Riepilogo II

Sommario

- Il preprocessore
- I puntatori
- Gli array
- I puntatori a funzione

Dividere un programma

- Se il programma diventa complesso conviene dividerlo in più parti
- Ogni parte dovrebbe contenere un gruppo di funzioni che sono tra loro correlate, ovvero che svolgono dei compiti simili o in relazione tra loro
- Ad es. si potrebbero mettere tutte le funzioni che si occupano della stampa dei dati in un file a se stante
- Inoltre utilizzare librerie di funzioni predefinite permette di non dover *reinventare la ruota* ogni volta

Il preprocessore

- Riunire parti separate di codice è una operazione che si colloca ad un diverso livello di astrazione rispetto alle istruzioni che sono rese disponibili da un linguaggio di programmazione
- queste operazioni vengono eseguite in una fase precedente alla compilazione e vengono chiamate direttive di preprocessore
- è possibile eseguire operazioni del tipo:
  - includere file, definire costanti simboliche e macro, effettuare compilazione condizionale del codice e esecuzione condizionale delle direttive al preprocessore

Il preprocessore

- Le direttive di preprocessore iniziano con un carattere # seguito da un comando
- la direttiva per l'inclusione è: #include <nome> o #include "nome"
- la differenza sta nella posizione nel file system in cui il preprocessore cercherà i file da includere:
  - in directory predefinite per <>
  - nella stessa directory del file da compilare per ""
- Includere un file significa sostituire la direttiva #include con una copia del file da includere

File di intestazione

- Si consideri il seguente programma:

```
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
   a();         // a has automatic local x
   b();         // b has static local x
   c();         // c uses global x
   cout << "local x in main is " << x << endl;
   return 0;
}
```

File di intestazione

- Si consideri il seguente programma:

```
void a( void )
{
   int x = 25;  // initialized each time a is called
    ++x;
}

void b( void )
{
    static int x = 50;  // Static initialization
    ++x;
}

void c( void )
{
    x *= 10;
}
```

File di intestazione

- Lo si può dividere in tre file nel modo seguente:

```
//file: my_functions.h
void a( void );   // function prototype
void b( void );   // function prototype
void c( void );   // function prototype

int x = 1;      // global variable
```

```
//file: main.cc
#include “my_functions.h”

int main()
{
   int x = 5;   // local variable to main
   a();         // a has automatic local x
   b();         // b has static local x
   c();         // c uses global x
cout << "local x in main is " << x << endl;
   return 0;
}
```

```
//file: my_functions.cc
#include “my_functions.h”

void a( void )
{
        int x = 25;  // initialized each time a is called
        ++x;
}

void b( void )
{
        static int x = 50;  // Static initialization
        ++x;
}

void c( void )
{
        x *= 10;
}
```

I puntatori

- A differenza delle variabili che contengono un valore specifico, il valore di una variabile di tipo puntatore è un indirizzo di memoria

- Una variabile puntatore contiene in genere l'indirizzo di una altra variabile che a sua volta contiene un valore

Dichiarazione

- Le variabili di tipo puntatore si dichiarano come:

```
int * varPtr;
```

- la dichiarazione avviene tramite l'uso dell'operatore *

- Nota: è un errore scrivere:

```
int * var1Ptr, var2Ptr, var3Ptr;
```

  infatti in questo modo si dichiara solo var1Ptr come puntatore e var2Ptr e var3Ptr come interi, ovvero il compilatore interpreta la dichiarazione come

```
int *var1Ptr;
int var2Ptr, var3Ptr;
```

Operatore di indirizzo

- Per ottenere l'indirizzo di una variabile si usa l'operatore di indirizzo &

```
int y=5;
int * ptr;
ptr=&y;
```

Operatore di risoluzione del riferimento

- Per ottenere il valore contenuto nella cella di memoria dato un indirizzo si usa l'operatore *

```
int x,y=5;
int * ptr;
ptr=&y;
x=*ptr;
```

Inizializzazione

- Un puntatore dovrebbe essere sempre inizializzato!
- Si può inizializzare un puntatore tramite l'indirizzo di una variabile ...
- ...oppure si può inizializzare un puntatore a 0 o NULL (costante simbolica equivalente a 0) per indicare che un puntatore non punta a nessun dato

- Nota: provoca un errore al tempo di esecuzione dereferenziare un puntatore nullo, ovvero eseguire:

  ```
  int * ptr=NULL;
  int x=*ptr;
  ```

Passaggio di parametri per riferimento

- Se si utilizza una variabile di tipo puntatore come argomento di una funzione allora si ha la possibilità di modificare il valore del dato puntato

- questo risulta utile quando una funzione deve modificare una struttura dati di grandi dimensioni (come ad esempio un vettore, una struttura o un oggetto)

Esempio

```
void cubeByReference( int * );   // prototype

int main()
{
   int number = 5;
   cout << "The original value of number is " << number;
   cubeByReference( &number );
   cout << "\nThe new value of number is " << number << endl;
   return 0;
}

void cubeByReference( int *nPtr )
{
   // cube number in main
   *nPtr = (*nPtr) * (*nPtr) * (*nPtr);
}
```

Aritmetica dei puntatori

- I puntatori possono essere utilizzati come operandi in espressioni aritmetiche, di assegnamento e di confronto
- E' possibile utilizzare solo un insieme limitato di operatori
  - incremento/decremento: ++ --
  - addizione o sottrazione di un intero: + - += -=
  - addizione o sottrazione di un altro puntatore

Nota

- Attenzione:

```
int val=5;
int * vPtr=&val;
vPtr+=2;
```

- se val è stata allocata nella cella di memoria 2000 allora vPtr=&val vale 2000, ma vPtr+=2 non vale 2002 ma 2004 o 2008 a seconda se gli interi sono rappresentati con 2 o 4 byte (dipende dalla architettura della macchina)

- in realtà non si lavora mai con gli indirizzi espliciti delle variabili

Array e puntatori

- In C/C++ gli array e i puntatori sono strettamente correlati
- infatti
  - int a[5]={10,20,30,40,50};
    int * ptr1=a;
    int * ptr2=&a[0]; //ptr1,ptr2 e a puntano alla stessa cella
    int val1=*(ptr1 + 1); //val1=20
    int val2=ptr1[3];//val2=40
- il nome di una variabile array è un puntatore *costante* al primo elemento dell'array
- la notazione a[n] o *(a+n) è equivalente

Esempio

```
int main()
{
   int b[] = { 10, 20, 30, 40 }, i, offset;
   int *bPtr = b;   // set bPtr to point to array b

   cout << "Array b printed with:\n"
        << "Array subscript notation\n";
   for ( i = 0; i < 4; i++ )
      cout << "b[" << i << "] = " << b[ i ] << '\n';

   cout << "\nPointer/offset notation where\n"
        << "the pointer is the array name\n";
   for ( offset = 0; offset < 4; offset++ )
      cout << "*(b + " << offset << ") = "
           << *( b + offset ) << '\n';

   cout << "\nPointer subscript notation\n";
   for ( i = 0; i < 4; i++ )
      cout << "bPtr[" << i << "] = " << bPtr[ i ] << '\n';

   cout << "\nPointer/offset notation\n";
   for ( offset = 0; offset < 4; offset++ )
      cout << "*(bPtr + " << offset << ") = "
           << *( bPtr + offset ) << '\n';

 return 0;
}
```

Puntatori a funzione

- Così come il nome di un array è in realtà un puntatore all'inizio dell'array, altrettanto il nome di una funzione è in realtà un puntatore che contiene l'indirizzo di partenza del codice della funzione

- i puntatori a funzioni possono solo essere
  - inizializzati
  - assegnati
  - dereferenziati

Puntatori a funzione

- Per una funzione come: `char f(int, double);`
- si deve utilizzare un puntatore a funzione dichiarato nel modo seguente: `char (*nome)(int, double);`
- cioè si esplicita il tipo di ritorno, il fatto che è un puntatore e i tipi dei parametri;

- Nota: le parentesi servono per impedire che il compilatore interpreti `char *nome(int, double);` come `char* nome(int, double);` e cioè come una funzione che restituisce un puntatore a char

Come chiamare una funzione tramite il puntatore a funzione

- Analogamente al caso dei puntatori e della loro relazione con gli array è possibile dereferenziare tramite l'operatore * un puntatore a funzione per ottenere il contenuto puntato, ovvero la funzione.

```
int func(int,int);
int (*funcPtr)(int, int);
funcPtr=func;
int a,b; a=1;b=2;
int x=(*funcPtr)(a,b);
```

Come chiamare una funzione tramite il puntatore a funzione

- Analogamente al caso dei puntatori e della loro relazione con gli array è possibile accedere al contenuto utilizzando la sintassi ordinaria (infatti il nome di una funzione è un puntatore a funzione)

```
int func(int,int);
int (*funcPtr)(int, int);
funcPtr=func;
int a,b; a=1;b=2;
int x=funcPtr(a,b);
```

Esempio

```
void swap_greater(int *, int *);
void swap_lesser(int *, int *);

int main()
{
   int a[] = { 10, 20, 30, 40 };
   int b[] = { 50, 60, 70, 80 };
   void (*func)(int *, int *);
    int i;

   func=swap_greater;
   for ( i = 0; i < 4; i++ )
       (*func)(&a[i],&b[i]);

   func=swap_lesser;
   for ( i = 0; i < 4; i++ )
       func(&a[i],&b[i]);

   return 0;
}
```

Esempio

```
void swap_greater(int *a, int *b){
   int tmp;
   if(*a < *b){
        tmp=*a;
        *a=*b;
        *b=tmp;
   }
}

void swap_lesser(int *a, int *b){
   int tmp;
   if(*a > *b){
        tmp=*a;
        *a=*b;
        *b=tmp;
   }
}
```

Uso dei puntatori a funzione

- Si possono scrivere algoritmi più flessibili con una struttura base che rimane invariante ma che adoperano funzioni diverse per compiti diversi
- Nota: in C++ i meccanismi di overloading e template potenziano questa tecnica

- Si possono creare dei vettori di puntatori a funzione e usare la funzione opportuna tramite un indice numerico