Funzioni e operatori

Programmazione di Sistema

A.A. 2017-18

Argomenti

- Puntatori a funzione
- Oggetti funzionali
- Funzioni lambda
- Overloading degli operatori

Puntatori a funzione

- In C++, come in C, è lecito salvare in una variabile il puntatore ad una funzione
  - Potrà essere utilizzato in seguito, permettendo di costruire programmi il cui comportamento cambia dinamicamente
- Per dichiarare una variabile di tipo puntatore a funzione si usa la sintassi
  - <tipo_ritornato> (* var) (<argomenti>)

Puntatori a funzione

- Si assegna un valore attraverso l'operatore =
  - int f(int i, double d) {
    /* corpo della funzione */
    };
  - int (***var**)(int, double);
  - **var** = f;
  - **var** = &f; //identico al precedente
- Si invoca la funzione il cui puntatore è contenuto nella variabile con la sintassi
  - **var**(10, 3.1415926);
- Sia il tipo di ritorno che tutti i parametri formali della funzione devono corrispondere a quanto dichiarato nella definizione della variabile

Puntatori a funzione

```
int f (int, int);
int f (int, double);
int g (int, int = 4);
double h (int);
int i (int);

int (*p) (int) = &g;        // ERRORE: Manca il
parametro
                        // di default
p = &h;                     // ERRORE: tipo di
ritorno                             //differente
p = &i;                     // OK
p = i;                      // OK

int (*p2) (int, double);
p2 = f;                     // OK: il compilatore
sceglie
                        //"int f (int, double)"
```

Oggetti funzionali

- In C++ esiste un ulteriore tipo invocabile:
  il «funtore» o «oggetto funzionale»
  - Istanza di una qualsiasi classe che abbia ridefinito la funzione membro **operator ()**

```
class FC {
public:
  int operator() (int v) {
    return v*2;
  }
};
```

```
{
  FC fc;
  int i= fc(5);
  // i vale 10
  i=fc(2);
  // i vale 4
}
```


Oggetti funzionali

- È possibile includere più definizioni di **operator ()**
  - Devono avere tipi differenti nell'elenco dei parametri
- Un oggetto funzionale può contenere variabili membro
  - Queste possono essere utilizzate all'interno delle funzioni operator() per tenere traccia di uno stato
- Il comportamento non è più quello di una funzione matematica (il cui output è sempre lo stesso a parità di input)
  - L'oggetto viene detto «funzionale»

Oggetti funzionali

```
class Accumulatore {
  int totale;
public:
  Accumulatore():totale(0){}
  int operator()(int v){
    totale+=v;
    return v;
  }
 int totale() { return totale;}
};

void main() {
  Accumulatore a;
  for (int i=0; i<10; i++)
    a(i);
  std::cout<<"Totale:"<<a.totale()<<std::endl;
  //stampa 45
}
```

Mescolare i due approcci

- Utilizzando la programmazione generica, è possibile scrivere funzioni che accettano indifferentemente come parametri
  - puntatori a funzione
  - oggetti funzionali
- Questo permette di aumentare il livello di generalizzazione dei propri programmi
  - Approccio largamente utilizzato nella libreria standard
- L'utilizzo di oggetti funzionali è spesso molto utile
  - C++11 ha introdotto una notazione particolare per definirlo in modo compatto: le funzioni lambda

Mescolare i due approcci

```
template <typename F>
void some_function(F& f) {
  //...
  f();  // f può essere un
funzionale o
        // un puntatore a funzione
  //...
}
```


Funzioni minimali

- Spesso, l'utilizzo degli algoritmi presenti nella libreria standard richiede l'introduzione di funzioni usate una volta sola

```
void print(int i) {
  std::cout<< i << " ";
}

int main() {
  std::vector<int> v;
  //...
  std::for_each(v.begin(), v.end(),
print);
}
```


Usare una funzione lambda

- Lo stesso comportamento può essere ottenuto usando una funzione lambda
  - Evitando di definire una funzione esplicita e doverle dare un nome che potrebbe collidere con altri nomi

```
int main() {
  std::vector<int> v;
  //...
  std::for_each(v.begin(), v.end(),
    [](int i) { std::cout << i << "
"; }
  );
}
```

Restituire un valore

- Se il suo corpo contiene una sola istruzione return
  - Il tipo ritornato può essere dedotto automaticamente dal compilatore
  - Altrimenti occorre esplicitarlo nella definizione

```
[](int num, int den) -> double {
  if (den==0)
     return std::NaN;
  else
     return (double)num/den;
}
```


Catturare delle variabili

- Le parentesi quadre "[ ]" introducono la notazione lambda
  - Al loro interno è possibile elencare variabili locali il cui valore o il cui riferimento si vuole rendere disponibili nella funzione
- [x, y] – "x" e "y" sono catturate per valore
  - Viene effettuata una copia
  - La funzione λ potrà essere invocata anche quando tali variabili saranno uscite dallo scope
- [&x, &y] - "x" e "y" sono catturate per riferimento
  - Eventuali cambiamenti influenzano l'originale
  - Attenzione a riferimenti pendenti!
- [&] – cattura «tutto» per riferimento
- [x, &y] – cattura "x" per valore e "y" per riferimento

Funzioni lambda: sintesi

- Sintassi
  - [<captured_vars>] (<params>) -> <return_type> { <function body> }
- Supportano diversi stili di programmazione
  - Algoritmi generici (funzione di confronto per sort)
  - Programmazione funzionale
  - Programmazione concorrente
- Migliorano la leggibilità di un programma
  - Permettendo di eliminare classi/funzioni piccole
  - ...a patto che si capisca cosa significano!

Funzioni lambda:

- Vantaggi
  - Aumentano la leggibilità
    - Il corpo appare dove viene usato
  - Aumentano l'espressività
    - Rendono più chiare le intenzioni del programmatore
  - Aumentano la compattezza del codice
- Svantaggi
  - Sintassi criptica
    - Per chi non conosce la notazione

Operator overloading

- Tutti gli operatori in C++(+,-,%,=, +=...) possono essere ridefiniti per l'uso con tipi di dati definiti dal programmatore
- Non è possibile definire nuovi operatori
- Non è possibile cambiare le precedenze
- Possono essere facilmente utilizzati in modo errato

Operator overloading

```
class Frazione {
    public:
        Frazione(int num = 0, int den = 1)
        {
            this->num = num;
            this->den = den;
        }
Frazione operator + (const Frazione &r)
        {
            Frazione   temp;
            temp.den = this->den * r.den;
            temp.num = r.den * this->num +
                this->den * r.num;
            return temp;
        }
private:
        int num;
        int den;
    };
```


Operator overloading

- La classe Frazione contiene due variabili intere private
  - numeratore (num) e denominatore (den)
- L'operatore "+ " viene ridefinito in modo da sommare due istanze della classe Frazione
- L'operatore ritorna la copia di un oggetto di tipo Frazione

Operator overloading

```
{
...
Frazione a(1,3);
Frazione b(3,5);
Frazione c;
c = a + b;
...
}
```

```
Frazione temp;
temp.den = this->den *b.den;
temp.num =
        b.den * this->num +
        this->den * b.num;
return temp;
```

Spunti di approfondimento

- Completare la classe Frazione aggiungendo le definizioni per gli operatori di sottrazione, moltiplicazione e divisione