Riepilogo sulla programmazione

Sommario

- Gli algoritmi
- Le strutture di selezione e iterazione
- Le funzioni
- La visibilità delle variabili

Gli algoritmi

- Il termine algoritmo deriva dal nome di un matematico arabo Abu Abdullah Muhammad bin Musa al-Khwarizmi (IX sec d.C.)
- Inizialmente la parola algorism veniva usata per indicare le regole del calcolo aritmetico con i numeri arabi.

Definizione moderna:

- Un algoritmo è una successione ordinata di istruzioni (o passi) che definiscono le operazioni da eseguire su dei dati per risolvere una classe di problemi

Proprietà degli algoritmi

- Perché una sequenza di istruzioni sia un algoritmo devono essere soddisfatti i seguenti requisiti (informali):
  - Finitezza
  - Generalità
  - Non ambiguità

Finitezza

- Il numero di istruzioni è finito
- Ogni istruzione è eseguita in un intervallo finito di tempo
- Ogni istruzione è eseguita un numero finito di volte

Generalita'

- Un algoritmo deve fornire la soluzione per una classe di problemi
  - dato un insieme di definizione o *dominio*
  - dato un insieme di arrivo o *codominio*
  - l'algoritmo può operare su tutti i dati appartenenti al dominio per fornire una soluzione all'interno dei codominio

Non ambiguita'

- le istruzioni sono definite in modo univoco
- non ci sono paradossi, contraddizioni, ambiguità
- il risultato dell’algoritmo è identico indipendentemente da chi lo sta eseguendo

Programmazione

- Lo scopo della programmazione è quello di definire un programma
- Programma=descrizione di un algoritmo in forma comprensibile per un elaboratore automatico
- Il programma è descritto facendo ricorso ad un linguaggio di programmazione o linguaggio formale

Linguaggio Formale

- Un linguaggio formale è costituito da strutture linguistiche per descrivere gli algoritmi in modo preciso e sintetico
- Un linguaggio formale si differenzia dal *linguaggio naturale* che contiene ambiguità e ridondanze

Linguaggio Formale

- Le proposizioni usate da un linguaggio formale descrivono due entità:
  - le operazioni che devono essere eseguite (istruzioni)
  - gli oggetti (dati) sui quali si devono eseguire le operazioni

- *In questa introduzione tralasceremo di parlare dei dati, ovvero delle costanti, delle variabili e dei tipi*
- *Questi concetti insieme a quelli di assegnazione e di operatore sono dati per già acquisiti*

Categorie istruzioni

- istruzioni operative
- istruzioni di selezione
- istruzioni di salto
- istruzioni di inizio/fine esecuzione
- istruzioni di ingresso/uscita

Istruzioni operative

- Istruzioni che producono un risultato se eseguite
- Ne fanno parte le operazioni aritmetiche e le assegnazioni

Istruzioni di selezione

- Istruzioni che controllano il verificarsi di condizioni specificate e che in base al risultato determinano quale istruzione eseguire

- Es.

  se … allora …altrimenti

Istruzioni di salto

- Istruzioni che alterano il normale ordine di esecuzione delle istruzioni di un algoritmo, specificando esplicitamente quale sia la successiva istruzione da eseguire
- Si distinguono istruzioni di salto condizionato o incondizionato
- Per quelle condizionate il salto è subordinato al verificarsi di una condizione specificata, per quelle incondizionate il salto è eseguito ogni volta che l'istruzione viene eseguita.

Istruzioni di inizio/fine

- Indicano quale istruzione dell'algoritmo debba essere eseguita inizialmente e quale determini la fine dell'esecuzione

Istruzioni di ingresso/uscita

- Istruzioni che indicano una trasmissione di dati o messaggi fra l'algoritmo e tutto ciò che è esterno all'algoritmo
- si dicono di ingresso o lettura quando l'algoritmo riceve dati dall'esterno
- si dicono di uscita o scrittura quando i dati sono comunicati dall'algoritmo all'esterno

Istruzioni in C/C++

- Quale e' la sintassi e quali sono le parole chiave utilizzate dal C/C++ per indicare le varie operazioni?

Input e Output in C/C++

- Useremo spesso le funzioni di I/O negli esempi
- A livello introduttivo basta sapere che:

Output

- in C: printf(“testo %d”, num);
- in C++: cout<<“testo”<<num;

Input

- in C: scanf(“%d”,&num);
- in C++: cin >> num;

La istruzione di selezione (if)

- La istruzione di selezione serve per decidere se intraprendere una azione o quale azione intraprendere tra diverse alternative
- in linguaggio naturale:
  - se si verifica X allora esegui Y

```
if(condizione1) {
   istruzione1;
   istruzione2;
   …
 }
```

La istruzione di selezione (if else)

- in linguaggio naturale:
  - se si verifica X allora esegui Y altrimenti esegui Z

```
if(condizione1) {
   istruzione1;
   istruzione2;
   …
 }
 else {
   istruzione3;
   istruzione4;
   ...
 }
```

La istruzione di selezione (if, else if, else)

- in linguaggio naturale:
  - se si verifica X allora esegui Y altrimenti se si verifica X2 allora esegui Y2 altrimenti esegui Y3

```
if(condizione1) {
   istruzione1;
 }
 else if(condizione2){
   istruzione2;
 }
 else {
   istruzione3;
 }
```

Le strutture iterative (while)

- Le istruzioni di iterazione consentono di eseguire ripetutamente una azione fintanto che una condizione specificata rimane vera
- In linguaggio naturale:

  finché la condizione1 è vera continua a eseguire un insieme di istruzioni

```
while(condizione1) {
   istruzione1;
   istruzione2;
   …
 }
```

Esempio

```
int main()
{
    int total,                          // sum of grades
        gradeCounter,                   // number of grades entered
        grade,                          // one grade
        average;                        // average of grades

   // initialization phase
   total = 0;                           // clear total
   gradeCounter = 1;                    // prepare to loop
   while ( gradeCounter <= 10 ) {       // loop 10 times
      cout << "Enter grade: ";          // prompt for input
      cin >> grade;                     // input grade
      total = total + grade;            // add grade to total
      gradeCounter = gradeCounter + 1;  // increment counter
   }

   // termination phase
   average = total / 10;                // integer division
   cout << "Class average is " << average << endl;

   return 0;   // indicate program ended successfully
}
```

Le strutture iterative (do while)

- In linguaggio naturale:

  esegui una volta l'istruzione1 e continua ad eseguirla finché la condizione1 è vera

```
do{
    istruzione1;
    istruzione2;
    …
 } while(condizione1);
```

Le strutture iterative (for)

- Se l'iterazione è controllata da una variabile contatore si può usare in modo compatto l'istruzione for

```
for(i=0;i<10;i=i+1){
   istruzione1;
   istruzione2;
   …
 }
```

Le strutture iterative (for)

- L'istruzione for si compone di tre parti:

```
for(init;condition;increment)
```

Esempio:

```
for(i=0;i<10;i=i+1)
```

Dove:

`init:i=0`

e' la parte di inizializzazione della variabile contatore

`condition:i<10`

e' la condizione che deve essere verificata per proseguire nella iterazione

`increment:i=i+1`

e' la modifica della variabile contatore

Esempio

```
int main()
{
   int sum = 0;

   for ( int number = 2; number <= 100; number += 2 )
      sum += number;

   cout << "Sum is " << sum << endl;

   return 0;
}
```

Equivalenza for-while

- L'istruzione *for* equivale ad una specifica struttura *while* in cui si fornisce l'inizializzazione e una istruzione di aggiornamento della variabile contatore

```
for(i=0;i<10;i=i+1){
   istruzione1;
 }

i=0;
while(i<10){
   istruzione1;
   i=i+1;
 }
```

Sottoprogrammi

- Per risolvere un problema si può scomporlo in problemi più semplici e comporre le loro soluzioni
- questo viene realizzato tramite l'uso di sottoprogrammi chiamati *funzioni*

Funzioni

- Una funzione è un sottoprogramma che può prendere delle variabili in ingresso e può restituire dei valori in uscita

- In C/C++ le funzioni hanno questa sintassi:

```
tipo nome(tipo variabile1, tipo variabile2){
   /*corpo della funzione*/
   return risultato;
}
```

- ovvero:
  - è sempre presente un tipo di ritorno (eventualmente void)
  - può essere presente una o più variabili di tipo specificato
  - può essere presente l'istruzione di restituzione

Funzioni

- Esempi:

```
int somma(int a, int b){
  int c;
  c=a+b;
  return c; //oppure return a+b;
}

int max(int a, int b){
   if(a>b)return a;
   else return b;
}
```

Funzioni

- Nota: è un errore scrivere la seguente funzione:

```
int funzione(int a,b, float c)
```

- perché ogni parametro di ingresso deve avere la sua specifica di tipo.
- Si deve scrivere:

```
int funzione(int a, int b, float c)
```

Funzioni

- Le funzioni devono essere "*dichiarate*" prima di poter essere utilizzate

```
#include<stdio.h>
int somma(int,int);

void main(){
  //…qualcosa
  ris=somma(x,y);
  //…qualcosa
}

int somma(int a, int b){
  return a+b;
}
```

Funzioni

- La dichiarazione di una funzione avviene tramite la scrittura del suo *prototipo*
- Il prototipo di una funzione ha la seguente sintassi:

```
tipo nome(tipo, tipo, ...);
```

- Nota: non è necessario specificare il nome delle variabili in ingresso

- Nota: la dichiarazione termina con un ';'

Funzioni

- Una volta dichiarate le funzioni si possono utilizzare nel main o nelle altre funzioni

```
int f1(int);
int f2(int);

void main(){
   int x,y;
   x=1;
   y=f1(x);
}

int f1(int a){return f2(a)*f2(a);}
int f2(int a){return a+1;}
```

Funzioni

- Una funzione può anche non ritornare alcun valore
- in questo caso si dichiara che il tipo di ritorno è *void* e non si include l'istruzione *return*

```
void print_ciao(int num){
   for(i=0;i<num;i++) cout<<"Ciao\n";
}
```

Funzioni

- Le funzioni possono anche non prendere alcun parametro in ingresso
- in questo caso si usa void come parametro per la funzione

```
void print_ciao(void){
  cout<<"Ciao\n";
}
```

Funzioni

- Si possono avere anche più istruzioni *return* all'interno della stessa funzione:

```
int max(int a, int b){
 if(a>=b) return a;
 if(b>a) return b;
}
```

Esempio

```
#include <iostream>
int maximum( int, int, int ); // function prototype

int main()
{
   int a, b, c;

   cout << "Enter three integers: ";
   cin >> a >> b >> c;

   // a, b and c below are arguments to
   // the maximum function call
   cout << "Maximum is: " << maximum( a, b, c ) << endl;

   return 0;
}
```

Esempio

```
// Function maximum definition
// x, y and z below are parameters to
// the maximum function definition
int maximum( int x, int y, int z )
{
   int max = x;

   if ( y > max )
      max = y;

   if ( z > max )
      max = z;

   return max;
}
```

Nota

- In un programma complesso il *main* dovrebbe essere semplice e contenere solo le chiamate alle varie funzioni che eseguono le parti differenti del programma
- un buon main e' costituito da una decina di linee

Regole di visibilità

- Una variabile esiste e può essere richiamata solo all'interno del blocco/funzione dove è stata definita
- Una variabile ha visibilità *globale* quando è definita al di fuori del blocco *main*
- Una variabile globale è visibile all'interno di una qualunque funzione

- ATTENZIONE: una variabile locale con lo stesso nome di una variabile globale ha precedenza su questa ultima e la nasconde

Regole di visibilità

```
int f(int);
int a; /*variabile globali*/

void main(){
int b,e; /*variabili locali*/
b=2; a=3; e=f(b);/* "a" è visibile, "d" no*/
}

int f(int c){
int d; /*variabile locale*/
d=c+a;/* "a" è visibile; "b,e" no */
return d;
}
```

Nota

- Si sconsiglia di utilizzare variabili globali per passare informazioni alle funzioni
- E' meglio passare in modo esplicito le informazioni tramite le variabili di ingresso
- Il pericolo potenziale è
  - perdere il controllo di quale funzione modifichi la variabile globale
  - non sapere che una funzione abbia bisogno di una particolare variabile globale

Static

- Le variabili dichiarate come static hanno sempre visibilità a livello di blocco o di funzione ma conservano il loro valore anche fra chiamate successive

- si dichiarano premettendo la parola chiave static

```
static int a;
```

- se non sono inizializzate esplicitamente sono poste automaticamente a 0

Esempio di scope

```
void a( void );   // function prototype
void b( void );   // function prototype
void c( void );   // function prototype

int x = 1;        // global variable

int main()
{
   int x = 5;   // local variable to main

   cout << "local x in outer scope of main is " << x << endl;
   {             // start new scope
     int x = 7;
     cout << "local x in inner scope of main is " << x << endl;
   }             // end new scope
   cout << "local x in outer scope of main is " << x << endl;
   a();          // a has automatic local x
   b();          // b has static local x
   c();          // c uses global x
   a();          // a reinitializes automatic local x
   b();          // static local x retains its previous value
   c();          // global x also retains its value
   cout << "local x in main is " << x << endl;
return 0;
}
```

Esempio

```
void a( void )
{
   int x = 25;  // initialized each time a is called
   cout << endl << "local x in a is " << x
        << " after entering a" << endl;
   ++x;
   cout << "local x in a is " << x
        << " before exiting a" << endl;
}
void b( void )
{
    static int x = 50;  // Static initialization only
                        // first time b is called.
    cout << endl << "local static x is " << x
         << " on entering b" << endl;
    ++x;
    cout << "local static x is " << x
         << " on exiting b" << endl;
}
void c( void )
{
   cout << endl << "global x is " << x
        << " on entering c" << endl;
   x *= 10;
   cout << "global x is " << x << " on exiting c" << endl;
}
```