Laboratorio di Sistemi Operativi

Modulo 1: Il linguaggio C

Renzo Davoli
Alberto Montresor

Copyright © 2002-2005 Renzo Davoli, Alberto Montresor
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found at: http://www.gnu.org/licenses/fdl.html#TOC1

Storia del linguaggio C

- **K&R C**
  - Creato da Dennis Ritchie presso gli AT&T Labs nel 1972
  - Creato originariamente per progettare e supportare il sistema operativo Unix
  - Molto semplice: ci sono solo 27 keyword nel C originale
  - K&R dal nome di Kernighan e Ritchie's, autori di "The C Programming Language", libro fondamentale per il C
- **ANSI Standard C**
  - Nel 1983, l'ANSI (American National Standards Institute) formò un comitato per definire una versione standard del C

© 2002-2005 Renzo Davoli, Alberto Montresor

Testi

- **Consigliato**
  - *The C Programming Language*, Second Edition
    Brian W. Kernighan and Dennis M. Ritchie.
    Prentice Hall, Inc., 1988.
- **Testi nella pagine del corso:**
  - Ansi C for programmers on Unix systems, di Tim Love
  - C Programming Notes, di Steve Summit
  - Programming in C - UNIX System Calls and Subroutines using C, di Dave Marshall

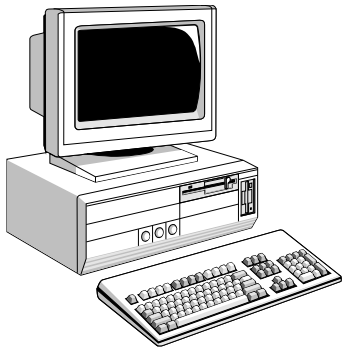
© 2002-2005 Renzo Davoli, Alberto Montresor

Caratteristiche del C

- **Il C è un linguaggio per "veri" programmatori...**
- **Il C è potente ed efficiente**
  - è possibile raggiungere la stessa efficienza dell'assembly
  - chiamate di sistema e puntatori vi permettono di fare più o meno tutto quello che può fare il linguaggio assembly
- **Il C è un linguaggio strutturato**
  - il codice può essere scritto e letto molto più facilmente
- **Il C è portabile**
  - esistono compilatori ANSI C per ogni piattaforma

© 2002-2005 Renzo Davoli, Alberto Montresor

Il ciclo di sviluppo del C



Edit Program

Source Code

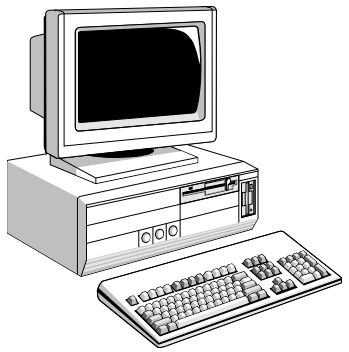
Compile

Object code

Librerie

Link Object Code

Executable



© 2002-2005 Renzo Davoli, Alberto Montresor

Il ciclo di sviluppo del C

- **Il compilatore trasforma il vostro codice sorgente in codice oggetto**
  - linguaggio macchina per la macchina che state utilizzando
- **Si noti la differenza con Java:**
  - Il compilatore Java crea bytecode
  - Il bytecode viene eseguito da una JVM, ed è indipendente dalla macchina
- **L'operazione di linking genera un file eseguibile**

© 2002-2005 Renzo Davoli, Alberto Montresor

Hello World

- Il programma più noto:

```
#include <stdio.h>
int main ( )
{
  printf ("Hello, World!\n");
  return 0;
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Hello World

- **Funzione `main()`**
  - deve esistere sempre
  - ritorna un intero (**`int`**)
  - E' la funzione che viene eseguita quando un programma è eseguito
  - Ritorna lo "status" con lo statement **`return`**
    (0 indica successo)
- **Direttiva `#include`**
  - Chiede al *preprocessore C* (cpp) di includere il file specificato
  - Questi file sono detti *header file* e hanno estensione .h
  - <> dice al preprocessore di cercare questo file in un insieme di directory di default

© 2002-2005 Renzo Davoli, Alberto Montresor

Per compilare il vostro programma

- **Per compilare da codice sorgente a codice oggetto**
  - `gcc -c hello.c`
  - crea il file `hello.o`
- **Per fare il linking in un file eseguibile**
  - `gcc hello.o –o hello`
  - crea il file eseguibile `hello`
- **Per fare i due passi assieme (senza creare un file .o):**
  - `gcc hello.c -o hello`
- **Per eseguire il programma:**
  - `./hello`
    `Hello World!`

© 2002-2005 Renzo Davoli, Alberto Montresor

Messaggi di errore

- **Inseriamo un errore:**
  - supponiamo di dimenticare il ; dopo il **`printf`**
- ```
  gcc hello.c -o hello
  "hello.c", line 5: syntax error before or at: return
  gcc: acomp failed for hello.c
  ```
- **Nota:**
  - il compilatore rileva l'errore al primo token non appropriato
  - in questo caso, lo statement **`return`**
  - provate sempre a correggere gli errori a partire dal primo; gli altri possono sparire

© 2002-2005 Renzo Davoli, Alberto Montresor
  ```

Primo esempio

```
#include <stdio.h>
int sumup(int n);
int main()
{
  int n;
  int result;
  printf("Enter a number:");
  scanf("%d",&n);
  result = sumup(n);
  printf("Result = %d\n",
    result);
  return 0;
}
```

```
/* Sum number between 1
   and n */
int sumup(int n)
{
  int i;
  int sum = 0;
  for(i=1;i<=n;i++)
  {
    sum=sum+i;
  }
  return sum;
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Componenti di un programma

- **Dichiarazione di variabili**
  - In C, le variabili devono essere dichiarate prima di essere utilizzate
  - Le dichiarazioni specificano il tipo e il nome di una variabile
  - Esempio: `int i;`
- **Statement**
  - Esempi:

```
printf("Enter a number:");
scanf(...);
result = sumup(n);
return 0;
```

  - Il ; è un terminatore di statement; deve essere aggiunto alla fine di ogni statement

© 2002-2005 Renzo Davoli, Alberto Montresor

Componenti di un programma

- **Prototipi di funzione**
  - Dicono al compilatore quale sarà il "formato" di una funzione
  - Devono apparire prima che una funzione sia utilizzata
  - Un prototipo di una funzione è diverso dalla sua definizione:
    - il primo descrive l' "interfaccia" della funzione
    - la seconda descrive l' "implementazione" della funzione
- **Definizione di funzione**
  - Una funzione è una sezione indipendente e autocontenuta di codice
  - il C contiene una libreria di funzioni standard
  - Esempi: **`printf()`** e **`scanf()`**

© 2002-2005 Renzo Davoli, Alberto Montresor

Componenti di un programma

- **Commenti:**
  - `/*  Questo è un commento */`
  - utilizzateli!
  - fare il debugging programmi in C scritti da altri è una esperienza terribile se i programmi non sono ben documentati
- **Parantesi graffe**
  - come Java
  - sono chiamati blocchi e sono il costrutto primario di raggruppamento

© 2002-2005 Renzo Davoli, Alberto Montresor

Variabili

- **Dichiarazione di variabili**
  - forma generica:
    - `typename varname1, varname2, ...;`
    - dichiara le variabili `varname1, varname2, ...` e gli associa il tipo `typename`
  - forma suggerita:
    - `typename varname; /* Comment */`
    - una variabile per linea e per tipo, associata ad un commento;
    - facilita la leggibilità
    - semplifica la manutenzione del codice (e.g., cambiare il tipo di una variabile)

© 2002-2005 Renzo Davoli, Alberto Montresor

Variabili

- **Concetti fondamentali**
  - Le variabili dichiarate fuori da ogni blocco (funzione) sono *variabili globali*
  - Le variabili dichiarate all'interno di un blocco sono *variabili locali* a quel blocco
- **Visibilità:**
  - Le variabili locali sono visibili solamente all'interno del blocco in cui sono dichiarate
  - Le variabili globali sono visibili a partire dal punto in cui sono dichiarate, per tutto il resto del file
  - Una dichiarazione di variabile locale nasconde le dichiarazioni con lo stesso nome definite all'esterno del blocco

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio

```
int a=1;
void test(){
  printf("%d\n",a);
}
int main( )
{
  int a=2;
  {
    int a=3;
    printf("%d\n",a);
  }
  printf("%d\n",a);
  test();
}
```

Qual è l'output di questo programma?

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio

```
int a=1;
void test(){
  printf("%d\n",a);
}
int main( )
{
  int a=2;
  {
    int a=3;
    printf("%d\n",a);
  }
  printf("%d\n",a);
  test();
}
```

Qual è l'output di questo programma?

3

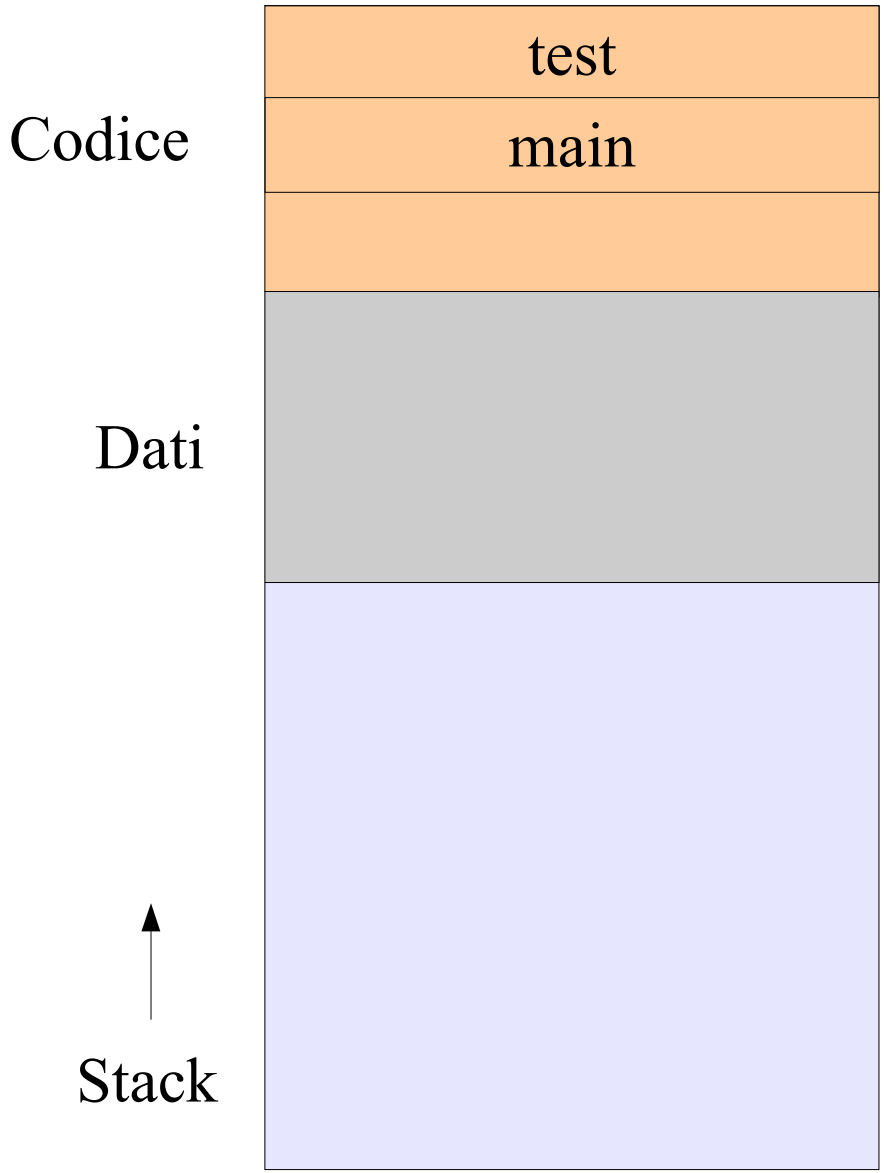
2

1

© 2002-2005 Renzo Davoli, Alberto Montresor

Layout di memoria

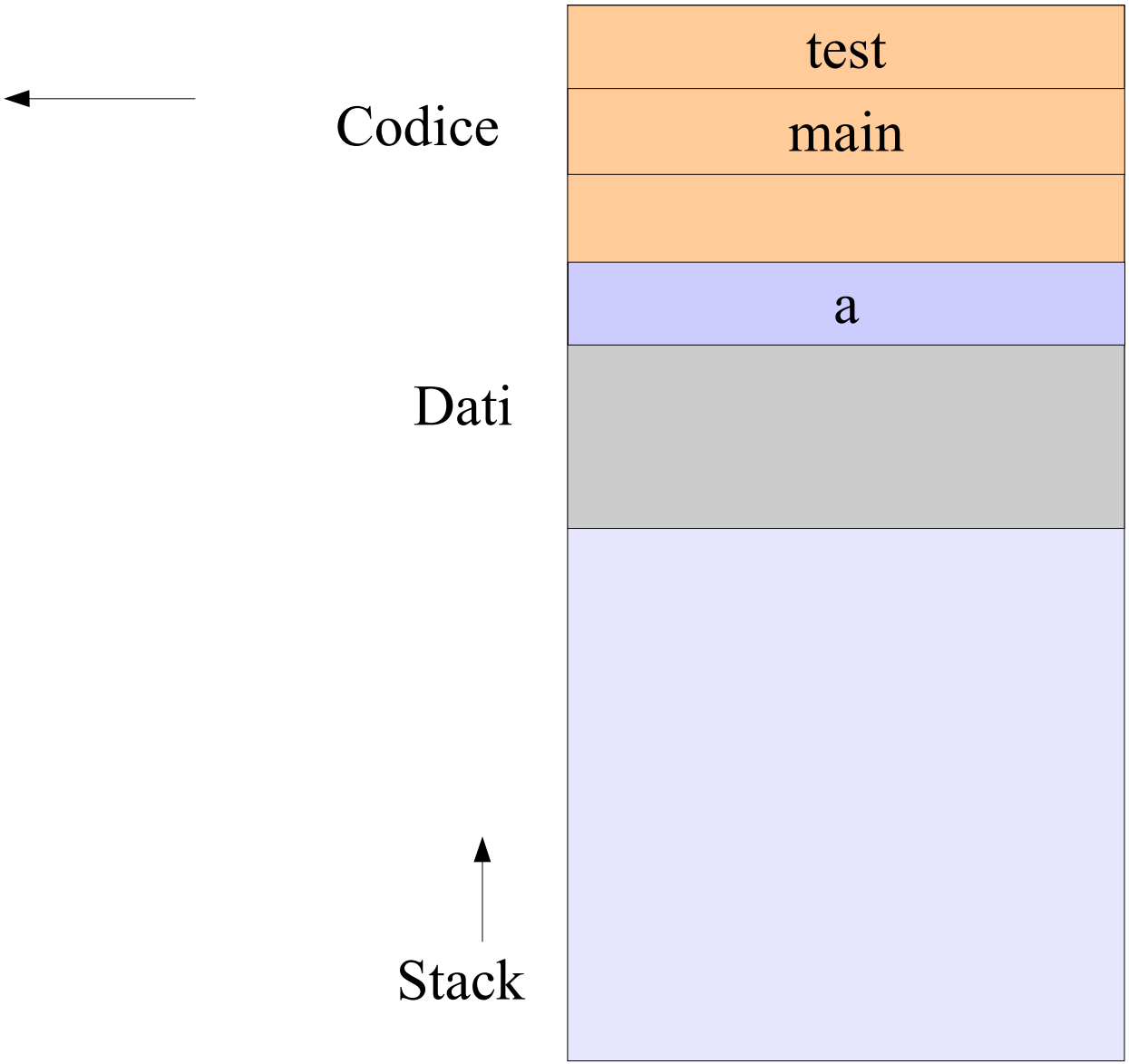
```
int a=3;
void test( )
{
  int b=2;
}
int main( )
{
  int c=1;
  test();
}
```



© 2002-2005 Renzo Davoli, Alberto Montresor

Layout di memoria

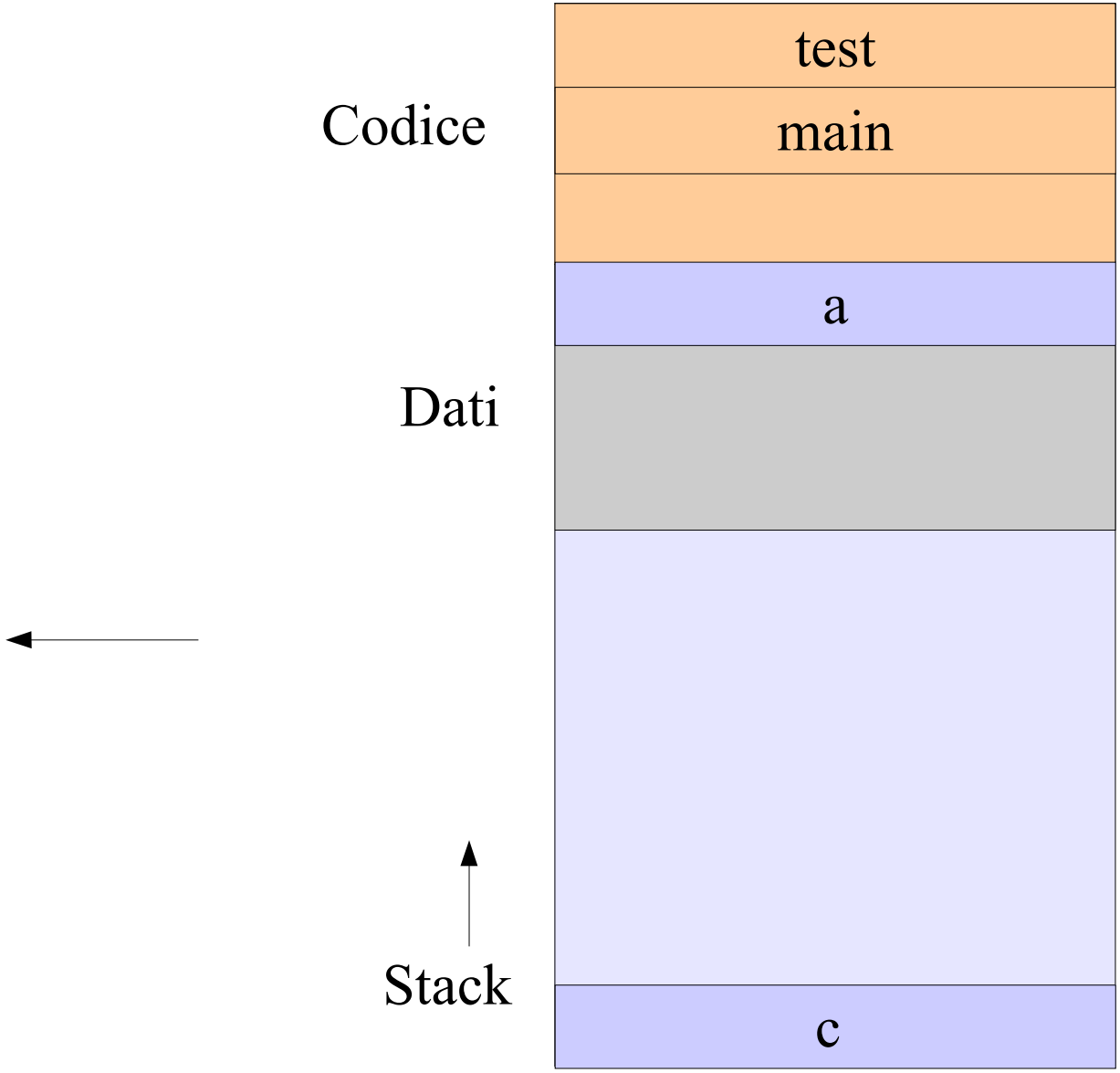
```
int a=3;
void test( )
{
  int b=2;
}
int main( )
{
  int c=1;
  test();
}
```



© 2002-2005 Renzo Davoli, Alberto Montresor

Layout di memoria

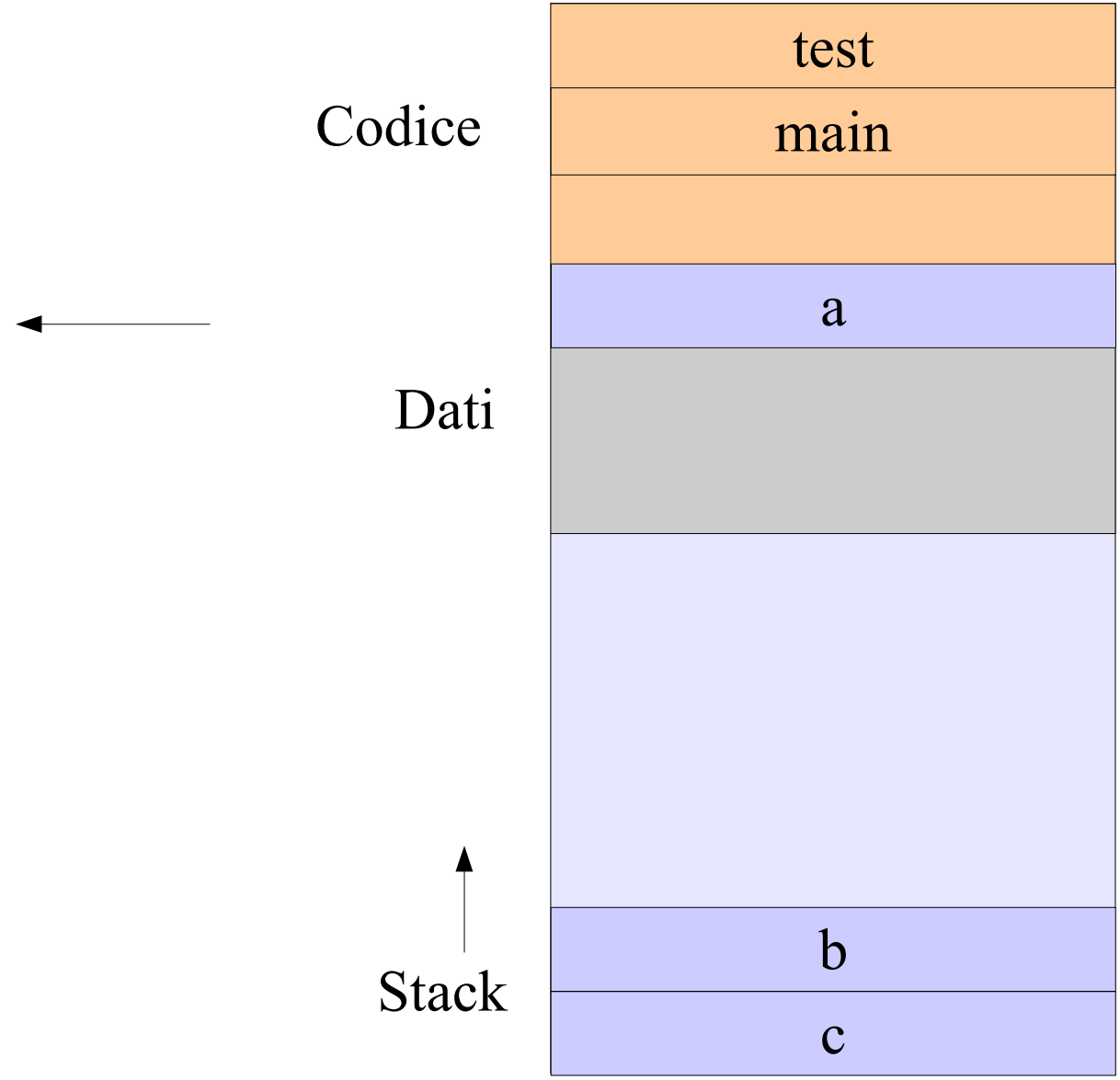
```
int a=3;
void test( )
{
  int b=2;
}
int main( )
{
  int c=1;
  test();
}
```



© 2002-2005 Renzo Davoli, Alberto Montresor

Layout di memoria

```
int a=3;
void test( )
{
  int b=2;
}
int main( )
{
  int c=1;
  test();
}
```



© 2002-2005 Renzo Davoli, Alberto Montresor

Variabili esterne

- **Keyword `extern`**
  - E' possibile dichiarare una variabile globale in qualsiasi parte di un programma, anche in un file separato
  - In questo caso, bisogna distinguere fra:
    - dichiarazione "reale" (alloca memoria per la variabile):
      la sintassi è la solita
    - dichiarazione "extern" (rende noto al compilatore l'esistenza della variabile):
      la sintassi prevede il qualificatore **`extern`** per informare il compilatore che ci si riferisce ad una variabile dichiarata in un altro file
- **Esempio**
  - `extern int var;`

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio - Variabili esterne

```
/* test_main.c */
int a=4;
extern int b;
int test( );
int main( )
{
  printf("a=%d,b=%d\n", a,b);
  a = b = 5;
  test();
  return 0;
}
```

```
/* test.c */
extern int a;
int b=3;
int test( )
{
  printf("a=%d,b=%d\n",a,b);
}
```

Cosa succede in compilazione/esecuzione?

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio - Variabili esterne

```
/* test_main.c */
int a=4;
extern int b;
int test( );
int main( )
{
  printf("a=%d,b=%d\n", a,b);
  a = b = 5;
  test();
  return 0;
}
```

```
/* test.c */
extern int a;
int b=3;
int test( )
{
  printf("a=%d,b=%d\n",a,b);
}
```

Cosa succede in compilazione/esecuzione?

gcc main.c test.c -o test

./test

a=2, b=3

a=5, b=5

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio - Variabili esterne

```
/* test_main.c */
int a=4;
int b=2;
int test( );
int main( )
{
  printf("a=%d,b=%d\n", a,b);
  a = b = 5;
  test();
  return 0;
}
```

```
/* test.c */
int a=3;
int b=3;
int test( )
{
  printf("a=%d,b=%d\n",a,b);
}
```

Cosa succede in compilazione/esecuzione?

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio - Variabili esterne

```
/* test_main.c */
int a=4;
int b=2;
int test( );
int main( )
{
  printf("a=%d,b=%d\n", a,b);
  a = b = 5;
  test();
  return 0;
}
```

```
/* test.c */
int a=3;
int b=3;
int test( )
{
  printf("a=%d,b=%d\n",a,b);
}
```

Cosa succede in compilazione/esecuzione?

gcc main.c test.c -o test

/tmp/cc8W0ciu.o(.data+0x0): multiple definition of `a'
/tmp/ccUvzasl.o(.data+0x0): first defined here
/tmp/cc8W0ciu.o(.data+0x4): multiple definition of `b'
/tmp/ccUvzasl.o(.data+0x4): first defined here

© 2002-2005 Renzo Davoli, Alberto Montresor

Variabili automatiche

- **Le variabili locali sono *automatiche*:**
  - Esistono solo mentre l'esecuzione è nel range del blocco in cui sono definite
  - Quando il processo entra nel blocco in cui la variabile è definita:
    - la variabile viene creata
  - Quando il processo esce dal blocco in cui la variabile è definita:
    - la variabile viene distrutta
  - Nel caso il processo esca da un blocco, per poi rientrare, la variabile è da considerare come una nuova istanza
- **Nota:**
  - Le variabili automatiche vengono allocate nello stack
  - Esiste una keyword **`auto`**, che però non è necessaria

© 2002-2005 Renzo Davoli, Alberto Montresor

Variabili automatiche

```
#include <stdio.h>
#include <stdlib.h>
void func1(){
   int x;
   printf("1-> %d\n",x);
   x = 10;
}
void func2(){
   int y = rand();
   printf("2-> %d\n",y);
}
```

```
int main(){
   int i;
   for(i=0;i<5;i++){
       func1();
       func2();
   }
   return 0;
}
```

- **Nota:**
  - Stampa numeri casuali
  - tecnica da non utilizzare MAI!

```
1-> -1073745260
2-> 1804289383
1-> 1804289383
2-> 846930886
1-> 846930886
2-> 1681692777
1-> 1681692777
2-> 1714636915
1-> 1714636915
2-> 1957747793
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Variabili statiche - Locali

- **Variabili statiche – Globali**
  - Sono visibili solo nel file in cui sono dichiarate
- **Variabili statiche - Locali**
  - A volte è necessario prolungare la vita di una variabile locale
  - Per esempio, per contare il numero di invocazioni di una funzione
- **Così non va (x è automatica):**

```
int count( ){
   int x=0;
   return (++x);
}
```

- **Keyword static**
  - Una variabile marcata **`static`** non è auto, ovvero non viene distrutta dopo l'uso

```
int count( ){
   static int x=0;
   return (++x);
}
int main() {
   printf("%d\n", count());
   printf("%d\n", count());
   printf("%d\n", count());
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio – Variabili static globali

```
/* test_main.c */
int a=4;
static int b = 1;
int test( );
int main( )
{
  printf("a=%d,b=%d\n", a,b);
  a = b = 5;
  test();
  return 0;
}
```

```
/* test.c */
static int a = 2;
int b=3;
int test( )
{
  printf("a=%d,b=%d\n",a,b);
}
```

Cosa succede in compilazione/esecuzione?

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio – Variabili static globali

```
/* test_main.c */
int a=4;
static int b = 1;
int test( );
int main( )
{
  printf("a=%d,b=%d\n", a,b);
  a = b = 5;
  test();
  return 0;
}
```

```
/* test.c */
static int a = 2;
int b=3;
int test( )
{
   printf("a=%d,b=%d\n",a,b);
}
```

Cosa succede in compilazione/esecuzione?

gcc main.c test.c -o test

./test

a=4, b=1

a=2, b=3

© 2002-2005 Renzo Davoli, Alberto Montresor

Riassunto

- **Le variabili locali automatiche:**
  - Esistono solo quando l'esecuzione è nel blocco in cui le variabili sono definite; il loro contenuto non persiste dopo che l'esecuzione lascia il blocco
- **Le variabili locali statiche:**
  - Esistono per l'intera vita di un processo, ma è possibile fare riferimenti ad esse solo quando l'esecuzione è nel blocco in cui sono definite
- **Le variabili globali**
  - Esistono sempre, sono visibili ovunque (salvo essere nascoste da variabili locali)
- **Le variabili globali statiche**
  - Sono visibili solo nel file in cui sono dichiarate

© 2002-2005 Renzo Davoli, Alberto Montresor

Variabili "registro"

- La keyword `register`
  - Suggerisce al compilatore che una variabile verrà usata pesantemente e che sarebbe meglio mantenerla direttamente in un registro del processore per aumentare l'efficienza
  - Esempio:

```
register int i;
for (i=0; i < 1000000; i++) ...
```

  - Note:
    - E' possibile dichiarare **`register`** solo variabili automatiche o parametri formali di una funzione
    - Il numero delle variabili dichiarabili **`register`** è limitato
    - Solo certi tipi di dati possono essere mantenuti nei registri
    - Il compilatore può comunque fare di testa sua...

© 2002-2005 Renzo Davoli, Alberto Montresor

Nomi di variabile

- **I nomi di variabile**
  - Possono contenere lettere, numeri e underscore _
  - Il primo carattere deve essere una lettera o un underscore
  - Case-sensitive
  - Le keyword C non possono essere usati come nomi di variabili
- **Esempi**
  - `present, hello, r2d2` `/* OK */`
  - `_1993_tar_return` `/* OK but don't */`
  - `Hello#there` `/* illegal */`
  - `double` `/* illegal */`
  - `2fartogo` `/* illegal */`

© 2002-2005 Renzo Davoli, Alberto Montresor

Nomi di variabile

- **Suggerimenti:**
  - FARE:
    - Utilizzare nomi significativi per le variabili
    - Adottare una "naming convention"
  - NON FARE:
    - Utilizzare nomi di variabile che iniziano con un underscore (spesso vengono utilizzati dal compilatore C)
    - Utilizzare variabili con tutte lettere maiuscole (per convenzione, utilizzate per costanti)

© 2002-2005 Renzo Davoli, Alberto Montresor

Tipi

- **Ci sono solo pochi tipi base in C**
  - `char`:
    Un singolo byte, in grado di contenere un singolo carattere
  - `int`:
    Un intero di lunghezza fissata, che normalmente riflette la lunghezza "naturale" degli interi sulla macchina host
    (i.e., 32 o 64 bit)
  - `float`:
    Floating point singola precisione
  - `double`:
    Floating point doppia precisione

© 2002-2005 Renzo Davoli, Alberto Montresor

Tipi

- **Ci sono un certo numero di qualificatori che possono essere applicati ai tipi base**
  - Lunghezza dei dati
    - `short int` intero "corto", numero di bit <= int
      si puo scrivere anche solo `short`
    - `long int` intero "lungo", numero di bit >= int
      si può scrivere anche solo `long`
    - `long double` generalmente estende la precisione
  - Segno
    - `unsigned int` un tipo intero senza segno
    - `unsigned char` un valore da 0 a 255
    - `signed char` un valore da -128 a +127

© 2002-2005 Renzo Davoli, Alberto Montresor

Tipi

- **Ogni tipo ha una dimensione fissa associato ad esso**
  - Questa dimensione può essere determinata a tempo di compilazione
  - Varia a seconda della architettura del computer
  - I programmi che si basano su queste dimensione devono prestare attenzione per rendere il loro codice portabile

© 2002-2005 Renzo Davoli, Alberto Montresor

Tipi

Una tipica macchina a 32 bit:

Type	Keyword	Bytes	Range
character	char	1	-128...127
integer	int	4	-2,147,483,648...2,147,438,647
short integer	short	2	-32768...32367
long integer	long	4	-2,147,483,648...2,147,438,647
long long integer	long long	8	-92233720368547758O8 ....
unsigned character	unsigned char	1	0...255
unsigned integer	unsigned int	2	0...4,294,967,295
unsigned short integer	unsigned short	2	0...65535
unsigned long integer	unsigned long	4	0...4,294,967,295
single-precision	float	4	1.2E-38...3.4E38
double-precision	double	8	2.2E-308...1.8E308

© 2002-2005 Renzo Davoli, Alberto Montresor

Tipi

La funzione *sizeof()* ritorna la dimensione di un tipo

```
int main() {
  printf("Size of char ......... = %2d byte(s)\n", sizeof(char));
  printf("Size of short ........ = %2d byte(s)\n", sizeof(short));
  printf("Size of int .......... = %2d byte(s)\n", sizeof(int));
  printf("Size of long long .... = %2d byte(s)\n", sizeof(long long));
  printf("Size of long ......... = %2d byte(s)\n", sizeof(long));
  printf("Size of unsigned char. = %2d byte(s)\n", sizeof (unsigned char));
  printf("Size of unsigned int.. = %2d byte(s)\n", sizeof (unsigned int));
  printf("Size of unsigned short = %2d byte(s)\n", sizeof (unsigned
   short));
  printf("Size of unsigned long. = %2d byte(s)\n", sizeof (unsigned long));
  printf("Size of float ........ = %2d byte(s)\n", sizeof(float));
  printf("Size of double ....... = %2d byte(s)\n", sizeof(double));
  printf("Size of long double .. = %2d byte(s)\n", sizeof(long double));
  return 0;
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Creare tipi semplici

- ***typedef*** **crea un nuovo nome per un tipo esistente**
  - Permette di creare un nuovo nome di tipo da utilizzare al posto di un vecchio nome di tipo
  - Perchè usarlo
    - per creare tipi "portabili", che possano essere adattati all'architettura semplicemente cambiando gli header file
    - per semplificare sintassi complesse (tipi strutturati)
- **Sintassi generica:**
  - `typedef oldtype newtype`
- **Esempi:**
  - `typedef long long int64;  /* intero a 64 bit */`
  - `typedef unsigned char byte;  /* tipo byte */`
  - `typedef long double extended;`

© 2002-2005 Renzo Davoli, Alberto Montresor

Casting

- E' possibile cambiare il tipo di un'espressione attraverso l'operazione di *casting*
- Esempio
  - `int approximation;`
  - `approximation = (int) (3.14 * raggio * raggio);`

© 2002-2005 Renzo Davoli, Alberto Montresor

Costanti

- **E' possibile dichiarare variabili come costanti**
  - Si utilizza il qualificatore **`const`**
  - Esempio:
    - `const double pi=3.1415926;`
    - `const int maxlength=2356;`
    - `const int val=(3*7+6)*5;`
- **Vantaggi nell'utilizzare le costanti:**
  - Informa chi legge il codice che un certo valore non cambia
  - Semplifica la lettura di codice di grande dimensione
  - Informa il compilatore che un valore non cambia
  - Il compilatore può ottimizzare il codice oggetto tenendo conto di questo fatto
- **Utilizzate costanti, quando appropriato**

© 2002-2005 Renzo Davoli, Alberto Montresor

Costanti

- **Esiste una forma più "vecchia" di definizione di costante:**
  - Utilizza il preprocessore C
  - Forma generica: `#define CONSTNAME literal`
  - Esempio: `PI 3.14159`
  - Generalmente, le costanti del preprocessore sono scritte in upper case (convenzione)
- **Cosa succede effetivamente:**
  - Il preprocessore C viene eseguito prima del compilatore
  - Ogni volta che vede il token `PI`, lo sostituisce con `3.14159`
  - Il compilatore esegue quindi il codice C "preprocessato"
  - Attenzione: può essere pericoloso!

© 2002-2005 Renzo Davoli, Alberto Montresor

Funzioni

- **Una funzione è una sezione indipendente di codice**
  - E' completamente auto-contenuta
  - Esegue un compito specifico
- **Una funzione può tornare zero o più valori al chiamante:**
  - Tramite `return`
  - Tramite variabili di tipo reference (vedremo in seguito)
  - Modificando variabili globali (male! non usare!)
- **Le funzioni rendono il codice riutilizzabile e leggibile**

© 2002-2005 Renzo Davoli, Alberto Montresor

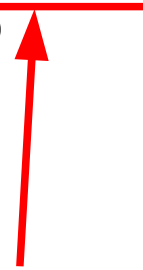
Funzioni: Sintassi

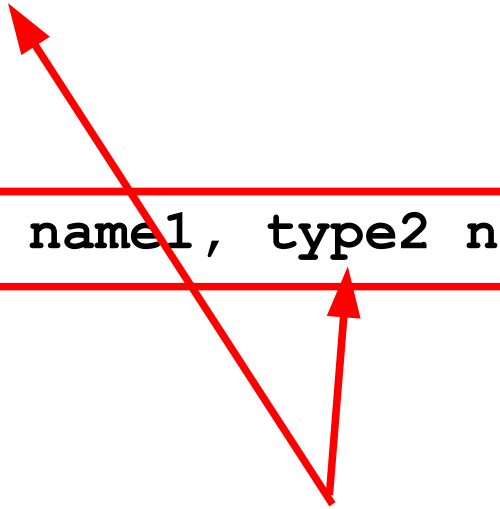
- **Prototipo di funzione:**

```
return_type function_name(type1 name1, type2 name2, ...,
   typen namen);
```

- **Definizione di funzione:**

```
return_type function_name(type1 name1, type2 name2, ...,
   typen namen)
{
}
```





© 2002-2005 Renzo Davoli, Alberto Montresor

Esempi di prototipi e definizioni

- **Esempi di prototipi di funzione**

```
double squared(double number);
void print_report(int);
int get_menu_choice(void);
```

- **Esempi di definizioni di funzione**

```
double squared(double number)
{
  return (number * number);
}
void print_report(int report_number)
{
  if (report_number == 1)
    printf("Printing Report 1");
  else
    printf("No report %d\n", report_number);
}
```

void significa che non prende in input parametri

void significa che non restituisce nessun valore

© 2002-2005 Renzo Davoli, Alberto Montresor

Passaggio parametri

- **Gli argomenti sono passati come in Java**
  - Chiamata di funzione: `func1 (a, b, c);`
  - Header di funzione: `int func1(int x, int y, int z)`
  - I parametri `x y z` sono inizializzati con i valori di `a b c`
- **Ogni argomento può essere una qualsiasi espressione C valida:**
  - Esempio: `x = func1(x+1,func1(2,3,4),5);`
  - Possono avvenire conversioni di tipo nel caso i tipi non corrispondano

© 2002-2005 Renzo Davoli, Alberto Montresor

Passaggio parametri

- **Tutti i parametri sono passati per valore**
  - In pratica, questo significa che sono variabili locali inizializzate dagli argomenti delle chiamata
  - Possono essere modificati, ma queste modifiche non possono essere viste nella funzione chiamante
  - Cosa stampa il codice a lato?

```
#include<stdio.h>
int twice(int x)
{
  x=x+x;
  return x;
}
int main()
{
  int x=10;
  int y;
  y=twice(x);
  printf("%d,%d\n",x,y);
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Come strutturare il codice

- **Programmi grandi**
  - Scrivete più file `.c` contenenti funzioni correlate
  - Scrivete file `.h` contenti costanti e prototipi delle funzioni
  - Usate `#include` per includere i file `.h`
- **Programmi piccoli**
  - Singolo file
    - prototipi
    - funzione main
    - altre funzioni

mymath.h

```
int min(int x,int y);
int max(int x,int y);
```

mymath.c

```
int min(int x,int y)
{
  return x>y?y:x;
}
int max(int x,int y)
{
  return x>y?x:y;
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Espressioni

- **Le espressioni**
  - Sono uno degli elementi fondamentali del C
  - Sono formate da operandi e operatori (!)
- **Esempi**
  - `x=0`
  - `x=x+1`
  - `printf("%d", x);`
- **Nota:**
  - Ovviamente, le espressioni in C sono molto simili a Java

© 2002-2005 Renzo Davoli, Alberto Montresor

Operandi di assegnamento

- **x=3**
  - = è un operatore
  - Il valore di questa espressione è 3
  - L'operatore = ha un side-effect:
    - assegna il valore del "right hand side" al "left hand side"
    - ovvero, assegna 3 a x
  - Il valore corrisponde al valore del right hand side
  - Assegna 3 a x
- **Esempio:**
  - x = (y = 3) + 1;
  - Qual è il valore di x?

© 2002-2005 Renzo Davoli, Alberto Montresor

Operatori di assegnamento composti

- **Il C è un linguaggio "stringato":**
  - Offre molte abbreviazione per costrutti comuni, come ad esempio operazioni di "aggiornamento" di una variabile
- **Operatori di aggiornamento**
  - Composti da un operatore (aritmetico/logico) e da una variabile
- **Esempi:** **Equivalenti a:**
  - x *= y; x = x*y;
  - y -= z+1; y = y- (z+1);
  - a &= b; a = a & b;

Pre-{in,de}cremento: ++x, --x
è equivalente a (x = x+1), (x = x-1)
ha valore $x_{(new)}$

Post-{in,de}incremento: x++, x--
ha il side-effect di incrementare x
ha valore $x_{(old)}$

© 2002-2005 Renzo Davoli, Alberto Montresor

Operatori

- **Operatori relazionali**
  - ==, >, < , >=, <=, !=
  - Necessari per confrontare valori numerici
  - Ritornano 1 per vero, 0 per falso
  - Non esiste un tipo boolean in C; invece, C utilizza un intero non-zero come vero e 0 come falso
- **Operatori logici**
  - && AND
  - || OR
  - ! NOT

© 2002-2005 Renzo Davoli, Alberto Montresor

Operatori logici binari

- **Lavorano su tutti i tipi interi**
  - & Bitwise AND
    - x= 0xFFF0
    - y= 0x002F
    - x&y == 0x0020
  - | Bitwise Inclusive OR
    - x|y == 0xFFFF
  - ^ Bitwise Exclusive OR
    - x^y= 0xFFDF
  - ~ Bitwise NOT
    - ~ y == 0xFFD0
  - x << n
    - shift a sinistra di n posizioni
  - x >> n
    - shift a destra di n posizioni
    - aritmetico oppure no a seconda del segno di x

© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture di controllo

- **Statement condizionali: *if else***
  - Sintassi:

```
if (condition)
   statement1
else
   statement2
```

```
if (condition)
   statement1
```

  - Lo statement `if` viene utilizzato per eseguire uno statement se una certa condizione è vera; viene eseguito lo statement contenuto nella clausola opzionale `else` altrimenti
  - *condition* è un espressione booleana

© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture di controllo

- **Statement condizionali: *switch case default***
  - Sintassi:

```
switch (espressione) {
   case value1:
      statement1
      break;
   case value2:
      statement2
      break;
   […]
   default:
      statementn
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture di controllo

- **Spiegazione:**
  - Valuta una espressione e cerca di fare il matching fra il valore ottenuto e uno o piu' valori associati ad elementi `case`
  - Se esiste un matching con uno degli elementi `case`, lo statement associato viene eseguito
  - Se non è possibile fare alcuna associazione, lo statement associato a `default` viene eseguito
  - Lo statement `break` forza l'uscita dallo switch; in sua assenza, l'esecuzione passerebbe al case successivo

© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture di controllo

- **Loop statement: *while***
  - Sintassi:

```
while (condition)
    statement
```

  - Un loop **`while`** esegue lo statement fintanto che la condizione associata è vera
  - All'interno di un ciclo while:
    - Lo statement **`break`** forza l'uscita dal ciclo
    - Lo statement **`continue`** forza la terminazione dell'iterazione corrente e torna a valutare l'espressione
  - Sconsigliati! Meglio utilizzare opportunamente variabili di condizione

© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture di controllo

- **Loop statement: *do ... while***
  - Sintassi:

```
do {
    statement
} while (condition)
```

  - Un loop **`while`** esegue lo statement fintanto che la condizione associata è vera
  - Lo statement viene eseguito almeno una volta prima che la condizione venga valutata
  - Anche in questo ciclo è possibile utilizzare **`break`** e **`continue`**

© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture di controllo

- **Loop statement: *for***
  - Sintassi:

```
for (expression1; condition; expression2)
    statement
```

  - All'inizio, il loop **`for`** valuta *`expression1`*
  - Dopo di che, entra in un ciclo in cui:
    - Valuta *`condition`*; se *`condition`* è **`false`**, esce
    - Esegue *`statement`*
    - Valuta *`expression2`*
  - Anche in questo ciclo è possibile utilizzare **`break`** e **`continue`**

© 2002-2005 Renzo Davoli, Alberto Montresor

Cerca il bug

```
#include <stdio.h>
/* Print numbers from 1 to 10 */
int main()
{
  int i;
  for (i=1; i <= 10; i=i+1);
    printf("i is %d\n", i);
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Cerca il bug

```
#include <stdio.h>
/* Check if ten different numbers are equal to 2 */
int main()
{
  int i;
  for (i=0; i < 10; i=i+1)
    if (i=2)
      printf("i is 2");
    else
      printf("i is not 2");
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Cerca il bug

```
#include <stdio.h>
/* Check if ten different numbers are less than 2 */
int main()
{
  int i;
  for (i=0; i < 10; i=i+1)
    if (i<2)
      printf("i is less than 2");
      printf("and is not 2, either");
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Librerie ANSI C

- **ANSI C**
  - Nel 1989-1990, lo standard ANSI C (ANSI X3.159-1989; ISO/IEC 9899:1990) è stato adottato dall'ISO e dall'ANSI
  - Lo scopo dell'ANSI C è quello di permettere la portabilità di programmi C ad una grande varietà di sistemi operativi
  - Definisce:
    - Sintassi e semantica del linguaggio di programmazione
    - Una libreria standard di funzioni
  - La libreria può essere suddivisa in 15 aree sulla base dei 15 header files definiti nello standard

© 2002-2005 Renzo Davoli, Alberto Montresor

Librerie ANSI C

- **Le librerie incluse in ANSI C contengono molte funzioni di utilità:**

`<assert.h>`:	asserzioni
`<ctype.h>`:	test sui caratteri
`<errno.h>`:	codici di errore
`<float.h>`:	limiti per tipi floating point
`<limits.h>`:	limiti
`<locale.h>`:	informazioni "locali" (lingua, punteggiatura, etc)
`<math.h>`:	funzioni matematiche
`<setjmp.h>`:	salti non-locali
`<signal.h>`:	segnali

© 2002-2005 Renzo Davoli, Alberto Montresor

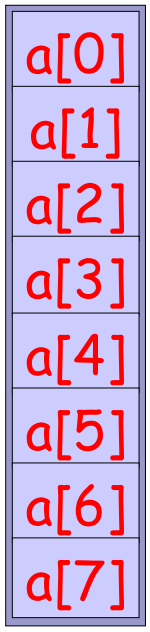
Librerie ANSI C

- **Elenco (continua...)**
  - `<stdarg.h>`: gestione argomenti
  - `<stddef.h>`: definizioni di uso generale
  - `<stdio.h>`: input e output
  - `<stdlib.h>`: funzioni varie
  - `<string.h>`: funzioni di stringa
  - `<time.h>`: funzioni relative a ora e data
- **Documentazione:**
  - Elenco breve di tutte le funzioni in ANSI C nella pagina di documentazione
  - Documentazione singole funzioni tramite:
    `man 3 nomefunzione`
  - Esempio: **`man 3 printf`**

© 2002-2005 Renzo Davoli, Alberto Montresor

Array

- **Dichiarazione generica: `typename varname[size]`**
  - **`typename`** è un qualunque tipo
  - **`varname`** è una qualunque variabile legale
  - **`size`** è un valore intero che possa essere ottenuto a tempo di compilazione
- **Esempio: `int a[8];`**
  - dichiara un array di 8 interi, indicizzati da 0 a 7
  - questo array occupa 8**`*sizeof(int)`** byte



© 2002-2005 Renzo Davoli, Alberto Montresor

Array: controllo sugli indici

- **Il C non effettua controlli sugli indici degli array a runtime**
  - a differenza di altri linguaggi di programmazione (es. Java)
  - alcuni compilatori possono fare controlli su indici calcolabili a tempo di compilazione
  - motivazione: maggiore efficienza
- **Cosa succede se si esce dal range "legale"?**
  - semplicemente si accede a dati fuori dall'array
  - spesso e volentieri, questo causerà eccezioni di memoria (segmentation fault)
  - è responsabilità del programmatore scrivere programmi corretti...

© 2002-2005 Renzo Davoli, Alberto Montresor

Array: Inizializzazione

- **Metodi alternativi per inizializzare un array:**
  - ```
    int array [4];
    array[0] = 100;
    array[1] = 200;
    array[2] = 300;
    array[3] = 400;
    ```
  - ```
    int array [4] = { 100, 200, 300, 400 };
    ```
  - ```
    int array [ ] =  { 100, 200, 300, 400 };
    ```
- **Note:**
  - se la dimensione dell'array è più grande del numero di inizializzatori, gli elementi rimanenti sono inizializzati con l'ultimo valore
  - nella terza versione, il compilatore conta il numero di elementi per voi

© 2002-2005 Renzo Davoli, Alberto Montresor
    ```

Esempio

```
#include <stdio.h>
int main() {
  float expenses[12]= { 10.3, 9, 7.5, 4.3, 10.5, 7.5,
                        7.5, 8, 9.9, 10.2, 11.5, 7.8};
  int count,month;
  float total;
  for (month=0, total=0.0; month < 12; month++) {
    total+=expenses[month];
  }
  for (count=0; count < 12; count++)
    printf ("Month %d = %.2f K$\n", count+1,
                                    expenses[count]);
  printf("Total = %.2f K$, Average = %.2f K$\n",
         total, total/12);
  return 0;
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Array multidimensionali

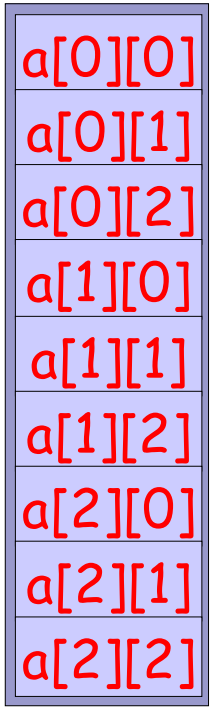
- **Gli array in C possono essere multidimensionali**
  - esempio:

```
int  a[4][3];
```

  - definisce un array bidimensionale
  - definisce un array di `int[3]`

- **Inizializzazione:**
  -

```
int a[3][3] = { {1, 2, 3} , { 4, 5, 6} ,
                {7, 8, 9} }
```



© 2002-2005 Renzo Davoli, Alberto Montresor

Array: esempio

```
#include <stdio.h>
#include <stdlib.h>
int main ()  {
  int random1[8][8];
  int a, b;
  for (a = 0; a < 8; a++)
    for (b = 0; b < 8; b++)
      random1[a][b] = rand()%2;
  for (a = 0; a < 8; a++) {
    for (b = 0; b < 8; b++)
      printf ("%c" , random1[a][b] ? 'x' : ' ');
    printf("\n");
   }
   return 0;
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Array: passaggio di parametri

- **Nota:**
  - Sembrerebbe che gli array vengano "passati per indirizzo"
  - E' possibile modificare gli elementi di un array
  - Nei lucidi sui puntatori, vedremo che gli array sono passati per valore

```
void inc_array(int a[ ],
               int size) {
  int i;
  for(i=0;i<size;i++) {
    a[i]++;
  }
}
```

```
int main()
{
  int test[3]={1,2,3};
  int ary[4]={1,2,3,4};
  int i;
  inc_array(test,3);
  for(i=0;i<3;i++)
    printf("%d\n",test[i]);
  inc_array(ary,4);
  for(i=0;i<4;i++)
    printf("%d\n",ary[i]);
  return 0;
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

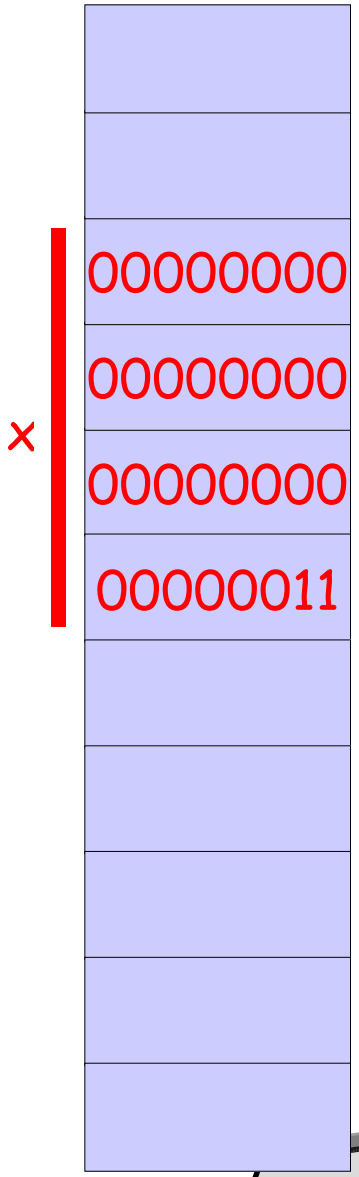
Array: Esempio

```
void mysort(int a[],int size)
{
  int i,j;
  int x;
  for(i=0; i<size; i++) {
    for(j=i; j>0; j--) {
      if(a[j] < a[j-1]) {
        /* Change the order of a[j] and a[j-1] */
        x=a[j];a[j]=a[j-1]; a[j-1]=x;
      }
    }
  }
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori - Premessa

- **Quando una variabile è definita:**
  - Viene allocata della memoria per contenere la variabile (segmento dati, segmento stack, etc.)
  - Es: `int x`
- **Quando un valore viene assegnato ad una variabile:**
  - Tale valore viene collocato nella memoria che è stata allocata.
  - Es. `x=3`
- **Quando una variabile è usata come left-handside in una espressione**
  - Il suo valore viene utilizzato nell'espressione
  - Es. `z = x;`

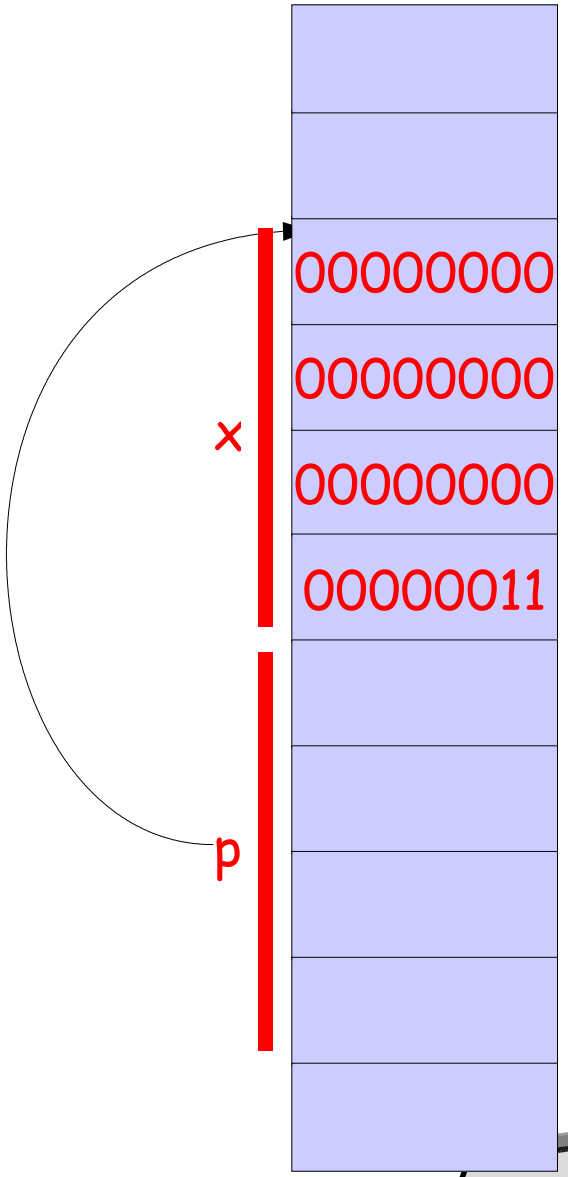


© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori - Indirizzo di una variabile

- **E' possibile ottenere l'indirizzo di una variabile:**
  - Utilizzando l'operatore **`&`**
  - Es. **`&x`**
- **L'indirizzo può essere passato ad una funzione:**
  - **`scanf("%d", &x);`**
- **L'indirizzo può essere memorizzato in una variabile:**
  - dichiarazione: **`type *pointername;`**
  - dichiara una variabile pointername che punta ad un oggetto di tipo type
  - Es.

```
int* p;
p = &x;
```



© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori - Esempio 1

```
int x;

int * p1;

char *p2;

p1 = &x;                /* Store the address in p1 */

scanf("%d", p1);        /* i.e. scanf("%d",&x); */

p2 = &x;                /* Will get warning message */
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori - Esempio 2

- **Suggerimento:**
  - come le altre variabili, inizializzate sempre i puntatori prima di usarli!
  - il compilatore non vi avviserà
- **Ad esempio:**
  - ```
    int main(){
      int x;
      int *p;
      scanf("%d",p);          /* Wrong! */
      p = &x;
      scanf("%d",p);          /* Correct *
    }
    ```

© 2002-2005 Renzo Davoli, Alberto Montresor
    ```

Puntatori: utilizzare i valori

- **E' possibile utilizzare i puntatori per accedere al contenuto delle variabili:**
  - per fare questo, utilizzate l'operatore * (dereferencing operator)
  - nota: a seconda del contesto, * ha significati differenti
- **Esempi:**

```
int m=3;
int n;
int *p;
p=&m;
n=*p;
printf("%d\n", m);
printf("%d\n", n);
printf("%d\n",*p);
```

```
int m=3;
int n=100;
int *p;
p=&m;
printf("m is %d\n",*p);
m++;
printf("now m is %d\n",*p);
p=&n;
printf("n is %d\n",*p);
*p=500;
printf("n is %d\n", n);
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori come parametri di funzione

- **A volte:**
  - è desiderabile che una funzione assegni un valore ad una variabile non locale
  - è desiderabile che una funzione ritorni più di un valore
- **Soluzione 1 (sbagliata):**
  - utilizzare variabili globali
    - la funzione chiamata assegna nuovi valori alle variabili globali
    - la funzione chiamante legge i valori dalle variabili globali
  - problema: codice non riutilizzabile
- **Soluzione 2 (corretta):**
  - passare alla funzione i puntatori alle variabili da modificare

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori come parametri di funzione

```
void minmax(int a, int b,
            int *min, int *max)
{
  if(a>b){
    *max=a;
    *min=b;
  }
  else {
    *max=b;
    *min=a;
  }
}
```

```
int main()
{
  int x,y;
  int small,big;
  printf("Two int: ");
  scanf("%d %d",&x,&y);
  min_max(x,y,&small,
    &big);
  printf("%d <= %d",
    small, big);
  return 0;
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Aritmetica dei puntatori

- Quando una variabile di tipo puntatore punta agli elementi di un array, è possibile aggiungere o sottrarre interi ai puntatori

```
int a[10], *p, *q;
p = &a[2];
q = p + 3;              /* q points to a[5] now */
p = q - 1;              /* p points to a[4] now */
p++;                    /* p points to a[5] now */
p--;                    /* p points to a[4] now */
*p = 123;               /* a[4] = 123 */
*q = *p;                /* a[5] = a[4]   */
q = p;                  /* q points to a[4] now */
scanf("%d", q);         /* scanf("%d", &a[4])  */
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Aritmetica dei puntatori

- Se due puntatori puntano a elementi dello stesso array, è possibile sottrarre e confrontare i puntatori

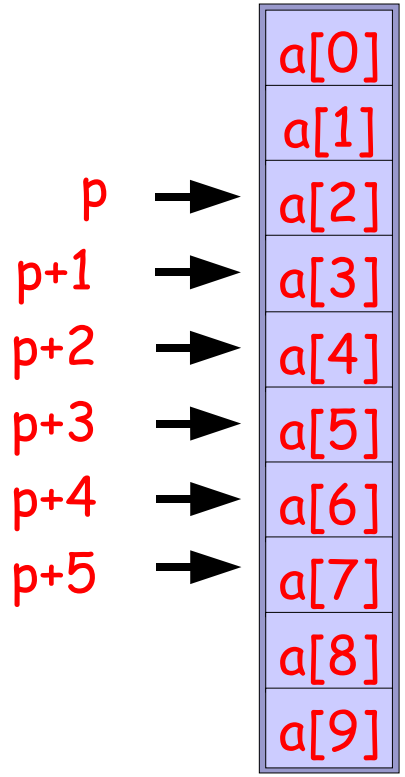
```
int a[10], *p, *q , i;
p = &a[2];
q = &a[5];
i = q - p;              /* i is 3 */
i = p - q;              /* i is -3 */
a[2] = a[5] = 2;
i = *p - *q;            /* i = a[2] - a[5]  */
p < q;                  /* true */
p == q;                 /* false */
p != q;                 /* true */
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Utilizzare puntatori per accedere ad array

- **Questi due esempi di codice sono equivalenti:**

```
int a[10];                  int a[10];
int *p;
p = &a[2];                  a[2] = 10;
*p = 10;                    a[3] = 10;
*(p+1) = 10;
printf("%d", *(p+3));       printf("%d", a[5]);
```

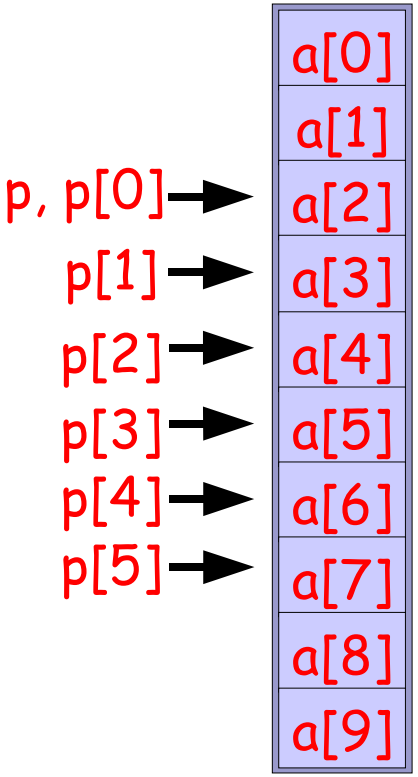


© 2002-2005 Renzo Davoli, Alberto Montresor

Utilizzare puntatori per accedere ad array

- Questi due esempi di codice sono equivalenti:

```
int a[10];                  int a[10];
int *p;
p = &a[2];                  a[2] = 10;
p[0] = 10;                  a[3] = 10;
p[1] = 10;
printf("%d", p[3]);         printf("%d", a[5]);
```



© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio

```
/* Sum - sum up the ints in the
   given array */
int sum(int *ary, int size)
{
  int i, s;
  for(i = 0, s=0; i<size; i++){
    s+=ary[i];
  }
  return s;
}
```

```
/* In another function */
int a[1000]
int x;
……
x= sum(&a[100],50);
/* This sums up a[100], a[101],
   …,
   a[149] */
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Array vs puntatori

- **Il nome di un array**
  - è simile a un "puntatore costante" che punta al primo elemento dell'array
  - non è possibile modificare il valore del puntatore
  - per questo motivo, è possibile "passare un array" ad una funzione; quello che viene passato, è il puntatore al primo elem.
- **Esempi:**

```
int a[10], *p, *q;
int b[ ] = { 5, 7, 8 , 2, 3 };
p = a;                  /* p = &a[0] */
q = a + 3;              /* q = &a[0] + 3 */
a++;                    /*  illegal !!! */
sum( b, 5 );            /* Equal to sum(&b[0],5) */
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori e memoria

- **Sbagliato**

```
#include <stdio.h>
int main()
{
  int *p;
  scanf("%d", p);
  return 0;
}
```

- **Corretto**

```
#include <stdio.h>
int main()
{
  int *p;
  int a;
  p = &a;
  scanf("%d", p);
  return 0;
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori e memoria

- **Spiegazione**
  - un puntatore deve puntare ad una zona di memoria effettiva
  - a parte dichiarare una variabile, esiste un altro modo per "associare memoria ad un puntatore"
- **Prototipi (stdlib.h)**
  - `void *malloc(size_t size);`
  - `void free(void * p);`
- **Spiegazione**
  - `malloc` alloca `size` bytes nello heap
    (la zona di memoria dedicata alla memoria dinamica)
  - `free` libera la memoria allocata

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori e memoria

- **Esempio:**

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int *p;
  /* Allocate 4 bytes */
  p = (int *) malloc( sizeof(int) );
  scanf("%d", p);
  printf("%d", *p);
  /* Free memory; important !!! */
  free(p);
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori e memoria

- **Dettagli aggiuntivi...**
  - per ogni chiamata a malloc, deve esserci (prima o poi) la corrispondente chiamata free
  - una tabella all'interno della libreria standard C tiene traccia dei blocchi allocati (indirizzi iniziali e loro dimensioni)
  - quando un indirizzo viene passato a free, viene cercato nella tabella e il blocco corrispondente viene "deallocato"
- **Nota: non è possibile deallocare:**
  - un blocco di memoria già deallocato
  - parte di un blocco di memoria

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori e memoria

- **Esempi**

```
int *p;
p = (int *) malloc(1000 * sizeof(int) );
for(i=0; i<1000; i++)
  p[i] = i;
p[1000]=3;          /* Wrong! */
free(p+100);        /* Wrong! */
free(p);
p[0]=5;             /* Wrong! */
free(p);            /* Wrong! */
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio: crivello di Eratostene

```
/* Print out all prime numbers
   which are less than m */
void print_prime( int m )
{
  int i,j;
  int stop;
  char *ary = malloc( m );
  if (ary == NULL) return -1;
  for(i=0; i<m; i++)
    ary[i] = 1;
  ary[0] = ary[1] = 0;
  ary[2] = 1;
  stop = 0;
  for (i=3;i<m;i++) {
    for (j=2; j<i && !stop; j++)
      if (ary[j] && i%j == 0) {
        ary[j]=0;
        stop = 1;
      }
    }
  }
  for(i=0;i<m;i++)
    if(ary[i])
      printf("%d ", i);
  free( ary );
}
```

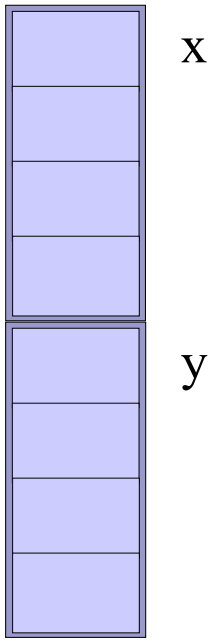
© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture

- **Definizione: *strutture***
  - Le strutture sono il meccanismo C per raggruppare insiemi di dati in singole unità singole "maneggevoli"
  - Sono anche il meccanismo "base" su cui è costruito gran parte del C++ (classi)
  - Simili a classi in Java, ma con la possibilità di definire solo attributi.
- **Per definire un tipo struttura:**

```
struct coord {
   int x ;
   int y ;
};
```

  - Questo definisce un nuovo tipo **`struct coord`**



© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture - Definizione variabili

- Per definire variabili di tipo struct - Primo approccio:

```
struct coord { int x; int y;
} first, second;    /* Definizione variabili */
```

- Per definire variabili di tipo struct - Secondo approccio:

```
struct coord { int x ; int y ;
};
struct coord first, second; /* Def. variabili */
```

- Per definire variabili di tipo struct - Terzo approccio:

```
struct coord { int x ; int y ;
};
typedef struct coord coordinate;
coordinate first, second; /* Def. variabili */
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture - Accesso ai campi

- **Per accedere ai campi di una struttura:**
  - Si utilizza l'operatore .
  - Forma generica: `structure_var.member_name`
  - Esempi:
    - `first.x = 50;`
    - `second.y = 100;`
  - Simili ai membri pubblici di una classe Java o C++
  - **`struct_var.member_name`** può essere utilizzato ovunque può essere utilizzata una variabile:
    - `printf ("%d , %d", second.x , second.y );`
    - `scanf("%d, %d", &first.x, &first.y);`

© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture - Copia

- **E' possibile copiare il contenuto di una struttura con un solo statement:**

  ```
  second = first;
  ```

  invece di scrivere

  ```
  second.x = first.x; second.y=first.y;
  ```

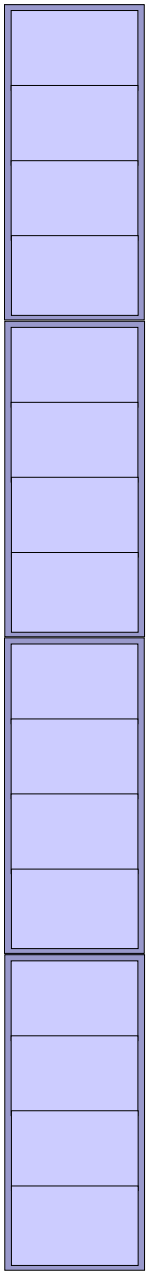
- **Vantaggi:**
  - riduce la probabilità di errore
  - è più conveniente con strutture di grandi dimensioni
- **Differenze con java**
  - le variabili non-primitive in Java sono di tipo reference
  - **`first=second`** fa in modo che sia **`first`** che **`second`** si riferiscano allo stesso oggetto

© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture - Annidate

- Oggetti di qualunque "tipo" possono far parte di una struttura:
- Esempio (rettangolo come coppia di punti):

```
struct rectangle {
   struct coord topleft;
   struct coord bottomrt;
} ;
struct rectangle mybox ;
mybox.topleft.x = 0 ;
mybox.topleft.y = 10 ;
mybox.bottomrt.x = 100 ;
mybox.bottomrt.y = 200 ;
mybox.bottomrt.x = 100 ;
mybox.bottomrt.y = 200 ;
```



© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture - Esempio

```
#include <stdio.h>
struct coord {
    int x;
    int y;
};
struct rectangle {
    struct coord topleft;
    struct coord bottomrt;
};
```

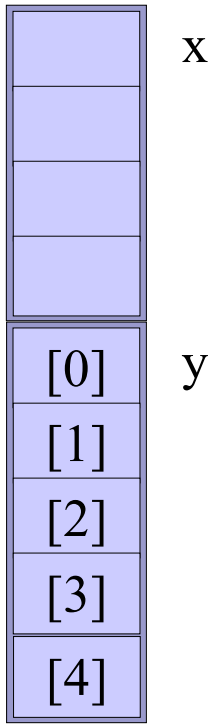
```
int main () {
  int length, width;
  long area;
  struct rectangle mybox;
  mybox.topleft.x = 0;
  mybox.topleft.y = 0;
   mybox.bottomrt.x = 100;
  mybox.bottomrt.y = 50;
  width = mybox.bottomrt.x -
    mybox.topleft.x;
  length = mybox.bottomrt.y -
    mybox.topleft.y;
  area   = width * length;
  printf ("The area is %ld units.\n",
   area);
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture contenenti array

- E' possibile inserire array in una struttura, come qualsiasi altro oggetto
- Esempio:

```
struct record {
   float  x;
   char  y[5] ;
};
struct record p;
p.x = 1.0;
p.y[0] = 0;
```



© 2002-2005 Renzo Davoli, Alberto Montresor

Array contenenti strutture

- Esempio (1000 entry cognome-nome-telefono):

```
struct  entry {
   char  fname [20] ;
   char  lname [20] ;
   char  phone [10] ;
} ;
struct entry list [1000];
```

- Assignments:

```
list [1] = list [6];
strcpy (list[1].phone, list[6].phone);
list[6].phone[1] = list[3].phone[4] ;
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio

```
#include <stdio.h>

struct entry {
    char fname [20];
    char lname [20];
    char phone [10];
};
int main() {
  struct entry list[4];
  int i;
  for (i=0; i < 4; i++) {
    ....
```

```
....
    printf ("\nEnter first name: ");
    scanf  ("%s", list[i].fname);
    printf ("Enter last name: ");
    scanf  ("%s", list[i].lname);
    printf ("Enter phone in 123-4567
      format: ");
    scanf  ("%s", list[i].phone);
  }
  printf ("\n\n");
  for (i=0; i < 4; i++) {
    printf ("Name: %s %s",
      list[i].fname, list[i].lname);
    printf ("\t\tPhone: %s\n",
    list[i].phone);
  }
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture - Inizializzazione

```
struct sale {
   char customer[20] ;
   char item[20] ;
   int amount ;
};

struct sale mysale = { "Acme Industries", "Zorgle blaster",
   1000 } ;
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture - Inizializzazione strutture annidate

```
struct customer {
  char firm [20] ;
  char contact [25] ;
};
struct sale {
  struct customer buyer ;
  char item [20] ;
  int amount ;
};
mysale = { { "Acme Industries", "George Adams"},
           "Zorgle Blaster",  1000
};
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture - Inizializzazione strutture annidate

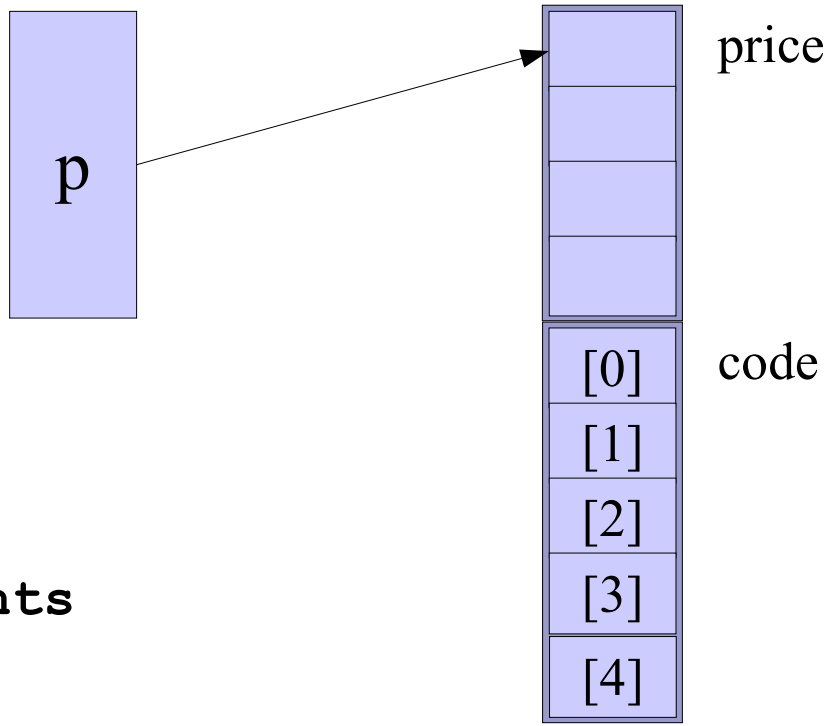
```
struct customer {
        char firm [20] ;
        char contact [25] ;
} ;
struct sale {
        struct customer buyer ;
        char item [20] ;
        int amount ;
} ;
```

```
struct sale y1990[2] =
{
   {
      { "Acme Industries",
        "George Adams"} ,
      "Left-handed Idiots",
      1000
   },
   {
      { "Wilson & Co.",
        "Ed Wilson"},
       "Thingamabob",
       290
   }
} ;
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori a struttura

```
struct part {
   float price ;
   char code[5] ;
} ;
struct part *p;
struct part thing;
p = &thing;
/* The following three statements
   are equivalent */
thing.price = 50;
(*p).price = 50;    /* () around *p is needed */
p -> price = 50;
```



© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori a struttura - Allocazione dinamica

```
struct part * p, *q;

p = (struct part *) malloc( sizeof(struct part) );

q = (struct part *) malloc( sizeof(struct part) );

p -> price = 199.99 ;

strcpy( p->code, "CFF00" );

(*q) = (*p);

q = p;

free(p);

free(q); /* This is wrong !!! Why? */
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori a struttura - Allocazione array di strutture

- E' possibile allocare una array di strutture

```
struct part *ptr;

ptr = (struct part *)
  malloc(10 * sizeof(struct part) );

for( i=0; i< 10; i++) {
  ptr[ i ].price = 10.0 * i;
  sprintf( ptr[ i ].name, "COD%d", i );
}

.....

free(ptr);
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori a struttura - Allocazione array di strutture

- E' possibile utilizzare l'aritmetica dei puntatori per accedere all'array

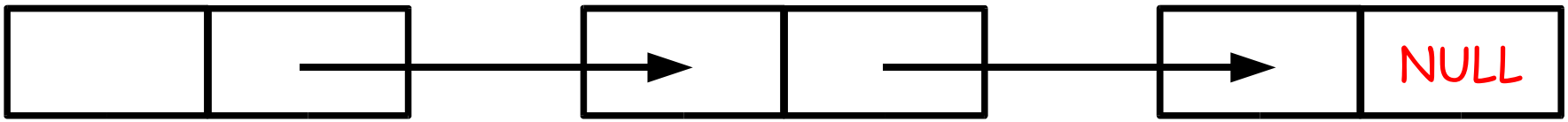
```
struct part *ptr,  *p;
ptr = (struct part *)
  malloc(10 * sizeof(struct part) );
for( i=0, p=ptr; i< 10; i++, p++) {
  p -> price = 10.0 * i;
  sprintf( p -> name, "COD%d", i );
}
.....
free(ptr);
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori come membri di struttura

```
struct node{
   int data;
   struct node *next;
};
struct node a,b,c;
a.next = &b;
b.next = &c;
c.next = NULL;
```

```
a.data = 1;
a.next->data = 2;
/* b.data =2 */
a.next->next->data = 3;
/* c.data = 3 */
```



© 2002-2005 Renzo Davoli, Alberto Montresor

Membri array vs Membri puntatori

```
struct book {
  float price;
  char name[50];
};
```

Cosa stampa?

```
int main()
{
  struct book a,b;
  b.price = 19.99;
  strcpy(b.name, "C handbook");
  a = b;
  strcpy(b.name, "Unix
   handbook");
  printf("%s\n", a.name);
  printf("%s\n, b.name);
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Membri array vs Membri puntatori

```
struct book {
   float price;
   char *name;
};
```

Cosa stampa?

```
int main()
{
   struct book a,b;
   b.price = 19.99;
   b.name = (char *) malloc
    (50);
   strcpy(b.name, "C
    handbook");
   a = b;
   strcpy(b.name, "Unix
    handbook");
   printf("%s\n",a.name);
   printf("%s\n",b.name);
   free(b.name);
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Strutture e funzioni

- **Le strutture sono passate per valore alle funzioni:**
  - i parametri sono variabili locali, a cui verranno assegnati i valori degli argomenti
  - questo significa che una struttura è copiata se viene passato come parametro
- **Problemi**
  - può essere inefficiente per strutture grandi
  - può non essere possibile modificare il contenuto di una struttura ricevuta come parametro
- **Soluzione:**
  - utilizzate un puntatore a struttura
- **NB: una struttura può essere ritornata da una funzione**

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio

```
struct book {
  float price;
  char abstract[5000];
};
void print_abstract(
  struct book *p_book)
{
  puts(p_book->abstract);
};
```

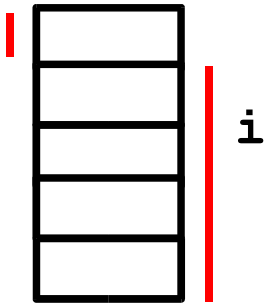
```
struct pairInt {
  int min, max;
};
struct pairInt min_max(
     int x,int y) {
  struct pairInt pair;
  pair.min = (x>y) ? y:x;
  pair.max = (x>y) ? x:y;
  return pairInt;
}
int main() {
  struct pairInt result;
  result = min_max(3,5);
  printf("%d<=%d",
   result.min,result.max);
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

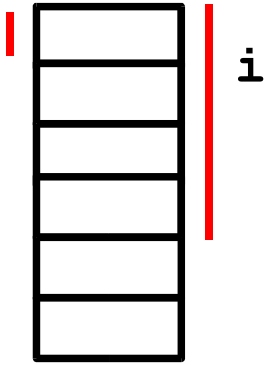
Unioni

- **Le "unioni":**
  - come le strutture, consistono di uno o più campi
  - la differenza è che nel caso delle unioni, questi campi utilizzano lo stesso spazio di memoria

```
struct number {
  char ch;
  int i;
};
```




```
union number {
  char ch;
  int i;
};
```




```
x.i = 100;
x.ch = 'a';
printf("%c,%
   d",x.ch,x.i);
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Unioni

- **Perché abbiamo bisogno delle unioni?**
  - Supponiamo di avere due tipi di merce: libri, cd
    - Libri: titolo, autore, numero di pagine
    - CD: titolo, autore, durata (minuti, secondi)
  - se volessimo raggruppare tutti questi oggetti in un array, possiamo utilizzare un solo tipo di dati per descrivere due tipi differenti di merce

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori a puntatori

- **Cos'è un puntatore a un puntatore?**
  - una variabile puntatore è una variabile che contiene un indirizzo di memoria come valore.
  - è una variabile come tutte le altre; è possibile avere un puntatore che punta ad essa
- **Esempio:**

```
int x;
int *px;
int **ppx;
ppx = &px;
px = &x;        /* i.e. *ppx = &x */
**ppx = 10;     /* i.e. *px =10;  i.e. x=10; */
ppx = (int **) malloc(sizeof(int *));
**ppx = 20;     /* Wrong, since *ppx has not been initialized
   */
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Array di puntatori

- **Un array di strutture può essere potenzialmente molto grande**
- **Per ordinare un array di strutture:**
  - sarebbe necessario un gran numero di operazioni di copia e di movimento di memoria, che possono essere costose
  - per efficienza, è possibile utilizzare un array di puntatori

© 2002-2005 Renzo Davoli, Alberto Montresor

Array di puntatori

```
struct book{
  float price;
  char abstract[5000];
};

struct book book_ary[1000];
struct book *pbook_ary[1000];
……
for (i=0;i<1000;i++)
  pbook_ary[i] = &book_ary[i];
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Array di puntatori

```
void my_sort(struct book *books[ ], int size)
{
  int i, j;
  struct book *p;
  for(i=0; i<size; i++) {
    for(j=i; j>0; j--) {
      if (books[j]->price < books[j-1]->price) {
        /* Change the order of books [j] and [j-1] */
        p=books[j];books[j]=books[j-1]; books[j-1]=p;
      }
    }
  }
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Array "triangolare"

```
const int SIZE = 10;
int main() {
  int** triangle; /* Triangular array */
  int i,j;
  triangle = (int**) malloc(sizeof(int*)*SIZE);
  for (i=0; i < SIZE; i++) {
    triangle[i] = (int*) malloc(sizeof(int)*i);
    for (j=0; j < i; j++) {
      triangle[i][j] = i*j;
    }
  }
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori const

- **Keyword const**
  - se usata prima di un parametro, indica che non modificheremo il valore del corrispondente parametro nella funzione
  - nel caso di puntatori, questo ha un significato particolare
  - Esempio: `int printf(const char * format_str, ……);`
    la stringa di formato non verrà modificata
- **Esempio:**

```
void test( const int k, const int * m)
{
k ++;        /* Warning */
(*m) ++;   /*  Warning */
m ++;        /*  Ok */
printf("%d,%d", k, *m);
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Puntatori a funzioni

- **Poiché un puntatore è semplicemente un indirizzo**
  - può puntare a qualunque cosa contenga un indirizzo
  - le funzioni possiedono un indirizzo di memoria (a partire dal punto in cui sono caricate)
  - quindi possiamo definire un puntatore a funzione

```
int (*compare)(int, int);
```

  - un puntatore a funzione
  - che prende in input due valori
  - che ritorna un valore

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio

```
#include <string.h>
typedef struct{
  float price;
  char title[100];
} book;
int (*ptr_comp)(const book *, const
   book *);
/* compare with
int  *ptr_comp (const book *, const
   book *);
*/
```

```
int compare_price(const book *
   p, const book *q)
{
  return p->price-q->price;
}

int compare_title(const book *
   p, const book *q)
{
  return strcmp(p->title,
    q-> title);
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio

```
int main( ){
  book a, b;
  a.price=19.99;
  strcpy(a.title, "unix");
  b.price=20.00;
  strcpy(b.title, "c");
  ptr_comp = compare_price;
  printf("%d",
    ptr_comp(&a, &b));
  ptr_comp = compare_title;
  printf("%d",
    ptr_comp(&a, &b));
  return 0;
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio: qsort

- **Funzione qsort()**
  - funzione standard di libreria C che permette di ordinare oggetti in base ad una funzione di confronto
- **SYNOPSIS**

```
#include <stdlib.h>
void  qsort(void  *base,  size_t  nel,  size_t  width,
            int (*compare)(const void *, const void *) );
```

- **Spiegazione:**
  - **`base`** punta al primo elemento dell'array da ordinare
  - **`nel`** contiene il numero di elementi
  - **`width`** specifica la dimensione in byte di ogni elemento
  - **`compare`** è un puntatore a funzione, che prende in input due elementi e ritorna -1, 0, +1 a seconda della loro relazione

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio: qsort

```
#include <stdlib.h>
.....
{
  book my_books[1000];
  .....
  qsort(my_books, 1000, sizeof(book),
        compare_price);
  .....
  qsort(my_books, 1000, sizeof(book),
        compare_title);
  .....
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Stringhe

- **Le stringhe in C sono semplici array di caratteri**
- **Esempio:** `char s[10];`
  - questo è uno array di dieci elementi che può contenere una stringa di al più nove caratteri (stringa null-terminated)
  - il carattere di terminazione è '\0'
- **Esempio:** `char str[10] = {'u', 'n', 'i', 'x', '\0'};`
  - è il terminatore di stringa (non la dimensione dell'array) che determina la lunghezza della stringa
- **I valori letterali per le stringhe sono racchiusi da apici doppi ""**
  - `printf("A long time ago, in a galaxy far far away...");`
- **Inizializzazione di array char**
  - `char s[10]="unix";  /* s[4] is '\0'; */`
  - `char s[ ]="unix";  /* s has five elements */`

© 2002-2005 Renzo Davoli, Alberto Montresor

Funzioni di libreria x stringhe

- **Le funzioni di stringa sono fornite in una libreria ANSI standard**
  - Accessibile tramite l'header `<string.h>`:
    ```
    #include <string.h>
    ```
- **Contiene funzioni per:**
  - calcolare la lunghezza di una stringa
  - copiare stringhe
  - concatenare stringhe
  - confrontare stringhe

© 2002-2005 Renzo Davoli, Alberto Montresor

strlen

- **Funzione `strlen`:**
  - ritorna la lunghezza di una stringa null-terminated
  - prototipo: `size_t strlen(const char *s);`
- **Note:**
  - `size_t`
    un tipo definito in string.h che è equivalente a unsigned int
  - `char* s`
    puntatore a stringa null-terminated
- **Trova il bug:**

```
char a[5]={'a', 'b', 'c', 'd', 'e'};
strlen(a);
```

© 2002-2005 Renzo Davoli, Alberto Montresor

strcpy

- **Funzione `strcpy`:**
  - copia una stringa null-terminated in un buffer destinazione
  - prototipo: `char *strcpy(char* dest, const char *src);`
- **Note:**
  - la destinazione deve avere abbastanza spazio per contenere la stringa sorgente
  - il valore di ritorno punta alla destinazione
- **Trova il bug:**

```
char a[] = "pippo";

char b[5];

strcpy(a, b);
```

© 2002-2005 Renzo Davoli, Alberto Montresor

I/O formattato: output

- **La funzione `printf` è usata per stampare informazioni sullo standard output**
  - funzione di libreria C definita in **`stdio.h`**
  - prende una stringa di formato e un elenco di parametri
- **Formato generale:** `int printf(const char *format, ...);`
- **Esempio:** `printf("The result is %d and %d\n", a, b);`
- **Il testo delle stringhe:**
  - testo letterale: viene stampato senza variazione
  - sequenze "escaped": caratteri speciali che iniziano con \
  - flag: % seguito da uno o più caratteri, indica che una variabile fra quelle che seguono devono essere stampate nello standard output

© 2002-2005 Renzo Davoli, Alberto Montresor

Flag di formato

- **Elenco di flag**
  - %c Single character
  - %d Signed decimal integer
  - %x Hexadecimal number
  - %f Decimal floating point number
  - %e Floating point in “scientific notation”
  - %s Character string
  - %u Unsigned decimal integer
  - %% Just print a % sign
- **Note:**
  - devi esserci un flag per ogni variabile da stampare
  - i tipi delle variabili e i flag devono corrispondere
  - la descrizione del formato non si esaurisce qui; consultate **`man printf`** per ulteriori informazioni

© 2002-2005 Renzo Davoli, Alberto Montresor

Caratteri di escape

- **Elenco**
  - \b Backspace
  - \n Newline
  - \t Horizontal tab
  - \\ Backslash
  - \' Single quote
  - \" Double quotation
  - \xhh ASCII char specified by hex digits hh
  - \ooo ASCII char specified by octal digits ooo

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio

- Un esempio d'uso di `printf`

```
#include <stdio.h>
int main() {
  int ten=10,x=42;
  char ch1='o', ch2='f';
  printf("%d%% %c%c %d is %f\n",
    ten,ch1,ch2,x, 1.0*x / ten );
  return 0;
}
```

- Qual è l'output?

© 2002-2005 Renzo Davoli, Alberto Montresor

I/O formattato: input

- **La funzione `scanf` è usata per leggere informazioni dallo standard input**
  - funzione di libreria C definita in **`stdio.h`**
  - prende una stringa di formato e un elenco di puntatori a zone di memoria in cui scrivere i valori letti
  - il formato è simile a quello di printf
- **Esempi:**
  - `scanf ("%d", &x);    /* reads a decimal int. */`
  - `scanf ("%f", &rate); /* reads a floating point */`

© 2002-2005 Renzo Davoli, Alberto Montresor

Argomenti di linea di comando

- **E' possibile passare agli argomenti di linea di comando ai programmi eseguibili**
  - Esempio:
    - `ls -l /etc`
    - `-l` e `/etc` sono argomenti della linea di comando
- **Parametri della funzione `main`: `argc` e `argv`**

```
int main (int argc, char *argv[ ] ) {
  /*  Statements go here */
}
```

numero di argomenti

array di argomenti

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio 1

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
  printf ("Program name: %s\n", argv [0]);
  if (argc > 1) {
    for (i=1; i<argc; i++)
      printf ("Argument %d: %s\n", i, argv[i]);
  }
  else
    puts ("No command line arguments entered.");
  return 0;
}
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio 1 – Esecuzione

- Supponiamo di aver compilato il programma precedente come myargs

```
elettra:~$ myargs first "second arg" 3  4

Program name: a.out

Argument 1: first

Argument 2: second arg

Argument 3: 3

Argument 4: 4
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Esempio 2

```
/* show file */
#include <stdio.h>
int main(int argc, char
          *argv[ ]) {
  FILE *fp;   int k;
  if(argc !=2) {
    printf("Usage: %s file\n",
           argv[0]);
    return 0;
  }
  if((fp=fopen(argv[1], "r"))
     ==NULL){
    printf("Cannot open
   file!\n");
    return 1;
  }
while((k=fgetc(fp))!=EOF)
   fputc(k, stdout);
  fclose(fp);
  return 0;
}
```

- Generalmente, `main` controlla se gli argomenti sono validi; in caso contrario stampa informazione di help

© 2002-2005 Renzo Davoli, Alberto Montresor

Preprocessore

- **Il preprocessore C**
  - modifica il vostro codice sorgente in base a "direttive del preprocessore" inserite nel vostro codice
  - le direttive del preprocessore iniziano con il carattere '#'
  - il preprocessore crea un nuovo "codice sorgente" che è il testo che viene effettivamente compilato
  - questo "codice intermedio"
    - normalmente non viene mandato in output
    - è possibile forzare il compilatore a dare in output questo codice per vedere cosa produce

© 2002-2005 Renzo Davoli, Alberto Montresor

La direttiva #include

- **Descrizione**
  - ne abbiamo già viste molte
  - include il contenuto di un file
  - il file incluso può contenere a sua volte direttive #include
- `#include <nomefile>`
  - dice al compilatore di cercare nelle librerie standard
- `#include "nomefile"`
  - dice al compilatore di cercare a partire dalla directory corrente
  - in alcuni preprocessori, *anche* nelle librerie standard

© 2002-2005 Renzo Davoli, Alberto Montresor

La direttiva #define

- **Semplice "macro substitution"**
  - esempio: **`#define text1 text2`**
  - chiede al preprocessore di trovare tutte le occorrenze di **`text1`** e di sostituirle con **`text2`**
  - normalmente utilizzato per definire costanti
    - **`#define MAX 1000`**
- **Note:**
  - separare **`text1`** e **`text2`** con uno spazio
  - non utilizzate un ; finale (perchè?)
- **Esempio:**
  - ```
    #define PRINT printf
    PRINT("hello, world");
    ```

© 2002-2005 Renzo Davoli, Alberto Montresor
    ```

Funzioni macro

- E' possibile definire macro più complesse:

```
#define max(a,b)   ( (a) > (b) ? (a) : (b) )

printf("%d", 2 * max(3+3, 7));  /* is equivalent to */

printf("%d", 2 * ( (3+3) > (7) ? (3+3) : (7) );
```

- Le parentesi sono importanti:

```
#define max(a,b)  a>b?a:b

printf("%d", 2 * max(3+3, 7));  /*is equivalent to */

printf("%d", 2 * 3+3 > 7 ? 3+3 : 7 );
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Funzioni macro

- **Le funzioni macro sono pericolose**

```
#define max(x,y) ((x)>(y)?(x):(y))
.....
int n, i=4, j=3;
n= max( i++, j);
/* Same as n= (( i++ )>( j )?( i++ ):( j )) */
printf("%d,%d,%d", n, i, j);
```

- **Qual è l'output?**
  - nel caso di funzioni macro
  - nel caso di funzioni reali

© 2002-2005 Renzo Davoli, Alberto Montresor

Compilazione condizionale

- **Le direttive `#if`, `#elif`, `#else`, `#endif` informano il preprocessore di compilare solo parte del codice**
  - possono essere utilizzate per creare codice più efficiente e più portabile
- **Struttura:**

```
#if condition_1
source_block_1
#elif condition_2
source_block_2
...
#elif condition_n
source_block_n
#else
default_source_block
#endif
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Compilazione condizionale

- E' possibile testare il valore di macro definite con `#define`
- Esempio

```
#define ENGLAND     0
#define FRANCE      1
#define ITALY 0
#if ENGLAND
#include "england.h"
#elif FRANCE
#include "france.h"
#elif ITALY
#include "italy.h"
#else
#include "canada.h"
#endif
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Compilazione condizionale

- **E' possibile aggiungere/rimuovere codice di debug dalla compilazione**
  - in fase di debug, compilate tutto, anche il codice di debug
  - in fase di rilascio del vostro codice, escludete la compilazione del codice di debug
- **Esempio:**

```
#define DEBUG 1
.....
#if DEBUG
printf("Debug: function my_sort()!\n");
#endif
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Compilazione condizionale

- **E' possibile utilizzare anche la direttiva `#ifdef`**
  - verifica se una certo nome di macro è definito
  - non ha nulla a che fare con il valore associato
- **Esempio:**

```
#define DEBUG
.....
#ifdef DEBUG
printf("Debug: function my_sort()!\n");
#endif
```

© 2002-2005 Renzo Davoli, Alberto Montresor

Compilazione condizionale

- La direttiva `#undef` rimuove la definizione di un certo parametro
- Esempio:
  - supponiamo che nella prima parte di un file, volete che DEBUG sia definito, mentre nella seconda parte volete che non sia definito
  - inserite **`#undef DEBUG`** dove necessario
- Una macro può essere settata anche a tempo di compilazione:
  - **`gcc -DDEBUG myprog.c`**

© 2002-2005 Renzo Davoli, Alberto Montresor