Sommario

Lezione 1

L’evoluzione del C++.

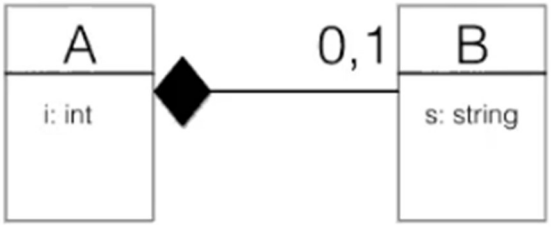
Il C++ nasce come C con le classi nel 1979.

All’inizio appunto le uniche differenze dal C sono classi, costruttori, distruttori, ereditarietà e astrazione.

Un punto fondamentale della storia del C++ è la standardizzazione (che segna l’ingresso ufficiale nel mondo delle aziende) che avviene alla fine degli anni 80.

Ripasso dei class diagram

Composizione opzionale



Il rombo colorato indica composizione opzionale, ovvero B segue il destino di A (se A viene eliminato anche B viene eliminato). Notta che A può avere 0 B o 1 B collegati.

Note sul codice:

Poniamo di avere una classe B, si può usare il costruttore a 0 parametri in questo modo *B b;* però attenzione: se si crea il costruttore a 1 parametro quello di default a 0 viene eliminato.

Il resto della lezione è una descrizione del codice presente. Tale codice va a parlare di vari argomenti come costruttori diversi (tra cui costruttore di copia), prassi legate a questi metodi, gestione della memoria e correlazione tra varie classi.

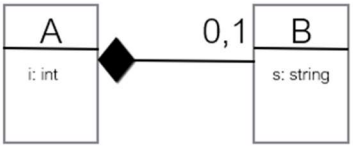
Lezione 2

Metodi ed operatori in una classe vuota

Quali sono i metodi che vengono creati automaticamente in ogni classe (anche vuota)?

Il costruttore a 0 parametri, il costruttore di copia e il distruttore.

Composizione opzionale:



Soluzione con puntatore a B in A, la classe A ha la responsabilità di gestire la memoria in cui si trova l'oggetto di tipo B

Si ottiene tramite l'inserimento di un puntatore a B in A. Questo implica che la classe A si deve anche accollare la gestione della memoria relativa all'istanza di B.

A questo punto ci siamo posti il problema della copia superficiale o profonda.

Proviamo a rivedere i passaggi.

Aggiungiamo alla classe vuota che abbiamo usato prima come esempio un attributo *int i*.

Chiaramente, se non andiamo a toccare il costruttore di copia, tutte le copie verranno create superficialmente.

Questo succede anche se andiamo a dichiarare una variabile di tipo puntatore.

Il problema sorge esattamente quando si va a copiare un puntatore che punta ad una variabile che è stata ottenuta da una new e la cui responsabilità spetta alla classe A.

Osserviamo questo esempio:

```
class A {int i; B* pb;};
main()
{
A a;// A::A()
…// codice che inizializza a.pb=new B
A a2(a); //A::A(const A&)
//Quanti B ci sono?
A a3;
a3=a2; //A& A::operator=(const A& a)
//Quanti B ci sono?
A *pa=new();
…(*pa)=a;
delete pa; A::~A()
};
```

Se dopo la deallocazione di pa si va a cercare pb in una qualche istanza copiata da a, naturalmente si va ad incorrere in problemi brutti.

La soluzione in questo caso è ridefinire il costruttore di copia (e l'operatore di assegnazione).

Nota che il costruttore di copia viene invocato anche ogni volta che un'istanza della classe viene passata per valore (si dice passaggio per valore/copia), quindi in realtà è usato molto più spesso di quanto non pensiamo.

Costruttori con parametri

Abbiamo visto come il costruttore senza parametri di default venga creato solo se non sono presenti altri costruttori.

I costruttori con un solo parametro hanno però in C++ un significato speciale perché vengono usati come convertitori impliciti di tipo.

Vediamo quindi il seguente esempio:

- Supponiamo di avere una

```
class A{int i;

public: A(int _i);};
```

- E di avere anche un funzione `void f(A);`
- Se la funzione f fosse chiamata con `f(3)` il sistema di deduzione dei tipi del C++ la compilerebbe come se fosse `f(A::A(3))` o equivalentemente `f(A(3))`.
- Come caso particolare questo avviene anche in casi come questo

```
A a;

a=3; // diventa A& A::operator=(A(3));
```

Nota come la conversione implicita di tipo venga utilizzata anche in occasione delle assegnazioni.

Tutto questo vale fino a quando non dichiariamo il costruttore come **explicit**, ovvero rendiamo il costruttore utilizzabile solo in maniera esplicita (solo se viene esplicitamente invocato dal programmatore).

Nota che questa strategia può essere utilizzata dal programmatore che non vuole rischiare di incorrere in problemi di dichiarazione implicita a run-time.

Overloading (ridefinizione) degli operatori

Gli operatori sono le operazioni che appaiono nelle espressioni.

Si distinguono principalemente per il numero di argomenti che prendono.

Non è possibile inserire nuovi operatori, gli operatori sono definiti nella definizione del linguaggio di programmazione.

Però gli operatori si possono *ridefinire*, o come metodi di una classe o come funzioni esterne (non entrambi!)

La differenza tra queste strategie è:

- Quando vengono definiti come funzioni esterne, tutti gli operatori vanno passati come parametri della funzione

- Quando vengono ridefiniti come metodi, il primo operando corrisponde all'istanza chiamante del metodo stesso {che viene riferito con il *this* all'interno del metodo}

L'operatore = può essere ridefinito solo come metodo. L'operatore di assegnazione ritorna una reference!

Questo a differenza di molti altri operatori tipo + che ritorna un valore (per valore).

Esempio di chiamata degli operatori: consideriamo operator+ e vogliamo scrivere

A a, a1, a2;

a=a1+a2;

Ridefinito operator+ come metodo a=a1+a2; corrisponde alla chiamata A::operator=(a1.operator+(a2));

Ribadiamo che operator= può essere ridefinito SOLO come metodo.

Vediamo un esempio di operator override come metodo

- Vediamo ora l'implementazione di operator+ come metodo

```
class A{ int i;
public: A(int _i){i=_i;}
A operator+(const A&);
};
A A::operator+(const A& _a)
{
return A(i+_a.i);
}
```

- Ma:

```
A a1(1), a2(2), a3(3);
a1=a2+a3; //ok
a1=2+a3; // non compila
a1=a2+3; //a1.operator=(a2.operator+(A(3))); compila grazie alla conversione implicita di tipo
```

Nota interessante: osserva come nel secondo caso il primo operando è un'istanza di intero, il compilatore non trova una traduzione per l'operatore + che prende un A e lo trasforma in intero.

Cosa succede invece se facciamo l'override tramite funzione?

- Vediamo ora l'implementazione di operator+ come funzione

```
class A{ int i;
public: A(int _i){i=_i;} /corretto
friend A operator+(const A&,const A&);
};
A operator+(const A& _x, const A& _y) //corretto anche lo scope
{
return A(_x.i+_y.i);
}
```

- Ma:

```
A a1(1), a2(2), a3(3);
a1=a2+a3; //ok
a1=2+a3; // a1.operator=(operator+(A(2),a3)) compila grazie alla conversione implicita di tipo
a1=a2+3; //a1.operator=(operator+(a2,A(3))) compila grazie alla conversione implicita di tipo
```

La parola chiave friend

Questa parola chiave fa si che la funzione rimanga esterna (non sia un metodo della classe) ma abbia la possibilità di accedere alle variabili private della classe.

Nota che non è strettamente necessario utilizzare il friend se esistono metodi pubblici che permettono di fare il get del valore delle variabili private della classe.

Friend va utilizzato con parsimonia perché viola il principio di incapsulamento delle classi, spesso un abuso di friend significa cattiva progettazione.

Operatori di incremento e decremento

Gli operatori di incremento e decremento sono unari.

Possono essere ridefiniti sia come metodi che come funzione esterna.

Entrambi i metodi hanno versione prefissa e postfissa:

- Nella modalità prefissa la variabile prima viene modificata e poi restituita
- Nella modalità postfissa viene ritornato il vecchio valore della variabile

Ma come facciamo a distinguere tra operatore prefisso e postfisso?

Il C++ utilizza un argomento dummy come indicatore della tipologia dell'operatore.

Entrambi possono essere modificati in C++, per distinguere viene usato nella versione postfissa aggiungendo un parametro int “dummy”, quindi nel caso di implementazione come metodi si ha che gli operatori vanno dichiarati come:

A& A::operator++(); //prefisso

A A::operator++(int);//postfisso

Nota anche come il prefisso ritorni per referenza, mentre il postfisso ritorna un valore.

Il postfisso ritorna un valore perché idealmente la variabile deve venire incrementata alla fine dell’operazione in corso, quindi deve rimanere costante mentre l’operazione è svolta: non ha senso ritornare una referenza modificabile.

Operatore postfisso

```
Class A {int i;
 A A::operator++(int);//postfisso
};
A A::operator++(int) {
A temp(*this); //corretto
i++;
return temp;
}
```

- Cosa succede se si scrivono cose come A a; (a++)++;?

Succede che ho un errore, perché non posso richiamare ++ su un rvalue.

Quindi fai attenzione! C’è un’importante distinzione sui valori di ritorno!!!

Ci sono operatori che ritornano il risultato per valore e operatori che ritornano il risultato per referenza.

Come nel caso degli incrementi o decrementi prefissi o postfissi la distinzione si ha quando il risultato è utilizzato successivamente nella stessa espressione.

Gli operatori che modificano l’oggetto chiamante ritornando una reference.

Esempio: operator+= e operator+ il primo ritornerà A& mentre il secondo A.

Nota importante: il C++ permette di ridefinire gli operatori anche fra tipi disomogenei.

“molto spesso il C++ permette di fare molte più cose di quelle che è sensato fare”

Elenco degli operatori

Quali operatori sono presenti in C++?

Pagina Wikipedia con tutti gli operatori di C++.

Quando è utile ridefinire gli operatori?

Nella programmazione generica ci sono librerie che assumono l'esistenza di operatori. Esempio: il std::set<A> richiede che sia ridefinito *operator<* .

Quando il significato degli operatori è "naturale" (per esempio in una classe per supportare l'uso di numeri complessi link) e questo aiuta la comprensibilità.

Altrimenti vanno usati con parsimonia e cautela.

Lezione 3

Oggi parliamo di ereditarietà.

Esistono 3 tipi di ereditarietà: public, private e protected. Il tipo di ereditarietà di una classe influisce poi sugli attributi a cui le classi figlie potranno accedere. Se la classe B eredita da A ecco le 3 possibili modalità:

- **Public** gli attributi public in A saranno public in B, i protected in A saranno protected in B
- **Protected** tutti gli attributi che B eredita da A sono protected in B
- **Private** tutti gli attributi che B eredita da A saranno private in B

Un'istanza classe derivata, se derivata in modo pubblico, può essere acceduta sia come istanza della classe derivata sia come istanza della classe base, in pratica costruisce una relazione IS-A di tipo.

In particolare, i puntatori della classe base possono essere usati per puntare ad un'istanza della classe derivata.

Keyword virtual

In una classe possono essere dichiarati dei metodi virtuali, ciò rende possibile le differenze tra early e late binding (ovvero ricerca di un metodo in una classe al momento della compilazione [early] o al momento della esecuzione [late]). In particolare, il qualificatore virtual specifica che un certo metodo potrebbe essere sovrascritto e quindi indica la necessità a runtime di andare a cercare late bindings nelle classi figlie.

Nota però che in C++ il late-binding si può avere solo se si accede tramite puntatore o reference.

In seguito, abbiamo osservato le classei A e B in cui abbiamo sperimentato un po' con i metodi ereditati.

Classe A

```
#ifndef CLASSE_A
#define CLASSE_A

#include<iostream>
using namespace std;
#include<string>

class A{
      int i;
      public:
      A();
      explicit A(int _i);
      A(const A& _a);//non necessario ora
      //A& operator=(const A& _a);//non necessario ora
      virtual ~A();// non necessario, ma necessario se si accede
      //alle istanze derivate con puntatori alla classe base
      int get_i()const;
      };

void test_A();
int quadrato(const A& _a); //funzione che ha un A& come parametro
int raddoppia(A _a);//passato per valore
#endif
```

```
#include "a.h"

A::A(){i=0;cout<<"A::A()"<<this<<endl;}

A::A(int _i){i=_i;cout<<"A::A(int)"<<this<<endl;}

A::A(const A& _a){i=_a.i;cout<<"A::A(const A&)"<<this<<endl;}

A::~A(){cout<<"A::~A()"<<this<<endl;}

int A::get_i()const {return i;}

void test_A(){
              A a; cout<<a.get_i();
              A a3(3); cout<<a3.get_i();
              cout<<quadrato(a3);
              cout<<raddoppia(a3);
              };

int quadrato(const A& _a){return _a.get_i()*_a.get_i();}
int raddoppia(A _a){return 2*_a.get_i();}
```

Classe B

```
#ifndef CLASSE_B
#define CLASSE_B
#include<iostream>
using  namespace std;
#include<string>

#include "a.h"

class B:public A{
      string s;
      public:
      B();
      B(int _i,string _s);
      ~B();// non e' necessario
      string get_s();
      };

void test_B();
#endif
```

```
#include "b.h"

B::B(){s="";cout<<"B::B()"<<this<<endl;}

B::B(int _i,string _s):A(_i){s=_s;cout<<"B::B(int,string)"<<this<<endl;}

B::~B(){cout<<"B::~B()"<<this<<endl;}

string B::get_s(){return s;}

void test_B()
     {B b; cout<<b.get_s()<<b.get_i();
      B b2(3,"Moli"); cout<<b2.get_s()<<b2.get_i();
      cout<<quadrato(b2);// b e' anche un A posso chiamare la funzione
      cout<<raddoppia(b2);
     }
```

Codice del main:

```
#include "a.h"
#include "b.h"
int main(int argc, char *argv[])
{   test_A();
    cout<<endl;
    test_B();
    cout<<"main:"<<endl;
    B* pb=new B(7,"note");
    cout<<quadrato(*pb);
    delete pb;
    cout<<"puntatore ad A:"<<endl;
    A* pa=new A(11);
    cout<<quadrato(*pa);
    delete pa;
    cout<<"puntatore ad A che punta ad un B:"<<endl;
    pa=new B(13,"giorni");
    cout<<quadrato(*pa);
    delete pa;

    A a;
    B b;
    a=b; // 1 12
    //b=a; non compila
    cout<<"assegnamento ad a"<<endl;
    //a=3;// viene chiamato A::A(int) usato come convertitore implicito di tipo
    a=A(3);

    A a2(b); // cosa sto chiamando?

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Vediamo che possiamo passare al metodo quadrato una variabile di tipo B, questo si può fare appunto perché B IS-A A;

Se runniamo il codice ci accorgiamo anche di come si possa dichiarare il distruttore come virtuale, quindi il distruttore stesso può essere sovrascritto dal distruttore delle classi ereditarie.

La stessa cosa non vale per i costruttori, i costruttori non vengono ereditati, quindi non possono essere definiti come virtuali.

Metodi e classi puramente virtuali

In una classe se io dichiaro un metodo virtual in questo modo

```
virtual void stampa()const =0;
```

rendo la classe non utilizzabile (classe puramente virtuale), che funziona solo come base per costruire altre classi per derivazione; in pratica le classi puramente virtuali in C++ funzionano come le interfacce in Java.

Lezione 4

Programmazione generica

Con programmazione generica si intende la possibilità data da un linguaggio di rappresentare tipi e implementare algoritmi che abbiano un tipo come parametro.

Il tipo parametrico viene poi specificato:

- Tempo di compilazione (es. templates in C++)
- Tempo di esecuzione (es generics in Java)

Lo scopo della programmazione generica è implementare algoritmi che siano indipendenti dal tipo su cui operano ("generici" appunto), idealmente al massimo livello di astrazione possibile.

Per capire cosa si intende osserviamo il meccanismo della programmazione generica applicato al problema dello swapping di variabili.

Il modo più semplice per implementare una sorta di programmazione generica (funzione astratta che si applica a multipli tipi di variabili) è l'utilizzo del meccanismo di overloading, ovvero creare più funzioni con lo stesso nome che prendono parametri di tipi diversi:

```
void my_swap (int &f, int &s ) {
int tmp = f; f=s; s=tmp;
}
void my_swap (string &f, string &s ) {
string tmp = f; f=s; s=tmp;
}
```

In questo caso, ad esempio, definiamo la funzione my_swap per due interi e per due stringhe, quello che facciamo è un overload del la funzione my_swap.

Questo metodo non assolutamente astratto ma in casi molto ristretti e semplici può essere comunque una buona soluzione.

Un secondo modo è quello di utilizzare il puntatore a void. Questo è un puntatore *crudo* che può contenere indirizzi di qualsiasi cosa.

```
void my_swap (void* &f, void* &s ) {
void* tmp = f;
f=s;
s=tmp;
}
```

In questo modo posso tranquillamente scambiare il valore delle due variabili f ed s. Questo metodo però ha un problema bello grosso, le due variabili che passo come parametri della funzione devono essere dichiarate come void*, con tutti i problemi che questo comporta in seguito: un void* può essere castato a qualsiasi tipo e allora si che si fa sagra e non si capisce più nulla.

Tuttavia, questo metodo ha anche un bel vantaggio: lo swap viene effettuato solo sui puntatori, quindi non si muove un byte in memoria, non si fanno copie né null''altro di resource-consuming.

Una variante del metodo void* è lo static_casting che permette di scoprire a tempo di compilazione errori di casting: è un casting scaltro.

Prima di approfondire il discorso sui template andiamo ad affrontare lo swap con il meccanismo delle classi virtuali.

Di fatto si può nascondere la diversità delle classi derivate dentro a dei metodi virtuali è possibile scrivere delle funzioni generiche che prendono in ingresso riferimenti alla classe base ed in seguito all'interno della funzione sfruttare l'ereditarietà dei metodi virtuali per far funzionare l'operatore =.

```
class A{
public:
virtual A& operator=(const A& _a)=0;
virtual A* clone()const=0;
virtual ~A(){};
};


#include "A.h"

class B:public A{
int i;
public:
B(int _i){i=_i;};
B& operator=(const A& _b);
B& operator=(const B& _b);
B* clone()const;
friend ostream& operator<<(ostream& os,const B& _b);
};

ostream& operator<<(ostream& os,const B& _b);
```

In questo esempio la classe A ha come metodo puramente virtuale l'operatore "=". La classe B che eredita dalla classe A ha due operatori "=", uno con argomento di tipo A& e l'altro con argomento di tipo B&.

```
B& B::operator=(const A& _b){
  A* temp=_b.clone();
  B b(*((B*)temp));
  *this=b;
  return *this;
}

B& B::operator=(const B& _b){
  i=_b.i;
  return *this;
}

B* B::clone()const{ return new B(*this);};

ostream& operator<<(ostream& os,const B& _b){return os<<_b.i;}
```

Nell'implementazione della classe B osserviamo l'interessante caso della definizione di operator=(const A& _b).

In primis viene creato un colne dell'argomento _b e viene assegnato ad un puntatore A*, poi posso fare il casting a B* di temp (ma solo perché A è completamente virtuale, quindi non possono esistere istanze di A ed ho la certezza che _b è di tipo B&, nota che questa è una cazzata e l'esempio, anche a detta di Blanzieri, non è

robusto) comunque sostanzialmente la cosa da notare qui è che con qualche giro articolato si può utilizzare l'ereditarietà virtuale come meccanismo per l'inserimento di genericità nelle funzioni.

Passiamo finalmente a parlare di generics e template.

Il metodo generic invece funziona utilizzando un template:

```
template <class T>
void my_swap(T& f, T& s) {
T tmp = f;
f = s;
s = tmp;
}
```

In qualsiasi modo, il funzionamento del templating è presto detto: la funzione my_swap fa proprio quello che ci si aspetta da una funzione che accetta parametri di tipi vari e variegati, ovvero se ne frega del tipo delle variabili e prova a far funzionare le cose lo stesso.

La differenza tra la genericità virtuale e la genericità tramite template è che la prima funziona come in java a runtime (in Java la genericità viene implementata gerarchicamente, dato che tutto deriva da Object); mentre la genericità tramite template viene verificata a compile time.

Guardiamo qualche esempio dell'utilizzo dei template:

```
template <typename T>
T min (T a, T b) {
return a < b ? a : b;
}
```

```
template <typename T1, typename T2>
T1 min (T1 a, T2 b) {
return a < b ? a : b;
}
```

I template possono essere anche utilizzati per generare classi, come ad esempio la classe Pair della STL.

```
template <typename F, typename S>
class Pair
{
public:
Pair(const F& f, const S& s);
F get_first() const;
S get_second() const;
private:
F first;
S second;
};
template <typename F, typename S>
Pair<F,S>::Pair(const F& f, const S& s)
{
first = f;
second = s;
};
```

Standard Template Library

La STL è essenzialmente una libreria dove sono organizzati i template stadard.

È stata creata perché così se uno la utilizza ha garanzia della portabilità del suo codice.

La STL contiene:

- Contenitori
- Iteratori
- Algoritmi

I contenitori ad esempio sono list, vector, set e map. Questi sono a tutti gli effetti "contenitori" di variabili di un qualche tipo parametrizzato. Una volta che si istanzia uno di questi contenitori lo si istanzia per un determinato tipo di variabile e da quel punto in poi si può inserire variabili di quel tipo nel contenitore e lavorarci con tutti i metodi parametrizzati offerti dal contenitore.

Gli iteratori sono classi template che riproducono in parte il funzionamento dei puntatori, di fatto fanno l'overloading di alcuni operatori con lo stesso significato che tali operatori hanno per i puntatori.

Gli iteratori servono per riportare un po' di efficienza nei contenitori, che pagano un po' in efficienza per essere generici.

Gli iteratori sono anch'essi organizzati in una gerarchia: Random Access->Bidirectionas->Forward->Input | Output

category				properties	valid expressions
all categories				*copy-constructible*, *copy-assignable* and *destructible*	`X b(a);` `b = a;`
				Can be incremented	`++a` `a++`
Random Access	Bidirectional	Forward	Input	Supports equality/inequality comparisons	`a == b` `a != b`
				Can be dereferenced as an *rvalue*	`*a` `a->m`
			Output	Can be dereferenced as an *lvalue* (only for *mutable iterator types*)	`*a = t` `*a++ = t`
				default-constructible	`X a;` `X()`
				Multi-pass: neither dereferencing nor incrementing affects dereferenceability	`{ b=a; *a++; *b; }`

				Can be decremented	`--a`
					`a--`
					`*a--`

				Supports arithmetic operators + and -	`a + n`
					`n + a`
					`a - n`
					`a - b`

				Supports inequality comparisons (<, >, <= and >=) between iterators	`a < b`
					`a > b`
					`a <= b`
					`a >= b`

				Supports compound assignment operations += and -=	`a += n` `a -= n`
				Supports offset dereference operator ([])	`a[n]`

Quelli elencati sopra sono i vari operatori disponibili per i diversi tipi di operatori.

Alcuni contenitori supportano solo alcuni iteratori, per questo abbiamo diverse classi di iteratori: una lista unidirezionale non può usare iteratori bidirezionali.

Prendiamo proprio il container list come esempio di partenza: possiamo farci tornare da una lista un iteratore tramite il metodo begin(), questo punterà il primo elemento della lista, se la lista è vuota in realtà punterà la fine della lista, che possiamo recuperare anche tramite il metodo list.end().

Questo strano gazzabuglio di metodi potrebbe essere risolto se il C++ volesse metterci una pezza, ma la filosofia del C++ è “fai tuti i controlli possibili a compile time e per il resto si arrangia il programmatore”, questo perché per C++ la cosa più importante è la velocità a runtime.

Gli algoritmi della STL sono implementati anch’essi con l’utilizzo dei generics e sono implementati nel modo più astratto possibile. Gli algoritmi qui definiti utilizzano appunto gli iteratori in maniera più generale possibile per creare metodi semplici ma sempre utili come ad esempio il for each.

Lezione 5

Oggi STL ed utilizzo dei contenitori

La volta scorsa abbiamo visto come contenitori ed ereditarietà siano la base della STL, oggi vedremo delle applicazioni.

I contenitori "contengono" altri tipi parametrizzati, diciamo che inglobano il contenuto e forniscono delle funzionalità in più, al costo di rendere indiretto l'accesso agli oggetti che contengono: vi si può accedere solo tramite iteratori.

La volta scorsa abbiamo visto i tipi di iteratori disponibili; iteratori diversi possono essere usati da contenitori diversi. L'iteratore che si ottiene per scorrere gli elementi di un contenitore dipende strettamente dal tipo del contenitore.

Contenitore lista

Come si usa la lista? Innanzitutto, si include.

Quando si istanzia si deve dichiarare il tipo di dato che racchiuderà. Dalla lista si ottiene un iteratore di tipo bidirezionale. L'iteratore end viene utilizzato come puntatore NULL a fine lista, è un segnaposto e non deve mai essere dereferenziato, esso è un riferimento esterno al contenitore!

Nota interessante: la keyword auto ci permette di semplificare l'operazione di iterazione su un contenitore: auto riconosce il tipo di una variabile a runtime.

Ci permette di fare range loop, se non è chiamata per riferimento (auto&) ci restituisce una variabile solo in lettura, se viene chiamata per riferimento invece rende la variabile modificabile.

Metodi di inserimento nella lista:

- Push_back()
- Push_front()
- Insert(iteratore, offset)
- Insert(iteratore1, iteratore2, iteratore3) [copia tutto ciò che si trova tra iteratore2 ed iteratore3 appena dopo iteratore1]

La lista ci permette di ottenere anche degli iteratori non modificabili, utilizzando il metodo **c**begin() che ci ritorna un iteratore che può essere dereferenziato ma non modificato (l'elemento a cui punta non può essere modificato, io ad esempio però posso fare ++ sull'iteratore).

L'operazione di splice ci permette di dividere una lista.

Si ha una lista l1, si crea una lista l2 secondaria, su l2 si chiama splice(iteratore1, iteratore2) con iteratore 1 ed iteratore2 che sono iteratori sulla lista l1; quello che si ottiene alla fine è che la lista l2 conterrà tutti gli elementi tra iteratore1 ed iteratore2, mentre in l1 rimarranno tutti gli elementi rimasti.

Una cosa interessante del metodo splice è che non va a copiare e spostare gli elementi della lista ma va solo a magheggiare con i puntatori.

Vediamo anche i metodi reverse(), sort(), unique().

Alcuni metodi sono safe per gli iteratori, nel senso che quando vengono applicati l'iteratore punta sempre allo stesso oggetto, prima e dopo l'applicazione; ma non tutti i metodi sono così.

Il metodo sort utilizza…

"The elements are compared using `operator<` for the first version, and *comp* for the second."

Il metodo unique() agisce su una lista *sortata* ed elimina i valori ripetuti.

Il metodo merge fa l'unione di due liste. Il merge è come quello di github (l.merge(r) unisce r ad l).

Il metodo resize può essere usato per tagliare degli elementi accorciando la lista, il profe non sa cosa succede se fai un resize con un numero maggiore della lunghezza della lista.

"If *n* is greater than the current container size, the content is expanded by inserting at the end as many elements as needed to reach a size of *n*. If *val* is specified, the new elements are initialized as copies of *val*, otherwise, they are value-initialized."

Poi ci sono i metodi remove(), erease(), emplace(iteratori, parametri) che permette di inserire in una posizione qualsiasi della lista un elemento che viene inizializzato al volo con i parametri passati, l.assign(it1, it2) che copia nella lista l tutti gli elementi tra it1 e it2.

Passiamo ora ai contenitori set e map

Questi sono due contenitori "ordinati e unici", ovvero gli elementi inseriti nel contenitore sono unici e su di essi è definita un'operazione di ordinamento.

A questo punto non ha più senso definire le funzioni di front e back perché questi sono rappresentati banalmente dall'elemento minimo e dal massimo, quindi non esistono nemmeno più funzioni come push_front o push_back, ma esiste solamente la funzione insert.

Naturalmente il tipo di dato che vado ad inserire nel contenitore deve avere il metodo < definito, altrimenti non può essere ordinato e quindi la dichiarazione di un set o una map per quel tipo darà errore.

Map è interessante perché vi si può accedere come fosse un vettore, ad esempio se ma è di tipo map posso accedere a ma[2] che significa che accedo al valore che è contenuto nella mappa che corrisponde alla chiave 2: se tale valore non esiste, viene creato un valore al volo che corrisponde alla chiave 2.

Lezione 6

La settimana scorsa abbiamo visto degli esempi di utilizzo dei contenitori, nello specifico abbiamo visto l'utilizzo del contenitore lista.

Procediamo a concludere tale trattazione con un esempio oggi: la classe fraction.h che ci è utile per calcolare i prodotti della distribuzione ipergeometrica.

La classe fraction.h ha due attributi privati:

- Due multiset num e den che sono multiset di unsigned int e rappresentano i numeratori ed i denominatori.
- Un boolean "zero" che ci dice se la frazione è zero o no

Nella parte pubblica abbiamo questi attributi:

```
class fraction{
    multiset<unsigned int> num,den;
    bool zero;
public:
    fraction() {zero=false;}
    fraction set_zero(){zero=true;num.clear();den.clear();return *this;}
    void mul(unsigned int _i);
    void div(unsigned int _i);
    fraction& operator*=(const fraction& _f);
friend ostream& operator<<(ostream& os,const fraction& _f);
long double compute();
long double compute2();//versione forse pi precisa
fraction simplify_as_factorial();
fraction simplify_with_primes();
bool is_zero()const{return zero;};
   };

ostream& operator<<(ostream& os,const fraction& _f);
unsigned long long int fact(unsigned int _i);
multiset<unsigned int> primeFactors(int n);
fraction hg(unsigned int _x, unsigned int _r, unsigned int _n, unsigned int _N);// following wu's notation
fraction coeff_binom(unsigned int n, unsigned int k); //following wikipedia's notation
void test_fraction();
```

Abbiamo un inizializzatore che rende la frazione non zero.

Abbiamo un set_zero che "azzera" l a frazione e svuota i set.

Abbiamo poi la moltiplicazione e la divisione.

Abbiamo poi un metodo per la stampa, due metodi per il calcolo, due metodi per la semplificazione.

Un metodo per il controllo se la frazione è zero.

Come metodi esterni abbiamo un metodo per la fattorizzazione e altri metodi molto complicati che non vedremo.

Vediamo ora le implementazioni di moltiplicazione e divisione:

```
void fraction::mul(unsigned int _i)
{multiset<unsigned int >::iterator it=den.find(_i);
 if (it!=den.end()) den.erase(it);
    else num.insert(_i);
};


void fraction::div(unsigned int _i)
{multiset<unsigned int >::iterator it=num.find(_i);
 if (it!=num.end()) num.erase(it);
    else den.insert(_i);
}
```

Nel caso della moltiplicazione si va a vedere se l'unsigned int è già presente nei denominatori, se è presente significa che possiamo semplificare direttamente e rimuovere dai denominatori l'unsigned per cui stiamo moltiplicando; setale unsigned non è presente nei denominatori lo inseriamo nei numeratori.

In modo speculare agiamo per la divisione.

Poi abbiamo l'override dell'operatore *, che va a sfruttare banalmente i metodi mul e div.

```
fraction& fraction::operator*=(const fraction& _f)
  { if(_f.zero) {set_zero(); return *this;}
      multiset<unsigned int >::iterator it;
   for(it=_f.num.begin();it!=_f.num.end();it++)
     {mul(*it);}
   for(it=_f.den.begin();it!=_f.den.end();it++)
     {div(*it);}
     return *this;
  }
```

Poi abbiamo il metodo per il calcolo compute() che va ad accumulare il numeratore ed il denominatore e solo alla fine a calcolare, inoltre il metodo utilizza i double per essere più preciso

```
long double fraction::compute()
{
  if (zero)
    return 0;
  long double temp_n, temp_d;
  temp_n = 1;
  temp_d = 1;
  multiset<unsigned int>::iterator it;
  for (it = den.begin(); it != den.end(); it++)
  {
    temp_d *= *it;
  }
  for (it = num.begin(); it != num.end(); it++)
  {
    temp_n *= *it;
  }
  return temp_n / temp_d; // come si fa con il tipo?
}
```

Il metodo compute2()

```
long double fraction::compute2()
{
  if (zero)
    return 0;
  long double temp = 1;
  long double temp_n, temp_d;
  multiset<unsigned int>::reverse_iterator itn, itd;
  itn = num.rbegin();
  itd = den.rbegin();
  while ((itn != num.rend()) && (itd != den.rend()))
  {
    temp_n = *itn;
    temp_d = *itd;
    temp *= temp_n / temp_d;
    //  cout<<"compute2 temp:"<<temp<<endl;
    itd++;
    itn++;
  }
  while (itn != num.rend())
  {
    temp_n = *itn;
    temp *= temp_n;
    itn++;
  }
  while (itd != den.rend())
  {
    temp_d = *itd;
    temp /= temp_d;
    itd++;
  }
  cout << "compute2 temp:" << temp << endl;
  return temp;
}
```

In questo caso l'idea è quella di fare le divisioni tra numeri che sono più simili come grandezza.

Il metodo simplify_with_primes() tenta di aumentare la precisione sostituendo numeratori e denominatori con i loro fattori primi.

```
fraction fraction::simplify_with_primes()
{
  fraction temp;
  if (zero)
    return temp.set_zero();
  multiset<unsigned int>::reverse_iterator itn, itd;
  multiset<unsigned int>::iterator it;
  multiset<unsigned int> primes;
  itn = num.rbegin();
  itd = den.rbegin();
  while ((itn != num.rend()) && (itd != den.rend()))
  {
    primes = primeFactors(*itn);
    for (it = primes.begin(); it != primes.end(); it++)
    {
      temp.mul(*it);
    }
    primes = primeFactors(*itd);
    for (it = primes.begin(); it != primes.end(); it++)
    {
      temp.div(*it);
    }
    itd++;
    itn++;
  }
```

Parte 2:

```
  while (itn != num.rend())
  {
    primes = primeFactors(*itn);
    for (it = primes.begin(); it != primes.end(); it++)
    {
      temp.mul(*it);
    }
    itn++;
  }

  while (itd != den.rend())
  {
    primes = primeFactors(*itd);
    for (it = primes.begin(); it != primes.end(); it++)
    {
      temp.div(*it);
    }
    itd++;
  }
  return temp;
}
```

Algoritmi della STL

Finora abbiamo utilizzato solo i metodi dei contenitori, ma il fatto che i contenitori utilizzano tutti gli iteratori ha reso possibile la creazione di metodi esterni che possono agire su tutti i contenitori, indifferentemente dal tipo del contenitore o dal tipo di dato contenuto.

Questi metodi sono suddivisi in primis in due categorie:

- Quelli che modificano il contenitore (es.: remove, replace, swap, merge, …)
- Quelli che no (find, …)

Queste funzioni sono create utilizzando l'iteratore più generale possibile, così da poter essere applicati al numero maggiore possibile di contenitori.

Introduciamo a questo punto la classe ranking.h, che ci servirà come base per testare i vari metodi della STL.

Un ranking non ha una struttura associata logicamente, quindi andremo noi a creare una classe wrapper che contenga le informazioni che ci servono.

```
class Ranking
{
private:
    list<string> l;
    set<string> s;

public:
    void add_item_back(string);

    friend ostream &operator<<(ostream &os, const Ranking &_r);
};

ostream &operator<<(ostream &os, const Ranking &_r);

void test_ranking();
```

Abbiamo una lista di stringhe per gli elementi che vogliamo valutare; abbiamo anche un set che ci serve per verificare che un elemento non venga inserito due volte.

Abbiamo poi una funzione di test ed una funzione di stampa.

Andiamo a vedere l'implementazione per la nostra classe.

```
#include "ranking.h"

void Ranking::add_item_back(string _s)
{
  if (s.find(_s) == s.end())
  {
    l.push_back(_s);
    s.insert(_s);
  };
}

ostream &operator<<(ostream &os, const Ranking &_r)
{
  list<string>::const_iterator it;
  for (it = _r.l.begin(); it != _r.l.end(); it++)
  {
    os << *it << " ";
  }
  return os;
}
```

Capiamo che questo ranking è stupido come un sasso: gli elementi si aggiungono solo in fondo, l'ordine deve essere definito in precedenza, inoltre non si possono avere pareggi.

Sostanzialmente l'unico controllo sull'inserimento nel ranking è che un elemento non venga inserito 2 volte.

La funzione di stampa è una funzione di stampa.

Il resto è troppo imbarazzante per essere descritto.

Ora che abbiamo una classe ranking possiamo iniziare a giocarci su con le funzioni di algorithm.

Aggiungiamo alla classe ranking le seguenti funzioni:

```
void  compare(const Ranking& _r)const;

void properties()const;
```

Cosa fanno queste due funzioni?

Funzione properties:

```
void Ranking::properties() const
{
  list<string>::const_iterator it;
  list<string>::const_iterator it3 = ++(++(++(l.begin())));

  int i = count_if(l.begin(), l.end(), short_name);
  cout << endl
       << "properties count_if: " << i;

  i = count_if(l.begin(), it3, short_name);
  cout << endl
       << "properties count_if*: " << i;

  it = find_if(l.begin(), it3, short_name);
  cout << endl
       << "properties find_if: " << *it;

  cout << endl
       << "properties all_of: " << all_of(l.begin(), l.end(), short_name);
  cout << endl
       << "properties any_of: " << any_of(l.begin(), l.end(), short_name);
  cout << endl
       << "properties none_of: " << none_of(l.begin(), l.end(), short_name);
}
```

```
bool short_name(string _s)
{
  if (_s.length() < 4)
    return true;
  else
    return false;
}
```

La funzione properties va a stampare il risultato di varie operazioni:

1) count_if va a contare gli elementi che rispettano una condizione dettata da una funzione passata come parametro. Nel nostro caso andiamo a contare tutti gli elementi che soddisfano la condizione dettata dalla funzione short_name all'interno del range definito da l.begin() e l.end().
2) In questo caso facciamo la stessa cosa ma restringiamo il range, poiché la fine è data da it3, che identifica il 4 elemento.
3) find_if trova se esite un tale elemento e lo ritorna (ritorna un iteratore alla prima occorrenza)
4) all_of ritorna true se tutti gli elementi del range rispettano la condizione short_name
5) any_of ritorna true se almeno un elemento del range rispetta la condizione short_name
6) none_of ritorna true se nessun elemento del range rispetta la condizione short_name

Nel nostro caso se andiamo ad inserire le università scritte in test_ranking() e poi lanciamo properties()

```
Output:
properties count_if: 3
properties count_if*: 1
properties find_if: MIT
properties all_of: 0
properties any_of: 1
properties none_of: 0
```

otteniamo questi risultati

Vediamo ora il metodo compare(); questo serve per confrontare tra loro due ranking.

```
void Ranking::compare(const Ranking &_r) const
{
  pair<list<string>::const_iterator, list<string>::const_iterator> p;
  p = mismatch(l.begin(), l.end(), _r.l.begin());
  cout << endl
       << "compare mismatch:" << *p.first << " " << *p.second;

  /*
pair<list<string>::const_reverse_iterator,list<string>::const_reverse_iterator> pr;
pr=mismatch(l.rbegin(),l.rend(),_r.l.rbegin());
cout<<endl<<"compare mismatch:"<<*pr.first<<" "<<*pr.second;
*/

  list<string>::const_iterator it3 = ++(++(++(l.begin())));
  cout << endl
       << "compare equal: " << equal(l.begin(), it3, _r.l.begin());

  cout << endl
       << "compare is_permutation: " << is_permutation(l.begin(), l.end(), _r.l.begin());
}
```

Innanzitutto anfdiamo ad ottenere il risultato dell'operatore mismatch() di algorithm, che va a prendere come argomenti un range (l.begin() ed l.end()) ed un altro iteratore (_r.l.begin()) e va a verificare quali elementi del range sono mismatchati con gli elementi di un secondo range che inizia dove è indicato dal secondo iteratore.

Se noi abbiamo due liste differenti un esempio possibile di output è il seguente:

```
Output:
compare mismatch:Oxford Caltech
compare equal: 1
compare is_permutation: 1
```

Il secondo output ci è dato dall'operatore equal. Funzione: equal(iteratr, iterator, iterator)

Il terzo output ci dice se le due sequenze sono l'una la permutazione dell'altra.

Funzione: is_permutation(iteratr, iterator, iterator)

Passiamo all'esempio 3, in cui andiamo a vedere delle funzioni di algorithm che vanno effettivamente a modificare le sequenze.

Andiamo ad aggiungere quindi il metodo `void modify();` nella classe ranking.

In questo caso inizializziamo il nostro ranking di test così:

```
void test_ranking(){
Ranking ordine;
ordine.add_item_back("Giovanna");
ordine.add_item_back("Franco");
ordine.add_item_back("Carla");
ordine.add_item_back("Martino");
ordine.add_item_back("Silvia");
cout<<ordine;
ordine.modify();
}
```

```
{list<string>::iterator it;
list<string>::iterator it3=++(++(l.begin()));
reverse(l.begin(),it3);
cout<<endl<<"modify reverse first two: "<<*this;

rotate(l.begin(),it3,l.end());
cout<<endl<<"modify rotate: "<<*this;

int d=l.size();
remove(l.begin(),it3,"Franco");
cout<<endl<<"modify remove Franco: "<<*this;
if (l.size()!=d) s.erase("Franco");

d=l.size();
remove(l.begin(),l.end(),"Franco");
cout<<endl<<"modify remove Franco: "<<*this;
if (l.size()!=d) s.erase("Franco");

replace(l.begin(),l.end(),string("Franco"),string("Francesco"));
cout<<endl<<"modify replace: "<<*this;
```

```
output:
Giovanna Franco Carla Martino Silvia
modify reverse first two: Franco Giovanna Carla Martino Silvia
modify rotate: Carla Martino Silvia Franco Giovanna
modify remove Franco: Carla Martino Silvia Franco Giovanna
modify remove Franco: Carla Martino Silvia Giovanna
modify replace: Carla Martino Silvia Giovanna
```

Vabbè, sostanzialmente vediamo le seguenti operazioni.

Reverse inverte un range passato come parametro.

Rotate prende un range dato in input e lo sposta (lo fa iniziare) in un punto definito da un secondo iteratore dato in input.

Remove rimuove un elemento in un dato range, se l'elemento non è presente nel range non succede nulla e questo si vede perché Franco non viene tolto la prima volta perché è fuori dal range, mentre nel secondo caso viene eliminato.

Replace sostituisce un elemento con un altro, se lo trova.

Notate che tutte le funzioni disponibili per algorithm sono ben documentoate su cplusplus.com quindi conviene dargli un'occhiata. La parte interessante da osservare è quella in cui sono presentate le fuzioni ed è specificato per quale tipo di iteratore possono essere applicate.

Dagli un'occhiata valà.

Lezione 7

Puntatori a funzione e oggetti puntatore

Il puntatore a funzione è proprio un puntatore ad una funzione, come tutto in C dev'essere tipizzato: per la tipizzazione è importante dichiarare i tipi degli argomenti ed il tipo di ritorno.

```
int (*pfunz)(double , int); //dichiarato

int f1(double,int);

int f2(double,int);

pfunz=f1;

…

pfunz=f2; //alternativamente

cout<<(*pfunz)(2.0,3); //oppure cout<<pfunz(2.0,3)
```

In questo esempio possiamo notare come viene dichiarato un puntatore a funzione con la tipizzazione del caso, inoltre si può osservare come il puntatore a funzione possa essere usato sia con dereferenziazione sia con chiamata semplice come fosse il nome stesso della funzione.

Naturalmente, essendoci un tipo per i puntatori a funzione, questi possono essere definiti tramite typedef. Una volta che ho definito un tipo puntatore ad un certo tipo di funzioni posso usarlo nei template.

Ecco un bell'esempio:

```
typedef double (*P)(double);

map<P,double> m;
m[f1]=2;
m[f3]=4;
```

Con f1 ed f2 definite in precedenza come funzioni del tipo definito come *P.

Arriviamo quindi a definire gli oggetti funzione: ogni classe in cui è definito l'operatore *operator()* è una funzione, questo significa che se una classe A ha operatore () definito allora ogni variabile a di tipo A può essere invocata così a(…) e usata come funzione.

```
class A{
      int i;
      public:
      A();
      A(int _i);
      double operator()(double _d)const;
      bool operator<(const A& _a)const;
      };
```

Eccezioni

Dovremmo averle già viste in Java, dove sono onnipresenti. Le eccezion sono un elemento che è stato inserito per gestire un flusso di controllo anomalo.

Nascono perché con l'aumentare della complessità del programma diventa più semplice anche che si verifichi qualche intoppo in livelli molto più bassi del nostro codice come errori nella gestione della memoria o nell'apertura di file; a questo punto diventa difficile trasportare gli errori avanti e indietro lungo tutto il codice, quindi si è creato il meccanismo di controllo delle eccezioni.

In C abbiamo quindi i controlli *throw, try, catch*.

Il blocco di codice all'interno di try viene eseguito ma il programmatore sa che potrebbe generare un'eccezione, il blocco di codice catch(…) viene usato per gestire le eccezioni che saltano fuori dal try.

```
try{//se qualcosa va storto lancio un'eccezione
    throw logic_error("grosso guaio");
    throw int(3);
    throw double(3.3);
   }
catch(int i){cout<<"intercettata eccezione"<<i;}
catch(logic_error e){cout<<e.what();throw;}
catch(...){cout<<"intercettata eccezione sconosciuta";}
}
```

In questo listato di codice il blocco try lancia una eccezione, una sola perché una volta che viene lanciata la prima non si eseguono più le righe sottostanti, e vediamo che si possono lanciare eccezioni di vario tipo.

I vari catch a seguito del try stampano il tipo di eccezione, oppure nel caso di *e.what()* stampano proprio il contenuto dell'eccezione. In fondo si vede un *catch(…)* che funziona come il *default* dello *switch* e gestisce quindi tutte le eccezioni non specificate dai catch precedenti.

Le eccezioni se non sono gestite da un catch e consumate allora interrompono il programma in malo modo.

In C++ la gestione delle eccezioni è pericolosa per questo motivo: se uno va ad includere ed usare una libreria che lancia eccezioni e non ne è al corrente potrebbe vedere il suo codice interrompersi brutalmente ed inaspettatamente.

Multithreading

I thread sono processi che condividono lo stesso spazio di indirizzamento della memoria, presentano molti vantaggi per l'efficienza al prezzo di una grande complessità implementativa.

Fino al C++11 non esisteva un modo standard ufficiale per il multithreading.

Oggi indagheremo tre librerie a sul multithreading, partiamo con thread.

Thread

Thread sostanzialmente è una classe.

Vediamo subito un esempio:

Supponiamo di avere una funzione f1() senza parametri ed una f2() che accetta un intero.

```
std::thread first (f1);     // nuovo thread  chiama f1()
std::thread second (f2,0);  // nuoveo thread chiama f2(0)

std::cout << "main, f1 e f2 lanciati...\n";

// sincronizza threads:
first.join();                // pausa attende il primo
second.join();               // pausa attende il secondo

std::cout << "f1 e f2  completati.\n";
```

Questo permesso ci rende possibile osservare come il primo parametro di un thread debba essere un callable ed i parametri successivi siano i parametri che vengono dati al callable stesso.

Il metodo *first.join()* chiamato nel main mette il main in attesa del thread *first*.

Osserviamo però ora il problema che se invochiamo due thread, uno che incrementa 10 000 volte un intero e l'altro che decrementa 10 000 volte lo stesso intero che era inizializzato a 0 all'inizio noi ci aspettiamo di trovare 0 alla fine, ma non è così!

Perché succede questo? Perché evidentemente ci sono problemi di sharing delle risorse, non è implementato un sistema di controllo sull'accesso alla memoria condivisa e, dato che incremento e decremento non sono azioni atomiche, è possibile che l'una sovrascriva l'altra e quindi si generino questi risultati non corretti e soprattutto non deterministici.

Questo problema si chiama data race.

Esistono due soluzioni, prevedono entrambe l'utilizzo di una libreria ad hoc, la prima che andiamo a vedere è la libreria Atomic.

Atomic

Altro non è che una template class che lavora con generics, in pratica io posso istanziare atomic per un intero, per un float o per tutti i tipi base, sostanzialmente atomic va a crearci attorno un wrapper che gestisce l'atomicità dell'accesso alla variabile.

Vediamo come si usa:

```
#include <atomic>
static std::atomic<int> condivisa(0);
```

Come si può osservare è molto semplice utilizzare atomic, l'unica nota negativa sta nel fatto cheuna volta dichiarata una variabile come atomic non si può riassegnare.

Si può definire come meccanismo ad alto livello perché nasconde all'utente tutta una serie di controlli che permettono di ottenere l'atomicità delle operazioni.

Vediamo ora il secondo metodo del C++ per risolvere il problema del data race.

Mutex

La libreria mutex permette di gestire a "basso livello" la mutua esclusione nell'accesso ad una variabile, viene detto a basso livello perché il programmatore deve gestire lock() e unlock() del mutex.

```
std::mutex myMutex;
static int condivisa=0;
void inc_thread(){
for(int i=0;i<10000;i++){myMutex.lock(); condivisa++; myMutex.unlock();}
}

void dec_thread(){
for (int i=0;i<10000;i++){myMutex.lock(); condivisa--; myMutex.unlock();}
}
```

Qui è presentata la soluzione del problema dei 10 000 incrementi che abbiamo visto in precedenza, si nota come il mutex venga usato come semaforo per la gestione delle precedenze.

Semplice no?

No! Perché possono succedere mille problemi:

- Il programmatore si dimentica di inserire l'unlock
- Un thread va in eccezione e non va più a sbloccare il mutex
- Vari ed eventuali

Per il problema degli unlock una soluzione semplice quella della classe lock_guard, che è una classe template che prende un mutex e ne fa da guardia, ovvero quando si esce dallo scope in cui la guardia è dichiarata, questa va a fare l'unlock del mutex: questo ci permette di risolvere anche i problemi con le eccezioni e più in generale alcuni problemi di dimenticanza di unlock.

Semplice no?

Ancora no! Non abbiamo parlato né di deadlock né di starvation: questi sono due problemi fondamentali ed onnipresenti quando si parla di mutex (semafori in generale).

Appuntiamo quissotto un breve riassunto dei problemi con la temporizzazione del multithreading.

- Deadlock
  - situazioni in cui uno o piu' processi/thread si bloccano
    - Omettere l'unlock un mutex
    - Terminazione inattesa di una funzione (lancio di eccezioni)
      - soluzione: lock_guard<mutex>
    - Funzioni annidate che chiamano lo stesso mutex
      - soluzione: recursive_mutex
    - Ordine di locking di diversi mutex
      - soluzione: lock(mutex, mutex)

- Starvation
  - la politica di accesso impedisce ad un processo di accedere alla risorsa

Esistono altre strategie che permettono di risolvere i problemi di multithreading, come semafori e monitor, che sono implementate in altri linguaggi di programmazione, ma il C++ rispettando la sua filosofia di base ha implementato il minimo indispensabile per mettere il programmatore in grado di risolversi i problemi da solo.

Lezione 8

R-value references e move semantics

Gli oggetti temporanei

Esistono in C++ degli oggetti temporanei che vengono creati per breve tempo e vengono subito dopo distrutti.

Vediamo subito un esempio

```
 A operator+(const A& _a1, const A& _a2) {

A temp(_a1);

…

return temp;}

A a1,a2,a3;

…

a1=a2+a3; //la parte a destra dell'assegnazione e' un oggetto
temporaneo

a1=a2; //la parte a destra non e' un oggetto temporaneo
```

Quello che succede ad esempio quando si usa l'operatore + con due variabili (quando si usa + come funzione esterna e non come metodo) viene creata una terza istanza temporanea di A per utilizzare il metodo.

Possiamo riscrivere la somma di sopra in questo modo:

a1.operator=(operator+(a2, a3));

dove la parte a destra del simbolo = è a tutti gli effetti un'istanza di A, che però è temporanea, poiché una volta copiato il suo valore in a1 verrà distrutto.

Questo ci introduce alla distinzione tra lvalues e rvalues.

Gli rvalues sono "cose" che possono stare a destra, che si possono assegnare.

Nota che la categoria dei valori è ortogonale rispetto al tipo delle variabili, ed in realtà le categorie sono molto più complicate: prvalues (pure r values), xvalues (extintion values, destinati a distruggersi), glvalues (generalized l values).

È possibile usare una referenza ad un valore temporaneo intero? (un rvalue intero)

int& x=6; //non compila

const int& x=6; //compila

Ma sorpresa sorpresa, dal c++11 le regole diventano queste

int&& i=7; // rvalue reference

Permette di fare riferimento ad un rvalue e di modificarlo utilizzando la doppia reference &&.

Le reference normali (int&) sono lvalue references.

Esiste una funzione template che restituisce una rvalue reference al suo argomento, questa funzione è move.

Sostanzialmente pare che questo sia utile per smettere di creare temporanei, poi copiarli ed infine distruggerli: grazie a move si può usare direttamente la referenza al temporaneo così da risparmiarsi il costo della copia.

Move semantics

L'idea di base della move semantics è di poter trasferire proprietà di dati allocai nella memoria dinamica senza doverle copiare. Lo scopo è quello di aumentare l'efficienza riducendo al minimo le copie necessarie.

Copia profonda

Dobbiamo andare a recuperare il concetto di copia profonda. La copia profonda è quella copia che viene invocata (e dovrebbe essere sempre implementata) quando si lavora con un oggetto che ha dei componenti dinamici, che quindi senza copia profonda vengono copiati solo per puntatore.

Per la copia profonda dobbiamo modificare in modo corretto *distruttore*, *costruttore di copia* e *operatore=*.

Ma ora che vogliamo usare la move semantics dobbiamo implementare anche il *costruttore move* e l'operatore *move operator*.

Ora riprendiamo l'esempio che abbiamo usato per esercitarci con la deep copy e lo evolviamo.

Osserviamo com'è implementato un tipico move constructor, in questo esempio la classe A ha attributo pb che è un puntatore ad un oggetto di tipo B

```
A::A(A&& _a){
 pb=_a.pb;
 _a.pb=NULL;
 cout<<"move costruttore"<<endl;
}
```

confronto copy constructor

```
A::A(const A& _a){
  i=_a.i;
  if (_a.pb!=NULL) pb=new B(*(_a.pb));
  else pb=NULL;
  cout<<"costruttore di copia"<<endl;
};
```

Nel move constructor succede questo, l'oggetto da cui viene prelevato il puntatore alla proprietà viene lasciato con un NULL di fatto in mano, questo si fa perché di solito tale oggetto sarà un temporaneo, quindi eliminato a breve e potrebbe richiamare distruttori sui dati che abbiamo appena copiato!

L'operatore move= (definito come *A::operator(A&& _a)*):

```
A& A::operator=(const A& _a){
  cout<<"operator="<<endl;
    if (this->pb==NULL) {//oggetto chiamante non ha B
      if (_a.pb!=NULL) pb=new B(*(_a.pb));
    }
    else //l'oggetto chiamante ha B
    {
      if (_a.pb!=NULL) (*pb)=*(_a.pb);
      else {delete pb; pb=NULL;}
    }
  return *this;// this e' un puntatore che punta all'oggetto chiamante
}; //a=(b=c)
```

Nel main dell'esempio possiamo trovare queste righe di codice

```
// prove move constructor
A a4(4,"Pinco");
cout<<a4.get_s();
A a5(std::move(a4));
cout<<a5.get_s();
cout<<a4.get_s();

A a6;
a6=std::move(a5);
cout<<a6.get_s();
cout<<a5.get_s();
```

Qui possiamo notare come il move venga invocato esplicitamente ogni volta: in caso contrario l'overloading dell'operatore farebbe partire il copy constructor invece che il move constructor.

Lezione 9

Move semantics 2

Come dicevamo la scorsa lezione, ogni classe che abbia al suo interno dei puntatori, quindi implementi meccanismi di memoria dinamica, dovrebbe implementare i seguenti metodi:

Distruttore
costruttore di copia
operator=
Costruttore move
move operator=

Gli ultimi due metodi sono specifici per l'implementazione della move semantics per quella classe.

Il costruttore move deve "consumare" l'istanza passata, perché appunto questa è temporanea, lo vediamo meglio in esempio 0:

```
A::A(A&& _a){
        i=_a.i;
        _a.i=0;
        cout<<"move costruttore"<<endl;
}
```

Default e delete

Queste sono due parole chiave introdotte in C++11 che permettono di specificare le azioni da compiere in questo caso, vediamo un esempio.

```
class A{
        int i;
        public:
        A();
        A(int _i);
        A(const A& _a) = default;
        A(A&& _a);
        A& operator=(const A& _a) = default;
        A& operator=(A&& _a);
```

default serve per esplicitare che vengono utilizzati i metodi di default.

delete invece serve per fare in modo che una classe non presenti il metodo (che altrimenti sarebbe quello di default)

```
A& operator=(const A& _a) = delete;
A& operator=(A&& _a);
```

Smart pointers (#include <memory>)

Gli smart pointers sono classi template che realizzano delle cose che assomigliano ai puntatori che gestiscono la memoria in modo "furbo" implementando dei controlli su come la memoria viene gestita.

Semplificano ad esempio quei processi che in C++ si dovrebbero implementare a mano, come ad esempio meccanismi di delete o di costruzione di copia.

Unique_poiner

L'idea di base è che questo puntatore non possa essere copiato: supporta soltanto l'assegnamento per move e non per copy. Si usa quando abbiamo degli oggetti che vogliamo vengano riferiti da un solo puntatore, questo è utile ad esempio perché il puntatore ad un oggetto è uno solo, quindi abbiamo un controllo molto più semplice della gestione della rimozione degli oggetti dalla memoria.

```
std::unique_ptr<int> p,p1;
p.reset(new int(54));
std::cout<<*p;
p1.reset(new int(23));
std::cout<<*p1;
p1=std::move(p);
// std::cout<<*p;
std::cout<<*p1;
```

reset è il metodo che si usa per inizializzare il puntatore unique.

Non si può assegnare il puntatore unique se non con la move, di fatto p dopo la move non contiene più nulla di sensato.

Abbiamo detto che sono puntatori furbi, ma in che senso? Beh ecco, mantengono informazioni sul loro proprietario per compiere tutta una serie di controlli come ad esempio "se il mio proprietario viene cancellato allora faccio in modo di cancellarmi pure io". Il prorietario è quell'oggetto che ha la responsabilità di gestire il puntatore.

Unique pointer – unique_ptr

unique_ptr è uno smart pointer che ammette di essere puntato da un solo oggetto, questo corrisponde poi anche all'owner dello smart pointer.

Vediamo un utilizzo di questo unique pointer nella nostra amata classe A

```
A& A::operator=(const A& _a){i=_a.i;
  cout<<endl<<"operator="<<endl;
    if (!upb) {//oggetto chiamante non ha B
                if (_a.upb) upb.reset(new B(*(_a.upb)));
                    }
            else //l'oggetto chiamante ha B
            {if (_a.upb) (*upb)=*(_a.upb);
               else {upb.reset(NULL);};
            }
       return *this;// this e' un puntatore che punta all'oggetto chiamante
 }; //a=(b=c)
```

Alcuni metodi dello uniquie_ptr

- reset per resettare il valore di un puntatore
- operator= per l'assegnamento, non esiste il copy assignment perché appunto il puntatore dev'essere unico
- diverse salse di costruttori diversi
- get è un metodo che permette di ritornare lo *stored pointer* ovvero il puntatore ai dati effettivi, tuttavia la ownership dello unique pointer viene mantenuta (l'owner attuale rimane il responsabile della gestione dello up)
- release è come get ma rilascia la ownership dell'up

Shared pointer – shared_ptr

Al contrario dell'up prevede di avere più proprietari, la responsabilità sulla gestione della memoria è data all'ultimo proprietario ad essere eliminato.

Questo è utile sempre per tenere traccia dei puntatori ed evitare i dangling pointers.

Weak pointer – weak_ptr

Condivide una risorsa ma senza averne l'ownership e quindi senza aumentare il suo use_count(). Per accedere al contenuto è necessario copiarlo in uno shared_ptr usando un metodo lock(), finché non si compie questa azione non si può dereferenziare il puntatore (quindi non si può accedere al contenuto).

Il weak pointer con l'uso di lock() non diventa proprietario del puntatore ma copia il puntatore in uno shared pointer e rende impossibile agli altri owners modificare il puntatore mentre il weak è ancora in lock(). Questo fino a che non viene rilasciato il puntatore creato con lock(), questa azione "sblocca" il puntaore. Può essere usat anche per sincronizzare l'accesso alle risorse.

A quanto pare anche 'sta volta Blanzieri non si è preparato una sega prima della lezione… studia scemo che non sei altro!

Lezione 10

Metaprogrammazione – Template meta-programming in C++

Un metaprogramma è un qualsiasi programma che abbia come input un altro programma, l'esempio più banale è il compilatore, che prende un programma e lo trasforma in eseguibile.

Oggi parleremo della metaprogrammazione tramite template, andiamo quindi a ripassare i template.

I template sono tipi nominali che possono essere usati al posto di un tipo effettivo per creare del codice che possa essere usato poi con più tipi diversi, ad esempio la keyword *template<E>* ci permette di scrivere funzioni o classi o strutture con E come parametro.

I template poi possono essere anche specializzati, ovvero si può andare a definire una particolare istanza del template per un certo tipo, l'esempio lampante è il template vector<bool> che è un overloading del template vector<T> (dove T è un placeholder); questa versione di vector è implementata in un modo particolare (per guadagnare spazio) e con metodi specifici dedicati.

Deduzione dei tipi di un template: come fa il compilatore a trovare il tipo?

La computazione dei tipi con i template (in metaprogrammazione) avviene in modo ricorsivo, vediamo un esempio di definizione ricorsiva di template:

```
#include <iostream>

template <unsigned int n>
struct factorial {
   enum {value=n*factorial<n-1>::value
   };
};

template<>
struct factorial<0>{
enum {value=1};
};


int main(int argc, char** argv) {
    std::cout<<factorial<7>::value;
   return 0;
}
```

Il primo template rappresenta un fattoriale di un certo unsigned int con parametro formale n.

Il secondo template invece rappresenta il fattoriale di 0.

Quando il compilatore compila questo main istanzia factorial<7> e per farlo si fa il calcolo ricorsivo, quindi calcola il fattoriale a compile time!!

Nota interessante sul primo codice metaprogramma: serviva per calcolare tutti i numeri primi… difatti non compilava.

Come si fa ad inserire delle condizioni in metaprogrammazione? In fondo posso scrivere solo definizioni di tipi e non posso usare variabili o memoria.

Abbiamo un primo template generico che prende tre tipi, un booleano, una classe Then ed una classe Else, ma vediamoci chiaro:

```
template <bool condition, class Then, class Else>
struct IF
{typedef Then RET;
};

template <class Then, class Else>
struct IF<false,Then,Else>
{typedef Else RET;
};


int main(int argc, char** argv) {
   IF<sizeof(int)<sizeof(long),long,int>::RET i;
   std::cout<<sizeof(i)<<std::endl;
   return 0;
}
```

Nel main nella prima riga stiamo definendo un tipo *i* in base alla condizione *sizeof(int)<sizeof(long)*, se risulterà true il tipo sarà deciso dall' Then altrimenti dall'Else.

```
template <bool condition, class Then, class Else>
struct IF
{typedef Then RET;
};

template <class Then, class Else>
struct IF<false,Then,Else>
{typedef Else RET;
};

template <class T1, class T2>
typename IF< sizeof(T1)<sizeof(T2),T2,T1 >::RET max(T1 a, T2 b)
{if (a>b)
   return a;
   else return b;
}

int main(int argc, char** argv) {
//std::cout<<max(2,3);
//std::cout<<max(3,2.3);
  std::cout<<max(int(2),double(3.14));
  std::cout<<sizeof(double);
  return 0;
}
```

In questo esempio implementiamo tramite metaprogramming la funzione max, ritornando un risultato che può essere di tipo diverso in base all'input! Questo secondo passaggio lo otteniamo grazie alla funzione IF che abbiamo visto prima.

Esempio max_IF

Stage	Template definition	Compilation time	Run-time
Role	Design of algorithms The templated code has been defined	Template instantiation Parameter deduction happen Metaprograms are executed	Run of the algorithm Program evaluates expressions
Example	Return type of max(T,S) defined with metaprogram	Parameter deduction determines T and S. `IF<T,S>::RET` is selected	Greater argument value is chosen to return

Table 2. Programming with template metaprograms

Questa è una tabella riassuntiva per spiegare come è costruito questo max_if, si noti particolarmente che durante la compilazione vengono effettivamente dedotti i tipi dei template e quindi eseguiti tutti i metaprogrammi.

Però occhio che non tutto viene calcolato a compile time, in questo caso sappiamo che IF viene sempre calcolato a compile time (come ad esempio il fattoriale che abbiamo visto prima) ma il calcolo del maggiore potrebbe tranquillamente essere compiuto a runtime.

Riassumiamo i punti chiave della metaprogrammazione

- C++ calcola a compile time tutte le espressioni possibili
- Vengono istanziati effettivamente solo i tipi che si usano (laziness)
- I template possono essere specializzati
- I template devono essere dedotti quando sono usati
- Non si possono fare assegnazioni, non c'è il concetto di variabile nel metaprogramming

Dato che non ci sono assegnamenti, la metaprogrammazione è funzionale; vediamo quindi qualche altro esempio di funzione.

```
// Accumulate(n,f):=f(0) + f(1)+...f(n)

template <int n, template<int> class F>
struct Accumulate{
  enum {RET=Accumulate<n-1,F>::RET+F<n>::RET};
  };


template <template<int> class F>
struct Accumulate<0,F>
{enum {RET=F<0>::RET};
};

template <int n>
struct Square
{ enum {RET=n*n};
};

int main(int argc, char** argv) {
  std::cout <<Accumulate<3,Square>::RET<<std::endl;
  return 0;
}
```

E chi lo capisce questo? Perché non parla come mangia?

Come ha fatto a diventare un professore?

Nota interessante: la metaprogrammazione non è stata inserita volutamente nel C++ ma è stata “scoperta” lì nel cuore dei template. Per questo la sintassi della metaprogrammazione fa schifo (a detta di tutti, pure della mamma di C++) in questo linguaggio.

Dunque il meccanismo della metaprogrammazione in C++ è un hacking del linguaggio che utilizza dei trucchi per fare a compile time cose che altrimenti avremmo dovuto fare a runtime, alcuni dei “sinonimi” usati per la metaprogrammazione sono i seguenti

	Runtime functional program	C++ template metaprogram
values	run-time data (constant, literal)	static const and enum class members
variables	variables	symbolic names (typenames, typedefs)
initialization	constants generators	static const initialisation enum definition
assignment	no	no
i/o helpers	monads	warnings, error messages no interactive input
branching	pattern matching function specialization	pattern matching template specialization
looping	recursive functions	recursive templates
subprogram	function	(template) class
data types	abstract data structures	typelists, boost::vector
types	type class (Haskell)	concepts

Table 3. Comparison of functional programs and template metaprograms

Quindi sostanzialmente la metaprogrammazione è utile perché

- Può spostare delle computazioni da runtime a compile time (efficienza)
- Può generare tipi che dipendono dall’hardware
- Può generare tipi specializzati più efficienti

Ma il metaprogramming è turing-completo?

Sì! C’è un articolo interessante a riguardo.

Lezione 11

Bitset

I bitset sono un template della STL che si concretizza in un array di bit. La particolarità della libreria che implementa i bitset è che li rappresenta esattamente come un bit fisico: la dimensione è il minimo necessario, p questo il vantaggio del bitset.

Le operazioni sul singolo bit sono ad esempio:

- Set (mette il bit a 1)
- Reset (mette il bit a zero)
- Flip (flippa il bit)

Vediamo un piccolo listato di codice con qualche funzione dei bitset:

```
#include<bitset>

using namespace std;

int main(int argc, char** argv) {
   bitset<64> b;
   cout<<b.size()<<endl;
   cout<<endl<<b<<" b";
   string s="10101010101011010101";
   cout<<endl<<s<<" s";
   b=bitset<64>(s); cout<<endl<<b<<" b=bitset<64>(s)";
   b<<=4; cout<<endl<<b<<" b<<=4";
   cout<<endl<<(b<<3)<<" b<<3";

   cout<<endl<<(b&(b<<3))<<" b&(b<<3) ";
   cout<<endl<<(b|(b<<3))<<" b|(b<<3)";
```

Qui con b<<=4 si fa uno shift a sx con inserimento di 0 ,lo shift non è circolare, le cifre più significative se escono dalla rappresentazione vengono eliminate.

La differenza con b<<3 è che in questo caso viene restituita una copia e non viene sovrascritto l'originale.

In seguito possiamo vedere le operazioni di and logico (operatore *&*) e or logico (*|*)

Output:

```
64


0000000000000000000000000000000000000000000000000000000000000000 b

10101010101011010101 s

0000000000000000000000000000000000000000000010101010101011010101 b=bitset<64>(s)

0000000000000000000000000000000000000000000101010101010111010101 0000 b<<=4

0000000000000000000000000000000000000000001010101010101110101010000000 b<<3

0000000000000000000000000000000000000000000000000000100100000000000 b&(b<<3)

0000000000000000000000000000000000000010111111111111111111111010000 b|(b<<3)
```

Esiste anche un altro modo per inizializzare direttamente il bitset con una stringa (diverso dall'esempio precedente), la stringa viene inserita a partire dalle posizioni meno significative.

Questo metodo, tuttavia, non fa uso di una stringa ma di uno isstringstream, quindi dobbiamo fare la conversione.

```
istringstream(s)>>b; cout<<endl<<b<<" s>>b";
```

Output:

```
00000000000000000000000000000000000000000000101010101010111010101 s>>b
```

Inoltre, si possono convertire i bitset in interi, long, long long ecc. (se hanno una dimensione che lo permette).

```
cout<<endl<<(b<<32)<<" b<<32";
cout<<sizeof(long long)<<" ";
cout<<(b<<32).to_ullong();
cout<<endl<<b<<" b"<<b.to_ulong();
//cout<<(b<<32).to_ulong();
```

Infine, vediamo queste ultime operazioni

```
b.flip(); cout<<endl<<b<<" b.flip()";
b.reset(42); cout<<endl<<b<<" b.reset(42)";
if (!b[42]) b[41]=0; cout<<endl<<b<<" (!b[42]) b[41]=0";
```

Il flip() chiamato così flippa tutto il bitset, il reset(x) va a resettare specificatamente il bit in posizione x.

Inoltre, dall'ultima riga possiamo vedere diverse funzioni del bitset: possiamo usarlo come un array, possiamo usarne i singoli elementi come booleani ed infine possiamo fare assegnamenti alla singola cella dell'array. Nota però che tutto queste operazioni che sembrano molto semplici in realtà nascondono della notevole complessità dato che i singoli bit non possono essere indirizzati, quindi sotto c'è tutta una logica per nascondere queste operazioni di traduzione.

Lezione 12

Richiami alle espressioni lambda

Le espressioni lambda sono espressioni senza nome.

La loro differenza fondamentale dalle altre funzioni sta nel fatto che possono *catturare* le variabili.

La cattura permette alle lambda di ottenere per copia o per riferimento delle variabili che si trovano nello stesso blocco in cui la lambda stessa è invocata. Ricordiamo che la struttura della lambda è

[captures] (params) -> ret {body}

Nella cattura si può specificare se questa avviene per riferimento (tramite &) o per valore/copia (simbolo =, si può specificare anche un valore di default).

Negli esempi successivi utilizzeremo la libreria functional.h, che ci mette a disposizione la classe function che useremo per istanziare oggetti funzione, useremo in particolare il costruttore che prende come parametri una lambda expression.

```
0 0 2 9 24 50 90 147 224 324 450 605 792 1014 1274 1575 0 0.00833333
0.0166667 0.025 0.0333333 0.0416667 0.05 0.0583333 0.0666667 0.075
0.0833333 0.0916667 0.1 0.108333 0.116667 0.125
--------------------------------
```

```
#include <iostream>
#include <vector>
#include <functional>
using namespace std;
int main()
{
vector<int> v(16);
vector<function<float()> > vf(16);
 int j=0;
for (unsigned int i=0;i<v.size();++i)
{
      //v[i]=i;
      auto x=[&,j](int i){return i*j;};
      v[i]=x(i);
      function<float()> f=[i,&j](){return i/(1.0*j);};
      vf[i]=f;
      j+=i;
}

for (auto x:v) std::cout<<x<<" ";

for (auto x:vf) std::cout<<x()<<" ";
return 0;
}
```

All'interno del ciclo for troviamo una x tipizzata con auto che prende il valore di una lambda, questa lambda prende come parametro di cattura j e lo prende per riferimento (nota che la notazione [&, *argomenti* …] rende tutte le catture specificate al posto di *argomenti* vengano effettuate per riferimento).

In ogni iterazione del ciclo viene quindi creata una lambda diversa. Poi insomma sì vengono fatte delle operazioni ma sono complicate af dannato prof.

Ma l'esempio non è finito. Nella 4^ riga del for abbiamo una cosa interessante, tipizziamo staticamente e creiamo una funzione f (una per loop) che cattura i per copia, j ed ha il suo body in cui fa dei calcoli per i e j. Alla fine, questa f viene salvata all'interno del vettore vf.

Cosa abbiamo salvato in vf? Dei valori? No! Abbiamo salvato delle funzioni che non sono ancora state calcolate! E la cosa interessante è che i valori di i li abbiamo presi per copia, quindi ogni elemento di vf[] avrà il suo proprio valore di i, ma siccome j è stata passata per riferimento il suo valore è uguale per ogni elemento di vf[] e corrisponde al valore finale di j.

Cosa succede se lo scope di j termina e io invoco la lambda? Beh, j per essere catturata deve essere nello stesso scope, quindi non si dovrebbe porre facilmente il problema. Se si va a ritornare la lambda ad uno scope diverso da quello della sua creazione allora lì sono dolori…

Considerazioni conclusive e wrap-up

Si possono trovare informazioni interessanti sull'evoluzione degli standard del C++ sul sito isocpp.org