsigned max_size ) {
    auto corta = [max_size]( const std::string& s ) {
```

```
                    return s . size () <= max_size;
                };
    std::ostream_iterator<std::string> out(std::cout, "\n");
    out = std::copy_if(v.begin(), v.end(), out, corta);
    out = std::copy_if(l.begin(), l.end(), out, corta);
}
```

//////////////////////////

```
template <typename Iter, typename UnaryPred>
Iter find_if(Iter first, Iter last, UnaryPred pred) {
    for ( ; first != last; ++first) {
        if (pred(*first))
            return first;
    }
    return last;
}

bool pari (long i) { return i % 2 == 0; }

struct Pari{
    //...
};
void foo(const std::vector<long>& vl) {

    auto it = vl.begin();

    for ( ; it != vl.end(); ++it) {
        if (*it % 2 == 0)
            break;
    }

    auto it = std::find_if(vl.begin, vl.end(), pari); //chiama la funzione bool pari
(...)
    auto it = std::find_if(vl.begin(), vl.end(), Pari{}); //chiama la classe struct
Pari{...}
    auto it = std::find_if(vl.begin(), vl.end(),
                [](long i) { return i % 2 == 0; });

    std::cout<< *it;

    for (const auto& i : vl) { //for range loop, zucchero sintattico, itera su tutti
gli elementi contenuti in vl
        if (i % 2 == 0) {
            std::cout << i;
            break;
        }
    }
    //il for range loop funziona anche sugli array
    long al[1000];

    for (const auto& i : al) {
        if (i % 2 == 0) {
```

```
        std::cout << i;
        break;
        }
    } //funziona anche senza le funzioni .begin() e .end(), perche tra i vari algoritmi forniti dalla libreria standard, ci sono anche alcuni funzioni esterne ai contenitori (e. g. vector) che danno la possibilità di prendere l'inizio della sequenza memorizzata all'interno del contenitore:
}

//versione base, begin templatica, prende un typename Cont (contenitore) se passano un contenitore per riferimento (Cont& c)
template <typename Cont>
auto begin(Cont& c) -> decltype(c.begin()) { //decltype operatore che prende un'espressione e torna il tipo di quella espressione
    return c.begin();
}

//stessa cosa per il template .end()
template <typename Cont>
auto end(Cont& c) -> decltype(c.end()) {
    return c.end();
}

//dal momento che gli array non sono classi, mettiamo in overloading un altro template di funzione pensato per funzionare con gli array di n elementi di tipo T
template <typename Cont, std::size_t N>
T* begin( T(&array)[N]){ //array di tipo T, lungo esattamente N, passato per riferimento per evitare il typedecay (arriva il puntatore al primo elemento e non sa più quant'è grande l'array)
    return array;
}
//funzione begin parametrica su due argomenti, il primo è il tipo degli elementi contenuti nell'array, la seconda è una costante nota a tempo di compilazione; restituisce un puntatore ad intero

template <typename Cont, std::size_t N>
T* end( T(&array)[N]) {
    return array + N;
} //puntatore a uno dopo l'ultima posizione dell'array
```