Corso di Laboratorio 2 – Primo semestre

Silvia Arcelli

6 Ottobre 2014

Librerie

- Una libreria è un file contenente **codice compilato** di utilità comune a diversi programmi che può essere successivamente incorporato in un programma in fase di linking;

- Nell'ambiente Linux (come nella maggior parte dei sistemi moderni) le librerie si suddividono in due famiglie principali:
  - librerie **statiche** (*static libraries*)
  - librerie **dinamiche** o **condivise** (*shared libraries*)

Librerie Statiche

Costruzione di una libreria statica a partire dai moduli oggetto:

g++ -c myfunc.cpp myfunc2.cpp

Il comando **ar** (*archiver*) invocato con la flag ''**r**'' crea la libreria e vi inserisce i moduli oggetto:

ar r libmytest.a myfunc.o myfunc2.o

Per il nome di una libreria statica è generalmente utilizzata la seguente convenzione: il nome del file della libreria inizia con il prefisso ''**lib**'' e termina con il suffisso "**.a**".

Librerie Statiche

- effettuare il link ad una libreria statica:

  g++ -o test main.cpp -L. –lmytest

- omesso il prefisso ''**lib**'' e il suffisso ''**.a**'' con la flag "**-l**“ (Ci pensa il linker ad attaccare queste parti alla fine e all'inizio del nome di libreria).

- L'uso della flag ''**-L.**'' che dice al compilatore di cercare la libreria anche nella directory corrente e non solo nelle directory standard dove risiedono le librerie di sistema (per es. /usr/local/lib/, /usr/lib).

Librerie Statiche

- viene fatta una ricerca nei moduli all'interno di questa libreria per localizzare le funzioni non definite nel main. Una volta localizzati, solo questi moduli vengono estratti dalla libreria ed inclusi nell'eseguibile del programma.

- I programmi utilizzano la libreria statica distintamente, cioè ognuno ne possiede una copia: se sono eseguiti contemporaneamente nello stesso sistema, i requisiti di memoria si moltiplicano inultilmente per utilizzare funzioni identiche.

- Questa limitazione si supera con l'utilizzo di librerie dinamiche

Librerie Dinamiche

- Le librerie condivise (dette anche dinamiche) vengono collegate ad un programma in due passaggi.

- In un primo momento, durante la fase di compilazione (*Compile Time*), il linker verifica che tutti i simboli richiesti dal programma siano effettivamente collegati o al programma o ad una delle sue librerie , ma i moduli oggetto della libreria dinamica non vengono inseriti nel file eseguibile.

- In un secondo momento (*Run Time*), quando l'eseguibile viene lanciato un programma di sistema (*dynamic loader*) controlla quali librerie dinamiche sono state collegate al nostro programma, le carica in memoria, e le "connette" alla copia del programma in memoria.

Librerie Dinamiche

- La fase di caricamento dinamico rallenta il lancio del programma, ma si ha il vantaggio che, se un altro programma collegato alla stessa libreria viene lanciato, questo utilizza la stessa copia della libreria dinamica già in memoria: ciò consente un notevole risparmio delle risorse del sistema.

- Per esempio, le librerie standard del C e del C++ sono delle librerie condivise utilizzate da tutti i programmi C/C++. L'uso di librerie condivise permette di avere eseguibili molto più "snelli" e utilizzare meno memoria in esecuzione.

Librerie Dinamiche

- Una libreria condivisa ha un formato specifico che dipende dall'architettura sulla quale è stata creata. Per generarla si usa la flag "**-shared**" :

  g++ -fPIC -c myfunc.cpp

  g++ -fPIC -c myfanc2.cpp

  g++ -shared -o libmytest.so myfunc.o myfunc2.o

- Si utilizzata la convenzione il nome del file della libreria inizia con il prefisso "**lib**" e termina con il suffisso "**.so**".
- Il link ad una libreria condivisa è del tutto analogo a quello ad una libreria statica

  g++ -o test main.cpp -L. -lmytest

ROOT

un programma sviluppato per la comunita' dei Fisici al Cern (il Laboratorio di Fisica Subnucleare di Ginevra) e attualmente utilizzato in molti laboratori di ricerca nazionali ed internazionali.

ROOT Home page:
http://root.cern.ch

ROOT Team:

- autori principali: Rene Brun & Fons Rademakers
- una decina di collaboratori "senior"
- >150 "contributors" (utenti)

ROOT

Un Framework di Analisi Object Oriented:

- Scritto in C++, come la maggior parte del software utilizzato negli esperimenti di fisica dell'attuale generazione
- E' scritto da Fisici per i Fisici
- E' Open Source: sorgente e eseguibile "free", scaricabili dall'utente
- Molte piattaforme supportate
- Ottimale per l'analisi e per la gestione di grandi volumi di dati: molti esperimenti di fisica delle alte energie hanno attualmente volumi di dati dell'ordine del PB/anno di presa dati

ROOT

Circa 1200 classi (~650’000 linee di codice), organizzate in una sessantina di librerie:

- Classi Base
- Classi per analisi avanzata di istogrammi (1,2,3 & n-D)
- Classi di Grafica (2-D e 3-D)
- Trees (strutture ottimizzate per la gestione di grandi volumi di dati dello stesso tipo, strutturalmente complessi)
- Classi per manipolazione di Matrici, Vettori , funzioni matematiche, minimizzazione, generatori random, Reti Neurali
- Classi Container
- Interfaccia con il sistema operativo
- Database SQL, Networking, Parallel Processing
- Classi per user interface (GUI, interattiva, macros)

ROOT

Documentazione utile:

- Manuale scaricabile al sito http://root.cern.ch/drupal/content/users-guide
- Reference Guide: http://root.cern.ch/drupal/content/reference-guide
- ROOT Tutorials: http://root.cern.ch/root/html/tutorials

- Installatelo al più presto! Sempre al Link:

http://www.bo.infn.it/~arcelli/LezioniLabII.html

è disponibile una mini-guida per l'installazione di root a partire dal sorgente.Se avete difficoltà segnalatemelo (mandatemi un e-mail)

Linguaggi di Programmazione-Generalità

- Componenti principali di un programma software:
  - Algoritmi: l'insieme di istruzioni che svolgono un particolare compito
  - Dati: ciò su cui operano gli algoritmi

- La relazione fra queste due componenti definisce il cosiddetto *paradigma di programmazione*:
  - Programmazione Procedurale: problemi modellati dagli algoritmi (serie di istruzioni raggruppate in *subroutine* che operano sui dati, immagazzinati in aree comuni o passati come argomenti)
  - Programmazione a Oggetti: problemi modellati dalle relazioni fra tipi di dati astratti (oggetti), che incorporano dati (attributi) e metodi (member functions)

- Il C, FORTRAN sono linguaggi che supportano lo stile di programmazione procedurale. Il C++, "nato" dal C, supporta la programmazione a oggetti.

Un programma in C++:

•La funzione main è il punto di accesso principale al programma, obbligatoria (in C/C++ ogni modulo di programma è una funzione)

•Il programma si compone di statement. Uno statement è costituita da unità individuali interpretate dal compilatore, i "*tokens*" (keywords, identificatori, costanti letterali o numeriche, operatori,..). Blanks e separatori vengono ignorati. In C++ ogni statement deve terminare con ";".

•Il C++ è un linguaggio *"tipato"* ed esercita un controllo forte sui tipi (*"strong type checking")* che limita le conversioni (*casting*) tra variabili di tipo diverso. Per ogni variabile (in generale, per ogni *identificatore*) deve essere specificato il tipo.

Identificatore: nome simbolico assegnato ad un'entità di linguaggio, riconosciuto dal compilatore: variabili, funzioni, array, strutture, classi

Dichiarazione di variabili

•Il tipo è una caratteristica di tutte le variabili che sono memorizzate nello stesso modo e a cui si applicano lo stesso tipo di operazioni :

Cinque tipi nativi ("intrinseci", noti a priori al compilatore):

		# bytes
char.	numero intero interpretabile come codice ASCII di un carattere	1
bool:	tipo booleano, true o false	4
int:	numero intero	(4-16)
float:	numero in virgola mobile con 6-7 cifre significative	4
double:	numero in virgola mobile con 15-16 cifre significative	(8-16)
void:	*(assenza di tipo)*	non definita

•Dimensioni tipi nativi (eventualmente con *qualificatori:* short, long,unsigned,...) dipendenti dalla piattaforma. Il linguaggio fissa solo dei limiti al numero di bit dedicati alla rappresentazione dei tipi predefiniti

•Operatore sizeof per conoscere la lunghezza in byte degli identificatori appartenenti ad un certo tipo predefinito: sizeof(int), sizeof(float) , etc

Dichiarazione di variabili

•In C++ il primo statement in cui compare una variabile all'interno di un programma (o meglio, di un ambito di visibilità) è appunto una dichiarazione (definizione):

```
int myvar;
double myvar1,myvar2;
```

• All'interno dello stesso ambito di visibilità, non sono consentite definizioni multiple (anche se consistenti) della stessa variabile.

•Regole per i nomi di identificatore :

- possono contenere solo lettere, numeri e _ (63 caratteri)
- devono cominciare per una lettera o _
- lunghezza minima 1 carattere, massima 256 caratteri
- C++ è ***case sensitive***

•Non possono essere utilizzate come nomi di variabile le parole riservate del linguaggio (return, else, if, typedef, inline, etc..)

Dichiarazione di variabili e scope

•La definizione può essere fatta in qualunque punto del programma ((non necessariamente all'inizio del programma, come in C, ma, ovviamente, prima di usare la variabile) . Si raccomanda di fare contestualmente anche l'inizializzazione:

```
float g = 10.; // definizione con inizializzazione della variabile g
```

•La sua visibilità (scope) corrisponde al blocco più interno che contiene la sua definizione, o all'intero file se è definita nel global scope (al di fuori di qualunque graffa)

•I blocchi in C++ sono definiti dalle parentesi graffe, che sono usate per definire le funzioni o i namespace, e possono essere usate anche per limitare la visibilità di una variabile

Dichiarazione di variabili e scope

•Variabili locali, dette anche *"automatiche",* ovvero il cui tempo di vita è limitato all'esecuzione della blocco/funzione:

- •Ogni variabile è visibile e utilizzabile solo nello stesso ambito in cui è definita . Se si tenta di utilizzarla in ambiti esterni (o in ambiti superiori se ci sono scope annidati) si ha un errore (esempio: indice di un blocco for)
- •Nel C++ è ammesso ridefinire la variabile solo se ciò avviene in ambiti diversi (anche annidati fra loro, in questo caso riconosce la variabile definita nel proprio ambito)

•Variabili globali, definite al di fuori di qualunque ambito e visibili a tutto il programma (anche in altri files, a patto che siano lì dichiarate extern):

- •In caso di ambiguità fra variabile globale e locale il compilatore dà prevalenza alla variabile locale.
- •Le variabili globali sono sempre variabili *statiche,* che sono persistenti in memoria per tutto il tempo di vita del processo

Dichiarazione di variabili e scope

```
#include <iostream>
using namespace std;
int i =10; // global scope
int main(void){
int i=20; // function scope, nasconde i a file scope
{
  int i =30; //block scope, nasconde i a function scope
  cout << "i (block scope) è =" << i <<endl; //accessibile solo nel block
}
cout << "i (function scope) è =" << i <<endl;
cout << "i (global scope) è =" << ::i <<endl; //"::" è lo scope resolution
                                                //       operator!
return 0;
}
```

sullo standard output:
i (block scope) è = 30
i (function scope) è = 20
i (global scope) è = 10

Dichiarazione di variabili e namespace

Per evitare in uno stesso ambito il conflitto fra variabili o funzioni che hanno lo stesso nome, (***name clash***) si possono usare i namespace :

```
#include <iostream>
using namespace std;
int i =10; // global scope
namespace myspace
{
  int i =30; //namespace scope
}
int main(void){
int i=20; // function scope, nasconde i a file scope
cout << “i (global scope) è =” << ::i <<endl;
cout << “i (function scope) è =” << i <<endl;
cout << “i (namespace) è =” << myspace::i <<endl;
return 0;
}
```

sullo standard output:
i (global scope) è = 10
i (function scope) è = 20
i (namespace) è = 30

Operatori dei tipi nativi

- Un operatore opera su uno o più operandi. In C++ sono definiti una serie di operatori per i tipi nativi del linguaggio, che possono essere unari, binari, ternari *("arietà").* Per ogni operatore esiste un ben definito set di tipi a cui può essere applicato.

- Gli operatori sono contraddistinti da un *simbolo riservato* e da un livello di *precedenza* rispetto agli altri, che definisce l'ordine di valutazione delle espressioni che contengono questi operatori.

  a*b+c equivale a (a*b)+c : l'operatore * ha precedenza su +

- da una regola di *associatività* : a parità di precedenza, qual è la sequenza di applicazione degli operatori

  *p++ equivale a *(p++) : dereferenziazione e incremento postfisso associativi a destra

- Precedenza ed associatività sono modificabili attraverso l'uso delle parentesi tonde

Operatori Unari e Binari

•Operatori di auto incremento e decremento (++,--):

Sono tra gli operatori con **massima priorità** (come la chiamata a funzione e la selezione di un membro). Agiscono sull'operando (di qualunque tipo nativo) in(de)crementando di 1 prima/dopo di usarne il valore nell'espressione, a seconda se prefisso o postfisso.

```
int a,b,c=5;
a=c++;
b=++c;
```

Il risultato di queste operazioni è
a=5,b=7, c=7

```
int scale(int n){
  n*=3;  cout << "n= " << endl;
 return n;}
int a=5, myvec[10];
cout << scale(a++)<<endl;
myvec[++a]=3
```

stampa 15
incrementa l'elemento con indice 6

Operatori Unari e Binari

•Operatori aritmentici (+,-,*,/): si applicano a tutti i tipi nativi, con la condizione che i due operandi siano dello stesso tipo.

- •Sottrazione e somma hanno minore priorità rispetto a prodotto e divisione.
- • A parità di priorità si valutano da sinistra a destra. →

L'operatore modulo (%) si applica solo ad interi.

•Operatori relazionali: si applicano a tutti i tipi nativi, e restituiscono un bool; >,< ,>= etc hanno priorità su == e !=. Si valutano da sinistra a destra: →

3 > 5 == 5 >= 8 restituisce true

3 > (5 == 5) >= 8 restituisce ?

Operatore di Assegnazione

- Nell'operatore di assegnazione l'operando destro può essere sia un l-value (una variabile) che un r-value (un'espressione, una chiamata a funzione,una costante numerica) ,mentre l'operando sinistro deve essere necessariamente un l-value (una variabile)

  a=3; //ok

  a=func(3.);//ok

  3=a; // no

- Le assegnazioni hanno priorità tra le più basse e sono valutate da destra a sinistra:

  a=b+3;//prima calcola b+3, poi il valore è copiato in a

  a=++j;//prima incrementa j, poi il valore è copiato in a

  a=j++;// ?

Operatore di Assegnazione

- Si possono fare assegnamenti multipli in cascata:

```
a=b=c=3;
a=b=3=c;        // è valido?
a=(b=5)=c=3;    // è valido? quanto vale a?
```

- Si possono combinare assegnamenti e operatori binari (notazione compatta)

```
c+=b;// prima viene sommato b a c, e il risultato è copiato in c
a*=3;// equivale a   a=a*3;
```

- Un assegnamento può essere parte di un'espressione aritmetica o condizionale:

```
a=b*(c=10);// c=10; a=b*c
a=b*c=10;// è valido?
```

Tipi Predefiniti in C++ - conversioni

Conversioni implicite in espressioni e assegnazioni (eseguite automaticamente dal compilatore):

x=espressione;

- Il valore finale di un'espressione mista ha sempre il tipo massimo tra quelli che la compongono. Gerarchia di *promozione:*

  int < unsigned int < long < unsigned long < float <double < long double

  - char, short, bool sono promossi a int
  - bool è un tipo intero che viene promosso a 0/1 a seconda se false/true

- Ulteriore (eventuale) conversione, determinata ora dal tipo della variabile x:

espressione → x

Dato che prevale il tipo di x, l'operazione può risultare anche in una perdita di informazione *(loss of data)*,come ad esempo un troncamento (warning del compilatore)

Tipi Predefiniti in C++ - conversioni

L'utente può poi operare **conversioni esplicite**:

```
x = (float) i;          // cast in C++, notazione C
x = float(i);           // cast in C++, notazione funzionale
```

Consentito per tutti i tipi nativi. I due tipi di cast sono equivalenti (anche se il casting in notazione funzionale non è consentito per convertire puntatori)

Tipi Predefiniti in C++- conversioni esplicite

Qual è il valore di “n” al termine della seguente espressione?
int n = int(1. + 2 / 3 / 0.5 - 4./3);

Test esame 14/07/2014

A) 1
B) - 0.33
C) 0
D) 0.66

Quale sarà il valore di x dopo la seguente linea di codice:
float x = 1/(1/3);

Test esame 10/06/2014

A) 0.
B) 3.
C) la linea di codice produrrà questo errore in esecuzione: "Error:operator '/' divided by zero"
D) 0.33333

Operatore Ternario ?:

L'operatore condizionale "?:" è l'unico operatore ternario in C/C++

expr1 ? expr2 : expr3

Che restituisce il valore dell'espressione expr2 se expr1 è vera, altrimenti restituisce expr3. Ad esempio si può scrivere in maniera molto compatta la funzione che ritorna il massimo fra due numeri: `double max(double a, double b){return a>b?a:b;}`

Inoltre, ha la particolarità di essere l'unico operatore che può ritornare un l-value, a patto che a e b siano variabili e non espressioni:

```
( expr1 ? a : b) = c;
```

Memorizza il valore di c (che può essere anche un'espressione):

- **in a**, se expr1 è vera,
- altrimenti memorizza il valore di c **in b**

Istruzioni di Controllo del Flusso

istruzioni che modificano l'esecuzione sequenziale di un programma:

Strutture condizionali:

- if
```
if(n>10){cerr <<"too many elements, exit";}
```
- if-else
```
if(x==y){cout <<"x and y are the same"<<endl;}
else {cout << "x and y are different"<< endl;}
```
- if -else if-else
```
if(x==y){cout <<"x and y are the same"<<endl;}
else  if(x>y) {cout << "x is greater than y "<< endl;}
else  {cout << "x is smaller than y "<< endl;}
```
- switch
```
switch(index){}// il valore di index seleziona il caso
```

Cicli:

- for
```
for(int i=0; i<10;i++)a[i]=i;
```
- while
```
int i=0; while(i<10){a[i]=i; i++;}
```
- do-while
```
do{ y=y-1;}while(y<0);
```

if statement

Un semplice codice di generazione simula l'occorrenza di tre casi distinti:

```
float x;
for(int i=0; i<ntrials;i++){
x=random();
if(x<0.3) n1++;
else if(x<0.2)n2++;
else n3++;
}
```

Test esame 18/09/2014

Dove x è un numero random uniformemente distribuito fra 0 e 1. In media, quali saranno le percentuali di casi nelle categorie corrispondenti ai contatori n1,n2,n3, rispettivamente?

if statement

- Usare le parentesi graffe anche quando non strettamente necessarie per essere sicuri del corretto comportamento del blocco:

```
if(i!=0)   // protezione contro la divisione per zero, funziona?
  a++;
  a/=i;
```

- In assenza di parentesi l'else si associa automaticamente all'if più vicino, e si possono generare errori di logica:

```
int a=2,i=0;
if(i==0)
    if(a<0) cout<< " i è zero, a = "<< a << endl;
else a/=i;       // protezione contro la divisione per zero, funziona?
```

for statement

```
for(expr1;expr2;expr3){
    statements;
 }
```

expr1:inizializzazione
expr2: condizione
expr3: modifica

```
expr1;
while(expr2){
        statements;
        expr3;
    }
```

- break e continue interrompono il ciclo e passano all'iterazione successiva, rispettivamente:

```
for(int i=0;i<10;i++){
   if (i>7)break;
   if(i<4)continue;
   cout << "i = " << i << endl;
} // stampa sullo standard output  4,5,6,7
```

For vs while e do-while

I costrutti for, while e do-while sono funzionalmente identici:

- L'uso del for è raccomandato se il numero di volte che bisogna ripetere il il corpo di istruzioni su cui si esegue il ciclo, nel momento in cui inizia l'esecuzione del comando, è noto
- while e do-while: si usano quando il numero di volte che bisogna ripetere il corpo di istruzioni dipende da un valore letto/calcolato all'interno del corpo

- while(condizione) {.....}; esegue il blocco finchè la condizione rimane vera. Può non venire mai eseguito. Per eseguire l'insieme di istruzioni nel blocco almeno una volta, si può usare il costrutto do-while;

```
int n=0;
do{ cout <<" please input a two-digit integer number:"<< endl; cin>>n;
} while(!(n<10 || n>99));
```