Pillole di programmazione in C

Come iniziare a programmare in C

Nicola Corti & Michael Sanelli

GULP - Gruppo Utenti Linux Pisa

04/05/2011

Indice

Indice

Indice

Indice

Indice

Cosa vuol dire Programmare?

Tutto parte da un **algoritmo!**

Per iniziare a programmare dobbiamo avere ben chiaro in testa cosa vogliamo fare, dobbiamo creare un **Algoritmo**.

Algoritmo

Un metodo per ottenere un certo risultato (risolvere un certo tipo di problema) attraverso un numero finito di passi.

Da Wikipedia, l'enciclopedia libera

Dobbiamo dunque capire quale sequenza di passi risolvere il nostro problema.

Tutto parte da un **algoritmo!**

Per iniziare a programmare dobbiamo avere ben chiaro in testa cosa vogliamo fare, dobbiamo creare un **Algoritmo**.

Algoritmo

Un metodo per ottenere un certo risultato (risolvere un certo tipo di problema) attraverso un numero finito di passi.

Da Wikipedia, l'enciclopedia libera

Dobbiamo dunque capire quale sequenza di passi risolvere il nostro problema.

Tutto parte da un **algoritmo!**

Per iniziare a programmare dobbiamo avere ben chiaro in testa cosa vogliamo fare, dobbiamo creare un **Algoritmo**.

Algoritmo

Un metodo per ottenere un certo risultato (risolvere un certo tipo di problema) attraverso un numero finito di passi.

Da Wikipedia, l'enciclopedia libera

Dobbiamo dunque capire quale sequenza di passi risolvere il nostro problema.

Tutto parte da un **algoritmo!**

Per iniziare a programmare dobbiamo avere ben chiaro in testa cosa vogliamo fare, dobbiamo creare un **Algoritmo**.

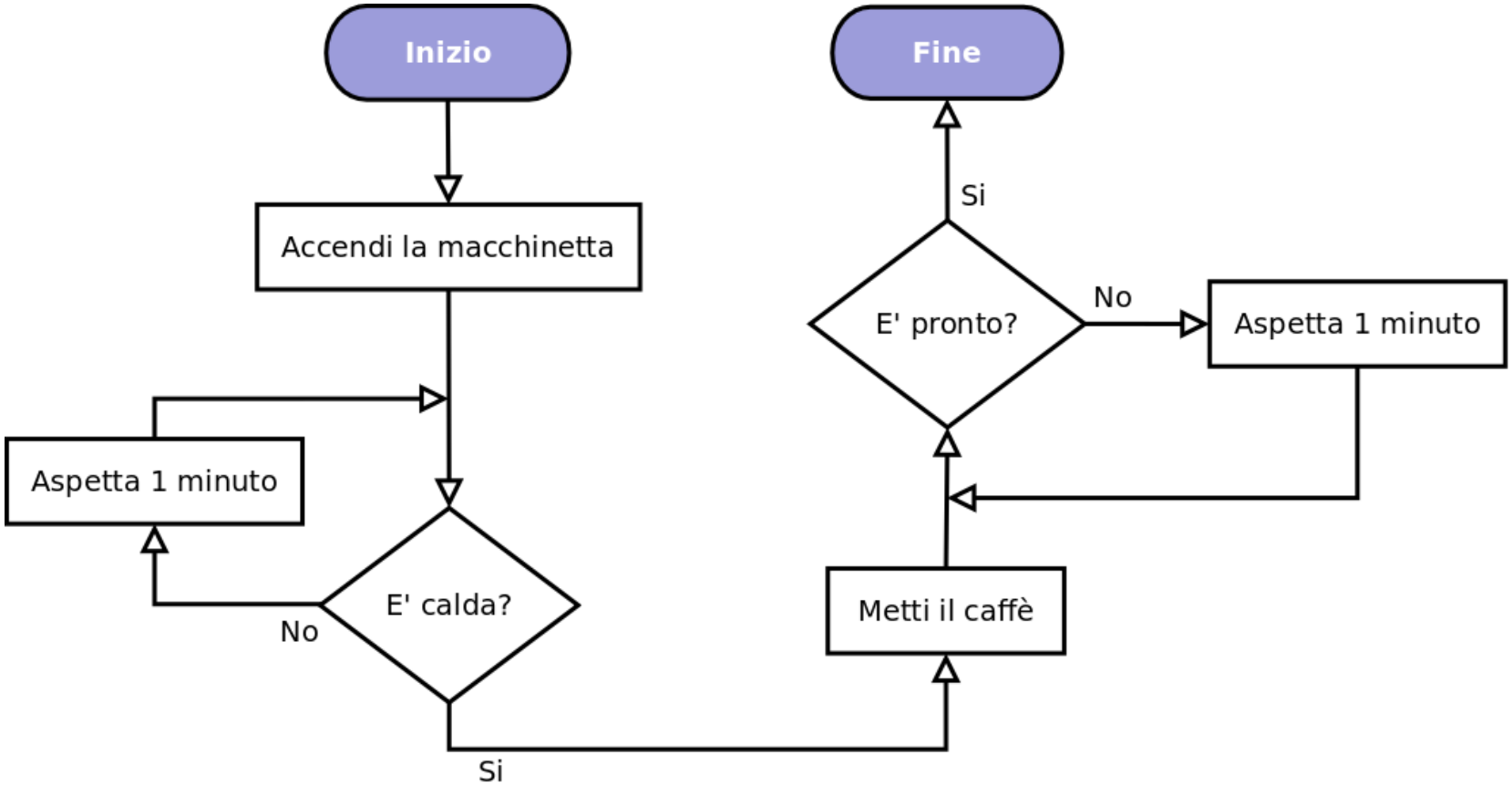
Algoritmo

Un metodo per ottenere un certo risultato (risolvere un certo tipo di problema) attraverso un numero finito di passi.

Da Wikipedia, l'enciclopedia libera

Dobbiamo dunque capire quale sequenza di passi risolvere il nostro problema.

Esempio di algoritmo



Dall'algoritmo al programma

Ora possiamo codificare il nostro algoritmo con un **Linguaggio di Programmazione**

C, C++, Java, Python, Visual Basic, Ruby, OCaml, Lisp, Perl, Prolog, etc...

Quale Scegliere?

Noi utilizzeremo un linguaggio di programmazione **Imperativo**.

Dall'algoritmo al programma

Ora possiamo codificare il nostro algoritmo con un **Linguaggio di Programmazione**

C, C++, Java, Python, Visual Basic, Ruby, OCaml, Lisp, Perl, Prolog, etc...

Quale Scegliere?

Noi utilizzeremo un linguaggio di programmazione **Imperativo**.

Dall'algoritmo al programma

Ora possiamo codificare il nostro algoritmo con un **Linguaggio di Programmazione**

C, C++, Java, Python, Visual Basic, Ruby, OCaml, Lisp, Perl, Prolog, etc...

Quale Scegliere?

Noi utilizzeremo un linguaggio di programmazione **Imperativo**.

Dall'algoritmo al programma

Ora possiamo codificare il nostro algoritmo con un **Linguaggio di Programmazione**

C, C++, Java, Python, Visual Basic, Ruby, OCaml, Lisp, Perl, Prolog, etc...

Quale Scegliere?

Noi utilizzeremo un linguaggio di programmazione **Imperativo**.

Dall'algoritmo al programma

Ora possiamo codificare il nostro algoritmo con un **Linguaggio di Programmazione**

C, C++, Java, Python, Visual Basic, Ruby, OCaml, Lisp, Perl, Prolog, etc...

Quale Scegliere?

Noi utilizzeremo un linguaggio di programmazione **Imperativo**.

Il paradigma Imperativo

Linguaggio Imperativo

In un linguaggio imperativo, un programma puó essere inteso come un **insieme di "ordini"** che vengono impartiti alla macchina virtuale del linguaggio di programmazione utilizzato

Un esempio di sequenza di passi:

```
LEGGI primo_numero
LEGGI secondo_numero
somma = primo_numero + secondo_numero
STAMPA somma
```

Il linguaggio che useremo è il C.

Il paradigma Imperativo

Linguaggio Imperativo

In un linguaggio imperativo, un programma puó essere inteso come un **insieme di "ordini"** che vengono impartiti alla macchina virtuale del linguaggio di programmazione utilizzato

Un esempio di sequenza di passi:

```
LEGGI primo_numero
LEGGI secondo_numero
somma = primo_numero + secondo_numero
STAMPA somma
```

Il linguaggio che useremo è il C.

Il paradigma Imperativo

Linguaggio Imperativo

In un linguaggio imperativo, un programma puó essere inteso come un **insieme di "ordini"** che vengono impartiti alla macchina virtuale del linguaggio di programmazione utilizzato

Un esempio di sequenza di passi:

```
LEGGI primo_numero
LEGGI secondo_numero
somma = primo_numero + secondo_numero
STAMPA somma
```

Il linguaggio che useremo è il C.

Il paradigma Imperativo

Linguaggio Imperativo

In un linguaggio imperativo, un programma puó essere inteso come un **insieme di "ordini"** che vengono impartiti alla macchina virtuale del linguaggio di programmazione utilizzato

Un esempio di sequenza di passi:

```
LEGGI primo_numero
LEGGI secondo_numero
somma = primo_numero + secondo_numero
STAMPA somma
```

Il linguaggio che useremo è il C.

Il paradigma Imperativo

Linguaggio Imperativo

In un linguaggio imperativo, un programma puó essere inteso come un **insieme di "ordini"** che vengono impartiti alla macchina virtuale del linguaggio di programmazione utilizzato

Un esempio di sequenza di passi:

```
LEGGI primo_numero
LEGGI secondo_numero
somma = primo_numero + secondo_numero
STAMPA somma
```

Il linguaggio che useremo è il C.

Quando è nato il C?

Il C è nato nei Bell Laboratories della AT&T nel 1972 da **Dennis Ritchie** come evoluzione del linguaggio B di Ken Thompson usato per la scrittura dei primi sistemi operativi UNIX.

La definizione formale si ha nel 1978 a cura di **B. W. Kernighan e D. M. Ritchie.**

Quando è nato il C?

Il C è nato nei Bell Laboratories della AT&T nel 1972 da **Dennis Ritchie** come evoluzione del linguaggio B di Ken Thompson usato per la scrittura dei primi sistemi operativi UNIX.

La definizione formale si ha nel 1978 a cura di **B. W. Kernighan e D. M. Ritchie.**

Quando è nato il C?

Il C è nato nei Bell Laboratories della AT&T nel 1972 da **Dennis Ritchie** come evoluzione del linguaggio B di Ken Thompson usato per la scrittura dei primi sistemi operativi UNIX.

La definizione formale si ha nel 1978 a cura di **B. W. Kernighan e D. M. Ritchie.**

Perchè programmare in C?

- Anche se è un linguaggio ad Alto Livello, è molto **poco astratto** (rispetto ad altri linguaggi);
- Permette di relazionarsi facilmente con l'**hardware**;
- Presenta pochi **semplici costrutti**;
- È molto ben integrato nell'ambiente **UNIX/Linux**;
- È stato definito standard **(ANSI C)**, e ció lo rende molto portabile.

Ed è comunque un buon linguaggio con cui iniziare.

Perchè programmare in C?

- Anche se è un linguaggio ad Alto Livello, è molto **poco astratto** (rispetto ad altri linguaggi);
- Permette di relazionarsi facilmente con l'**hardware**;
- Presenta pochi **semplici costrutti**;
- È molto ben integrato nell'ambiente **UNIX/Linux**;
- È stato definito standard **(ANSI C)**, e ció lo rende molto portabile.

Ed è comunque un buon linguaggio con cui iniziare.

Perchè programmare in C?

- Anche se è un linguaggio ad Alto Livello, è molto **poco astratto** (rispetto ad altri linguaggi);
- Permette di relazionarsi facilmente con l'**hardware**;
- Presenta pochi **semplici costrutti**;
- È molto ben integrato nell'ambiente **UNIX/Linux**;
- È stato definito standard **(ANSI C)**, e ció lo rende molto portabile.

Ed è comunque un buon linguaggio con cui iniziare.

Perchè programmare in C?

- Anche se è un linguaggio ad Alto Livello, è molto **poco astratto** (rispetto ad altri linguaggi);
- Permette di relazionarsi facilmente con l'**hardware**;
- Presenta pochi **semplici costrutti**;
- È molto ben integrato nell'ambiente **UNIX/Linux**;
- È stato definito standard **(ANSI C)**, e ció lo rende molto portabile.

Ed è comunque un buon linguaggio con cui iniziare.

Perchè programmare in C?

- Anche se è un linguaggio ad Alto Livello, è molto **poco astratto** (rispetto ad altri linguaggi);
- Permette di relazionarsi facilmente con l'**hardware**;
- Presenta pochi **semplici costrutti**;
- È molto ben integrato nell'ambiente **UNIX/Linux**;
- È stato definito standard **(ANSI C)**, e ció lo rende molto portabile.

Ed è comunque un buon linguaggio con cui iniziare.

Perchè programmare in C?

- Anche se è un linguaggio ad Alto Livello, è molto **poco astratto** (rispetto ad altri linguaggi);
- Permette di relazionarsi facilmente con l'**hardware**;
- Presenta pochi **semplici costrutti**;
- È molto ben integrato nell'ambiente **UNIX/Linux**;
- È stato definito standard **(ANSI C)**, e ció lo rende molto portabile.

Ed è comunque un buon linguaggio con cui iniziare.

Perchè programmare in C?

- Anche se è un linguaggio ad Alto Livello, è molto **poco astratto** (rispetto ad altri linguaggi);
- Permette di relazionarsi facilmente con l'**hardware**;
- Presenta pochi **semplici costrutti**;
- È molto ben integrato nell'ambiente **UNIX/Linux**;
- È stato definito standard **(ANSI C)**, e ció lo rende molto portabile.

Ed è comunque un buon linguaggio con cui iniziare.

Strumenti per la Programmazione

Cosa ci serve?

Per iniziare a programmare in C ci servono sostanzialmente 2 cose:

1. Un posto dove scrivere il nostro codice sorgente, chiamasi **editor di testi**;
2. Qualcuno che traduca o interpreti il codice che noi scriviamo, e che lo renda eseguibile; chiamasi (nel nostro caso) **compilatore**.

E tanta voglia di programmare! :-)

Editor di testo e complilatori su Linux

Se si usa una qualsiasi fra le principali distribuzioni Linux, questi strumenti sono giá installati (es: gcc + gedit)

Cosa ci serve?

Per iniziare a programmare in C ci servono sostanzialmente 2 cose:

1. Un posto dove scrivere il nostro codice sorgente, chiamasi **editor di testi**;
2. Qualcuno che traduca o interpreti il codice che noi scriviamo, e che lo renda eseguibile; chiamasi (nel nostro caso) **compilatore**.

E tanta voglia di programmare! :-)

Editor di testo e complilatori su Linux

Se si usa una qualsiasi fra le principali distribuzioni Linux, questi strumenti sono giá installati (es: gcc + gedit)

Cosa ci serve?

Per iniziare a programmare in C ci servono sostanzialmente 2 cose:

1. Un posto dove scrivere il nostro codice sorgente, chiamasi **editor di testi**;
2. Qualcuno che traduca o interpreti il codice che noi scriviamo, e che lo renda eseguibile; chiamasi (nel nostro caso) **compilatore**.

E tanta voglia di programmare! :-)

Editor di testo e complilatori su Linux

Se si usa una qualsiasi fra le principali distribuzioni Linux, questi strumenti sono giá installati (es: gcc + gedit)

Cosa ci serve?

Per iniziare a programmare in C ci servono sostanzialmente 2 cose:

1. Un posto dove scrivere il nostro codice sorgente, chiamasi **editor di testi**;
2. Qualcuno che traduca o interpreti il codice che noi scriviamo, e che lo renda eseguibile; chiamasi (nel nostro caso) **compilatore**.

E tanta voglia di programmare! :-)

Editor di testo e complilatori su Linux

Se si usa una qualsiasi fra le principali distribuzioni Linux, questi strumenti sono giá installati (es: gcc + gedit)

Cosa ci serve?

Per iniziare a programmare in C ci servono sostanzialmente 2 cose:

1. Un posto dove scrivere il nostro codice sorgente, chiamasi **editor di testi**;
2. Qualcuno che traduca o interpreti il codice che noi scriviamo, e che lo renda eseguibile; chiamasi (nel nostro caso) **compilatore**.

E tanta voglia di programmare! :-)

Editor di testo e complilatori su Linux

Se si usa una qualsiasi fra le principali distribuzioni Linux, questi strumenti sono giá installati (es: gcc + gedit)

Cosa ci serve?

Per iniziare a programmare in C ci servono sostanzialmente 2 cose:

1. Un posto dove scrivere il nostro codice sorgente, chiamasi **editor di testi**;
2. Qualcuno che traduca o interpreti il codice che noi scriviamo, e che lo renda eseguibile; chiamasi (nel nostro caso) **compilatore**.

E tanta voglia di programmare! :-)

Editor di testo e complilatori su Linux

Se si usa una qualsiasi fra le principali distribuzioni Linux, questi strumenti sono giá installati (es: gcc + gedit)

L'editor di testi

Per iniziare, va bene un qualsiasi software che permetta di scrivere e di salvare del testo semplice.
Possono andar bene i seguenti

- **gedit** - di default nell'ambiente GNOME
- **vi/vim**
- **emacs**
- etc...

Tutti i software sopra elencati forniscono inoltre *l'highlight della sintassi*, strumento molto utile per la leggibitá del codice.

L'editor di testi

Per iniziare, va bene un qualsiasi software che permetta di scrivere e di salvare del testo semplice.

Possono andar bene i seguenti

- **gedit** - di default nell'ambiente GNOME
- **vi/vim**
- **emacs**
- etc...

Tutti i software sopra elencati forniscono inoltre *l'highlight della sintassi*, strumento molto utile per la leggibitá del codice.

L'editor di testi

Per iniziare, va bene un qualsiasi software che permetta di scrivere e di salvare del testo semplice.
Possono andar bene i seguenti

- **gedit** - di default nell'ambiente GNOME
- **vi/vim**
- **emacs**
- etc...

Tutti i software sopra elencati forniscono inoltre *l'highlight della sintassi*, strumento molto utile per la leggibitá del codice.

L'editor di testi

Per iniziare, va bene un qualsiasi software che permetta di scrivere e di salvare del testo semplice.
Possono andar bene i seguenti

- **gedit** - di default nell'ambiente GNOME
- **vi/vim**
- **emacs**
- etc...

Tutti i software sopra elencati forniscono inoltre *l'highlight della sintassi*, strumento molto utile per la leggibitá del codice.

L'editor di testi

Per iniziare, va bene un qualsiasi software che permetta di scrivere e di salvare del testo semplice.
Possono andar bene i seguenti

- **gedit** - di default nell'ambiente GNOME
- **vi/vim**
- **emacs**
- etc...

Tutti i software sopra elencati forniscono inoltre *l'highlight della sintassi*, strumento molto utile per la leggibitá del codice.

L'editor di testi

Per iniziare, va bene un qualsiasi software che permetta di scrivere e di salvare del testo semplice.
Possono andar bene i seguenti

- **gedit** - di default nell'ambiente GNOME
- **vi/vim**
- **emacs**
- etc...

Tutti i software sopra elencati forniscono inoltre *l'highlight della sintassi*, strumento molto utile per la leggibitá del codice.

L'editor di testi

Per iniziare, va bene un qualsiasi software che permetta di scrivere e di salvare del testo semplice.
Possono andar bene i seguenti

- **gedit** - di default nell'ambiente GNOME
- **vi/vim**
- **emacs**
- etc...

Tutti i software sopra elencati forniscono inoltre *l'highlight della sintassi*, strumento molto utile per la leggibitá del codice.

L'editor di testi

Per iniziare, va bene un qualsiasi software che permetta di scrivere e di salvare del testo semplice.
Possono andar bene i seguenti

- **gedit** - di default nell'ambiente GNOME
- **vi/vim**
- **emacs**
- etc...

Tutti i software sopra elencati forniscono inoltre *l'highlight della sintassi*, strumento molto utile per la leggibitá del codice.

Il Compilatore

Utilizzeremo **GCC (GNU Compiler Collection)** il compilatore del progetto GNU.

Invocazione

Puó essere invocato facilmente da un terminale:

```
bash:~$ gcc file.c
```

Se la compilazione è terminata correttamente, ci restituirá un file `a.out` che possiamo eseguire

Il Compilatore

Utilizzeremo **GCC (GNU Compiler Collection)** il compilatore del progetto GNU.

Invocazione

Puó essere invocato facilmente da un terminale:

```
bash:~$ gcc file.c
```

Se la compilazione è terminata correttamente, ci restituirá un file `a.out` che possiamo eseguire

Il Compilatore

Utilizzeremo **GCC (GNU Compiler Collection)** il compilatore del progetto GNU.

Invocazione

Puó essere invocato facilmente da un terminale:

```
bash:~$ gcc file.c
```

Se la compilazione è terminata correttamente, ci restituirá un file `a.out` che possiamo eseguire

Il Compilatore

Utilizzeremo **GCC (GNU Compiler Collection)** il compilatore del progetto GNU.

Invocazione

Puó essere invocato facilmente da un terminale:

```
bash:~$ gcc file.c
```

Se la compilazione è terminata correttamente, ci restituirá un file `a.out` che possiamo eseguire

Parametri di GCC

La sintassi con cui invocare gcc è la seguente:

```
gcc [opzioni] file
```

Fra le principali opzioni vi sono:

- `-o` seguito da un nome di file, indica dove salvare il file compilato
- `-Wall` indica a gcc di stampare tutti gli Warning che riscontra durante la compilazione
- `-pedantic` indica a gcc di generare degli Warning se il codice sorgente non rispetta gli standard ISO C
- `-g` permette di eseguire il codice compilato in un debugger (ne parleremo dopo...)

Parametri di GCC

La sintassi con cui invocare gcc è la seguente:

```
gcc [opzioni] file
```

Fra le principali opzioni vi sono:

- **-o** seguito da un nome di file, indica dove salvare il file compilato
- **-Wall** indica a gcc di stampare tutti gli Warning che riscontra durante la compilazione
- **-pedantic** indica a gcc di generare degli Warning se il codice sorgente non rispetta gli standard ISO C
- **-g** permette di eseguire il codice compilato in un debugger (ne parleremo dopo...)

Parametri di GCC

La sintassi con cui invocare gcc è la seguente:

```
gcc [opzioni] file
```

Fra le principali opzioni vi sono:

- **-o** seguito da un nome di file, indica dove salvare il file compilato
- **-Wall** indica a gcc di stampare tutti gli Warning che riscontra durante la compilazione
- **-pedantic** indica a gcc di generare degli Warning se il codice sorgente non rispetta gli standard ISO C
- **-g** permette di eseguire il codice compilato in un debugger (ne parleremo dopo...)

Parametri di GCC

La sintassi con cui invocare gcc è la seguente:

```
gcc [opzioni] file
```

Fra le principali opzioni vi sono:

- **-o** seguito da un nome di file, indica dove salvare il file compilato
- **-Wall** indica a gcc di stampare tutti gli Warning che riscontra durante la compilazione
- **-pedantic** indica a gcc di generare degli Warning se il codice sorgente non rispetta gli standard ISO C
- **-g** permette di eseguire il codice compilato in un debugger (ne parleremo dopo...)

Parametri di GCC

La sintassi con cui invocare gcc è la seguente:

```
gcc [opzioni] file
```

Fra le principali opzioni vi sono:

- **-o** seguito da un nome di file, indica dove salvare il file compilato
- **-Wall** indica a gcc di stampare tutti gli Warning che riscontra durante la compilazione
- **-pedantic** indica a gcc di generare degli Warning se il codice sorgente non rispetta gli standard ISO C
- **-g** permette di eseguire il codice compilato in un debugger (ne parleremo dopo...)

Parametri di GCC

La sintassi con cui invocare gcc è la seguente:

```
gcc [opzioni] file
```

Fra le principali opzioni vi sono:

- **-o** seguito da un nome di file, indica dove salvare il file compilato
- **-Wall** indica a gcc di stampare tutti gli Warning che riscontra durante la compilazione
- **-pedantic** indica a gcc di generare degli Warning se il codice sorgente non rispetta gli standard ISO C
- **-g** permette di eseguire il codice compilato in un debugger (ne parleremo dopo...)

Parametri di GCC

La sintassi con cui invocare gcc è la seguente:

```
gcc [opzioni] file
```

Fra le principali opzioni vi sono:

- **-o** seguito da un nome di file, indica dove salvare il file compilato
- **-Wall** indica a gcc di stampare tutti gli Warning che riscontra durante la compilazione
- **-pedantic** indica a gcc di generare degli Warning se il codice sorgente non rispetta gli standard ISO C
- **-g** permette di eseguire il codice compilato in un debugger (ne parleremo dopo...)

Parametri di GCC

La sintassi con cui invocare gcc è la seguente:

```
gcc [opzioni] file
```

Fra le principali opzioni vi sono:

- **-o** seguito da un nome di file, indica dove salvare il file compilato
- **-Wall** indica a gcc di stampare tutti gli Warning che riscontra durante la compilazione
- **-pedantic** indica a gcc di generare degli Warning se il codice sorgente non rispetta gli standard ISO C
- **-g** permette di eseguire il codice compilato in un debugger (ne parleremo dopo...)

Utilizzare un IDE

Se abbiamo bisogno di qualcosa di piú soffisticato possiamo utilizzare un **Ambiente di sviluppo (IDE)**, quali ad esempio Eclipse, Netbeans, etc...

Ci offrono molti strumenti:

- Compilatore integrato
- Debugger
- Gestore delle prestazioni
- Controllo versione
- etc...

In genere sono sconsigliati a chi inizia a programmare, onde evitare di "perdersi" nell'infinitá di opzioni...

Utilizzare un IDE

Se abbiamo bisogno di qualcosa di piú soffisticato possiamo utilizzare un **Ambiente di sviluppo (IDE)**, quali ad esempio Eclipse, Netbeans, etc...

Ci offrono molti strumenti:

- Compilatore integrato
- Debugger
- Gestore delle prestazioni
- Controllo versione
- etc...

In genere sono sconsigliati a chi inizia a programmare, onde evitare di "perdersi" nell'infinitá di opzioni...

Utilizzare un IDE

Se abbiamo bisogno di qualcosa di piú soffisticato possiamo utilizzare un **Ambiente di sviluppo (IDE)**, quali ad esempio Eclipse, Netbeans, etc...

Ci offrono molti strumenti:

- Compilatore integrato
- Debugger
- Gestore delle prestazioni
- Controllo versione
- etc...

In genere sono sconsigliati a chi inizia a programmare, onde evitare di "perdersi" nell'infinitá di opzioni...

Utilizzare un IDE

Se abbiamo bisogno di qualcosa di piú soffisticato possiamo utilizzare un **Ambiente di sviluppo (IDE)**, quali ad esempio Eclipse, Netbeans, etc...

Ci offrono molti strumenti:

- Compilatore integrato
- Debugger
- Gestore delle prestazioni
- Controllo versione
- etc...

In genere sono sconsigliati a chi inizia a programmare, onde evitare di "perdersi" nell'infinitá di opzioni...

Utilizzare un IDE

Se abbiamo bisogno di qualcosa di piú soffisticato possiamo utilizzare un **Ambiente di sviluppo (IDE)**, quali ad esempio Eclipse, Netbeans, etc...

Ci offrono molti strumenti:

- Compilatore integrato
- Debugger
- Gestore delle prestazioni
- Controllo versione
- etc...

In genere sono sconsigliati a chi inizia a programmare, onde evitare di "perdersi" nell'infinitá di opzioni...

Utilizzare un IDE

Se abbiamo bisogno di qualcosa di piú soffisticato possiamo utilizzare un **Ambiente di sviluppo (IDE)**, quali ad esempio Eclipse, Netbeans, etc...

Ci offrono molti strumenti:

- Compilatore integrato
- Debugger
- Gestore delle prestazioni
- Controllo versione
- etc...

In genere sono sconsigliati a chi inizia a programmare, onde evitare di "perdersi" nell'infinitá di opzioni...

Utilizzare un IDE

Se abbiamo bisogno di qualcosa di piú soffisticato possiamo utilizzare un **Ambiente di sviluppo (IDE)**, quali ad esempio Eclipse, Netbeans, etc...

Ci offrono molti strumenti:

- Compilatore integrato
- Debugger
- Gestore delle prestazioni
- Controllo versione
- etc...

In genere sono sconsigliati a chi inizia a programmare, onde evitare di "perdersi" nell'infinitá di opzioni...

Utilizzare un IDE

Se abbiamo bisogno di qualcosa di piú soffisticato possiamo utilizzare un **Ambiente di sviluppo (IDE)**, quali ad esempio Eclipse, Netbeans, etc...

Ci offrono molti strumenti:

- Compilatore integrato
- Debugger
- Gestore delle prestazioni
- Controllo versione
- etc...

In genere sono sconsigliati a chi inizia a programmare, onde evitare di "perdersi" nell'infinitá di opzioni...

Utilizzare un IDE

Se abbiamo bisogno di qualcosa di piú soffisticato possiamo utilizzare un **Ambiente di sviluppo (IDE)**, quali ad esempio Eclipse, Netbeans, etc...

Ci offrono molti strumenti:

- Compilatore integrato
- Debugger
- Gestore delle prestazioni
- Controllo versione
- etc...

In genere sono sconsigliati a chi inizia a programmare, onde evitare di "perdersi" nell'infinitá di opzioni...

Costrutti di base

Hello World!

Iniziamo con un primo programma di esempio:

```c
#include <stdio.h>

int main(){

    printf("Hello world! \n");
    return 0;
}
```

Lo possiamo compilare ed eseguire facilmente cosí:

```
bash:~$ gcc helloworld.c
bash:~$ ./a.out
Hello world!
bash:~$
```

Hello World!

Iniziamo con un primo programma di esempio:

```c
#include <stdio.h>

int main(){

    printf("Hello world! \n");
    return 0;
}
```

Lo possiamo compilare ed eseguire facilmente cosí:

```
bash:~$ gcc helloworld.c
bash:~$ ./a.out
Hello world!
bash:~$
```

Hello World!

Iniziamo con un primo programma di esempio:

```
1  #include <stdio.h>
2
3  int main(){
4
5      printf("Hello world! \n");
6      return 0;
7  }
```

Lo possiamo compilare ed eseguire facilmente cosí:

```
bash:~$ gcc helloworld.c
bash:~$ ./a.out
Hello world!
bash:~$
```

Hello World!

Iniziamo con un primo programma di esempio:

```
1  #include <stdio.h>
2
3  int main(){
4
5      printf("Hello world! \n");
6      return 0;
7  }
```

Lo possiamo compilare ed eseguire facilmente cosí:

```
bash:~$ gcc helloworld.c
bash:~$ ./a.out
Hello world!
bash:~$
```

Hello World!

Iniziamo con un primo programma di esempio:

```
1  #include <stdio.h>
2
3  int main(){
4
5      printf("Hello world! \n");
6      return 0;
7  }
```

Lo possiamo compilare ed eseguire facilmente cosí:

```
bash:~$ gcc helloworld.c
bash:~$ ./a.out
Hello world!
bash:~$
```

Regole sintattiche di base

- Ogni istruzione deve essere terminata da un ";"
- È bene scrivere ogni istruzione su una riga a parte...
- ...ed utilizzare un'**indentazione**;
- Ed inserire dei commenti per spiegare le parti "critiche".

```
1  if(a < b)
2      a = b + 1;
3
4  // Commento su una riga
5  /*
6     Commento su pi\'u righe
7  */
```

Regole sintattiche di base

- Ogni istruzione deve essere terminata da un ";"
- È bene scrivere ogni istruzione su una riga a parte...
- ...ed utilizzare un'**indentazione**;
- Ed inserire dei commenti per spiegare le parti "critiche".

```
1  if(a < b)
2      a = b + 1;
3
4  // Commento su una riga
5  /*
6     Commento su pi\'u righe
7  */
```

Regole sintattiche di base

- Ogni istruzione deve essere terminata da un ";"
- È bene scrivere ogni istruzione su una riga a parte...
- ...ed utilizzare un'**indentazione**;
- Ed inserire dei commenti per spiegare le parti "critiche".

```
1  if(a < b)
2      a = b + 1;
3
4  // Commento su una riga
5  /*
6     Commento su pi\'u righe
7  */
```

Regole sintattiche di base

- Ogni istruzione deve essere terminata da un ";"
- È bene scrivere ogni istruzione su una riga a parte...
- ...ed utilizzare un'**indentazione**;
- Ed inserire dei commenti per spiegare le parti "critiche".

```
1  if(a < b)
2      a = b + 1;
3
4  // Commento su una riga
5  /*
6     Commento su pi\'u righe
7  */
```

Regole sintattiche di base

- Ogni istruzione deve essere terminata da un ";"
- È bene scrivere ogni istruzione su una riga a parte...
- ...ed utilizzare un'**indentazione**;
- Ed inserire dei commenti per spiegare le parti "critiche".

```
1  if(a < b)
2      a = b + 1;
3
4  // Commento su una riga
5  /*
6     Commento su pi\'u righe
7  */
```

Regole sintattiche di base

- Ogni istruzione deve essere terminata da un ";"
- È bene scrivere ogni istruzione su una riga a parte...
- ...ed utilizzare un'**indentazione**;
- Ed inserire dei commenti per spiegare le parti "critiche".

```
1  if(a < b)
2       a = b + 1;
3
4  // Commento su una riga
5  /*
6     Commento su pi\'u righe
7  */
```

Le direttive al preprocessore

Le istruzioni che iniziano con # sono anche chiamate le **direttive al preprocessore**.

Vengono eseguite prima che il file venga effettivamente compilato.

Le utilizziamo per includere delle librerie, o per definire delle macro.

```
1 #include <stdlib.h>
2 /* Inclusione della libreria standard C */
3
4 #define LATI 4 /* Definizione di macro */
```

Per vedere cosa fa il preprocessore, provare ad invocare **cpp** sul file sorgente...

Le direttive al preprocessore

Le istruzioni che iniziano con # sono anche chiamate le **direttive al preprocessore**.

Vengono eseguite prima che il file venga effettivamente compilato.

Le utilizziamo per includere delle librerie, o per definire delle macro.

```
1 #include <stdlib.h>
2 /* Inclusione della libreria standard C */
3
4 #define LATI 4 /* Definizione di macro */
```

Per vedere cosa fa il preprocessore, provare ad invocare **cpp** sul file sorgente...

Le direttive al preprocessore

Le istruzioni che iniziano con # sono anche chiamate le **direttive al preprocessore**.

Vengono eseguite prima che il file venga effettivamente compilato.

Le utilizziamo per includere delle librerie, o per definire delle macro.

```
1 #include <stdlib.h>
2 /* Inclusione della libreria standard C */
3
4 #define LATI 4 /* Definizione di macro */
```

Per vedere cosa fa il preprocessore, provare ad invocare **cpp** sul file sorgente...

Le direttive al preprocessore

Le istruzioni che iniziano con # sono anche chiamate le **direttive al preprocessore**.

Vengono eseguite prima che il file venga effettivamente compilato.

Le utilizziamo per includere delle librerie, o per definire delle macro.

```
1 #include <stdlib.h>
2 /* Inclusione della libreria standard C */
3
4 #define LATI 4 /* Definizione di macro */
```

Per vedere cosa fa il preprocessore, provare ad invocare **cpp** sul file sorgente...

Le direttive al preprocessore

Le istruzioni che iniziano con # sono anche chiamate le **direttive al preprocessore**.

Vengono eseguite prima che il file venga effettivamente compilato.

Le utilizziamo per includere delle librerie, o per definire delle macro.

```
1 #include <stdlib.h>
2 /* Inclusione della libreria standard C */
3
4 #define LATI 4 /* Definizione di macro */
```

Per vedere cosa fa il preprocessore, provare ad invocare **cpp** sul file sorgente...

Le direttive al preprocessore

Le istruzioni che iniziano con # sono anche chiamate le **direttive al preprocessore**.

Vengono eseguite prima che il file venga effettivamente compilato.

Le utilizziamo per includere delle librerie, o per definire delle macro.

```
1 #include <stdlib.h>
2 /* Inclusione della libreria standard C */
3
4 #define LATI 4 /* Definizione di macro */
```

Per vedere cosa fa il preprocessore, provare ad invocare **cpp** sul file sorgente...

Le variabili

Variabile

Per **variabile** intendiamo una locazione di memoria che possiamo utilizzare per "salvare" un dato, in modo da poterlo riutilizzare in futuro (leggerlo o modificarlo).

In C dobbiamo dichiarare ogni variabile che utilizziamo,

All'atto della dichiarazione dobbiamo deciderne il **tipo**.

Le variabili

Variabile

Per **variabile** intendiamo una locazione di memoria che possiamo utilizzare per "salvare" un dato, in modo da poterlo riutilizzare in futuro (leggerlo o modificarlo).

In C dobbiamo dichiarare ogni variabile che utilizziamo,

All'atto della dichiarazione dobbiamo deciderne il **tipo**.

Le variabili

Variabile

Per **variabile** intendiamo una locazione di memoria che possiamo utilizzare per "salvare" un dato, in modo da poterlo riutilizzare in futuro (leggerlo o modificarlo).

In C dobbiamo dichiarare ogni variabile che utilizziamo,

All'atto della dichiarazione dobbiamo deciderne il **tipo**.

Le variabili

Variabile

Per **variabile** intendiamo una locazione di memoria che possiamo utilizzare per "salvare" un dato, in modo da poterlo riutilizzare in futuro (leggerlo o modificarlo).

In C dobbiamo dichiarare ogni variabile che utilizziamo,

All'atto della dichiarazione dobbiamo deciderne il **tipo**.

Le variabili

Per definirle usiamo la sintassi:

```
<tipo> <nomevar> [= <valiniz>];
```

```
1 int a;              /* Var. di tipo Intero */
2 int somma = 0;      /* Dich. con inizializzazione */
3 int num1, num2;     /* Dich. compatta */
4 char b;             /* Var. di tipo Carattere */
```

Le variabili

Per definirle usiamo la sintassi:

`<tipo> <nomevar> [= <valiniz>];`

```
1 int a;              /* Var. di tipo Intero */
2 int somma = 0;      /* Dich. con inizializzazione */
3 int num1, num2;     /* Dich. compatta */
4 char b;             /* Var. di tipo Carattere */
```

Le variabili

Per definirle usiamo la sintassi:

`<tipo> <nomevar> [= <valiniz>];`

```
1  int a;              /* Var. di tipo Intero */
2  int somma = 0;      /* Dich. con inizializzazione */
3  int num1, num2;     /* Dich. compatta */
4  char b;             /* Var. di tipo Carattere */
```

Assegnamenti ed espressioni

Assegnamenti ed espressioni sono i costrutti base per creare il nostro programma.

La sintassi generale è:

```
<variabile> = <espressione>
```

L'espressione puó essere una normale espressione algebrica, oppure puó contenere anche delle **funzioni**:

```
1  a = b;
2  a = a + 1;
3  a = a + (b * 4 - 1);
4  a = somma(a, 5) + differenza(b, 10);
```

Assegnamenti ed espressioni

Assegnamenti ed espressioni sono i costrutti base per creare il nostro programma.

La sintassi generale è:

```
<variabile> = <espressione>
```

L'espressione puó essere una normale espressione algebrica, oppure puó contenere anche delle **funzioni**:

```
1  a = b;
2  a = a + 1;
3  a = a + (b * 4 - 1);
4  a = somma(a, 5) + differenza(b, 10);
```

Assegnamenti ed espressioni

Assegnamenti ed espressioni sono i costrutti base per creare il nostro programma.

La sintassi generale è:

```
<variabile> = <espressione>
```

L'espressione puó essere una normale espressione algebrica, oppure puó contenere anche delle **funzioni**:

```
1  a = b;
2  a = a + 1;
3  a = a + (b * 4 - 1);
4  a = somma(a, 5) + differenza(b, 10);
```

Assegnamenti ed espressioni

Assegnamenti ed espressioni sono i costrutti base per creare il nostro programma.

La sintassi generale è:

```
<variabile> = <espressione>
```

L'espressione puó essere una normale espressione algebrica, oppure puó contenere anche delle **funzioni**:

```
1  a = b;
2  a = a + 1;
3  a = a + (b * 4 - 1);
4  a = somma(a, 5) + differenza(b, 10);
```

Assegnamenti ed espressioni

Assegnamenti ed espressioni sono i costrutti base per creare il nostro programma.

La sintassi generale è:

```
<variabile> = <espressione>
```

L'espressione puó essere una normale espressione algebrica, oppure puó contenere anche delle **funzioni**:

```
1  a = b;
2  a = a + 1;
3  a = a + (b * 4 - 1);
4  a = somma(a, 5) + differenza(b, 10);
```

Espressioni - Forme compatte

Esistono delle forme compatte per le espressioni.

Servono per rendere piú sintetico il codice, ma lo possono rendere **"poco leggibile"**

```
1  a = a + 5;
2  a += 5;
3
4  a = a * 10;
5  a *= 10;
6
7  a = a + 1;
8  a++;
```

I principali operatori aritmetici sono: +, -, *, / (div.), % (mod.)

Espressioni - Forme compatte

Esistono delle forme compatte per le espressioni.
Servono per rendere piú sintetico il codice, ma lo possono rendere **“poco leggibile”**

```
1  a = a + 5;
2  a += 5;
3
4  a = a * 10;
5  a *= 10;
6
7  a = a + 1;
8  a++;
```

I principali operatori aritmetici sono: +, -, *, / (div.), % (mod.)

Espressioni - Forme compatte

Esistono delle forme compatte per le espressioni.
Servono per rendere piú sintetico il codice, ma lo possono rendere **“poco leggibile”**

```
1  a = a + 5;
2  a += 5;
3
4  a = a * 10;
5  a *= 10;
6
7  a = a + 1;
8  a++;
```

I principali operatori aritmetici sono: +, -, *, / (div.), % (mod.)

Espressioni - Forme compatte

Esistono delle forme compatte per le espressioni.
Servono per rendere piú sintetico il codice, ma lo possono rendere **“poco leggibile”**

```
1  a = a + 5;
2  a += 5;
3
4  a = a * 10;
5  a *= 10;
6
7  a = a + 1;
8  a++;
```

I principali operatori aritmetici sono: +, -, *, / (div.), % (mod.)

Costrutto ifthenelse

Un altro costrutto molto importante è il costrutto **ifthenelse**.

La sintassi di base è:

```
if(<condizione>)
    {<istruzioni ramo then>}
[else
    {<istruzioni ramo else>}]
```

```
1 if(a == b)
2     {a = 0;}
3 else
4     {b = 0;}
```

Costrutto ifthenelse

Un altro costrutto molto importante è il costrutto **ifthenelse**.

La sintassi di base è:

```
if(<condizione>)
    {<istruzioni ramo then>}
[else
    {<istruzioni ramo else>}]
```

```
1 if(a == b)
2     {a = 0;}
3 else
4     {b = 0;}
```

Costrutto ifthenelse

Un altro costrutto molto importante è il costrutto **ifthenelse**.

La sintassi di base è:

```
if(<condizione>)
    {<istruzioni ramo then>}
[else
    {<istruzioni ramo else>}]
```

```
1 if(a == b)
2     {a = 0;}
3 else
4     {b = 0;}
```

Costrutto ifthenelse

Un altro costrutto molto importante è il costrutto **ifthenelse**.

La sintassi di base è:

```
if(<condizione>)
    {<istruzioni ramo then>}
[else
    {<istruzioni ramo else>}]
```

```
1  if(a == b)
2      {a = 0;}
3  else
4      {b = 0;}
```

Operatori di confronto

All'interno di una condizione possiamo utilizzare i seguenti operatorio di confronto:

==	Uguale
!=	Diverso
<,>	Minore/Maggiore
<=,>=	Minore/Maggiore o Uguale

Ed i seguenti operatori logici:

&	And
||	Or
!	Not

Operatori di confronto

All'interno di una condizione possiamo utilizzare i seguenti operatorio di confronto:

==	Uguale
!=	Diverso
<,>	Minore/Maggiore
<=,>=	Minore/Maggiore o Uguale

Ed i seguenti operatori logici:

&	And
||	Or
!	Not

Operatori di confronto

All'interno di una condizione possiamo utilizzare i seguenti operatorio di confronto:

==	Uguale
!=	Diverso
<,>	Minore/Maggiore
<=,>=	Minore/Maggiore o Uguale

Ed i seguenti operatori logici:

&	And
||	Or
!	Not

Operatori di confronto

All'interno di una condizione possiamo utilizzare i seguenti operatorio di confronto:

==	Uguale
!=	Diverso
<,>	Minore/Maggiore
<=,>=	Minore/Maggiore o Uguale

Ed i seguenti operatori logici:

&	And
||	Or
!	Not

Costrutto while

Tramite il costrutto **while** possiamo definire dei cicli che iterano fino al falsificarsi di una certa condizione, la sintassi è:

```
while(<condizione>)
    {<corpo del ciclo>}
```

```
1 while(a != b)
2     {a++;
3     somma++;}
```

Attenzione!

Se non scriviamo bene il ciclo e la condizione, il nostro programma rischia di andare in *loop*

Costrutto while

Tramite il costrutto **while** possiamo definire dei cicli che iterano fino al falsificarsi di una certa condizione, la sintassi è:

```
while(<condizione>)
    {<corpo del ciclo>}
```

```
1 while(a != b)
2     {a++;
3     somma++;}
```

Attenzione!

Se non scriviamo bene il ciclo e la condizione, il nostro programma rischia di andare in *loop*

Costrutto while

Tramite il costrutto **while** possiamo definire dei cicli che iterano fino al falsificarsi di una certa condizione, la sintassi è:

```
while(<condizione>)
    {<corpo del ciclo>}
```

```
1  while(a != b)
2       {a++;
3       somma++;}
```

Attenzione!

Se non scriviamo bene il ciclo e la condizione, il nostro programma rischia di andare in *loop*

Costrutto while

Tramite il costrutto **while** possiamo definire dei cicli che iterano fino al falsificarsi di una certa condizione, la sintassi è:

```
while(<condizione>)
    {<corpo del ciclo>}
```

```
1  while(a != b)
2       {a++;
3       somma++;}
```

Attenzione!

Se non scriviamo bene il ciclo e la condizione, il nostro programma rischia di andare in *loop*

Costrutto while

Tramite il costrutto **while** possiamo definire dei cicli che iterano fino al falsificarsi di una certa condizione, la sintassi è:

```
while(<condizione>)
    {<corpo del ciclo>}
```

```
1  while(a != b)
2       {a++;
3       somma++;}
```

Attenzione!

Se non scriviamo bene il ciclo e la condizione, il nostro programma rischia di andare in *loop*

Costrutto for

Mediante il costrutto **for** possiamo scrivere dei cicli piú compatti.
La sintassi di base è:

```
for(<init.>; <terminaz.>; <iteraz.>)
    {<corpo del ciclo>}
```

```
1 for(i = 0; i < MAX; i++)
2     {a = a + 1;}
```

While vs for

Dovremmo utilizzare il ciclo for soltanto quando il numero di iterazioni è noto a priori, ed utilizzare il ciclo while negli altri casi.

Costrutto for

Mediante il costrutto **for** possiamo scrivere dei cicli piú compatti.
La sintassi di base è:

```
for(<init.>; <terminaz.>; <iteraz.>)
    {<corpo del ciclo>}
```

```
1 for(i = 0; i < MAX; i++)
2     {a = a + 1;}
```

While vs for

Dovremmo utilizzare il ciclo for soltanto quando il numero di iterazioni è noto a priori, ed utilizzare il ciclo while negli altri casi.

Costrutto for

Mediante il costrutto **for** possiamo scrivere dei cicli piú compatti. La sintassi di base è:

```
for(<init.>; <terminaz.>; <iteraz.>)
    {<corpo del ciclo>}
```

```
1 for(i = 0; i < MAX; i++)
2       {a = a + 1;}
```

While vs for

Dovremmo utilizzare il ciclo for soltanto quando il numero di iterazioni è noto a priori, ed utilizzare il ciclo while negli altri casi.

Costrutto for

Mediante il costrutto **for** possiamo scrivere dei cicli piú compatti. La sintassi di base è:

```
for(<init.>; <terminaz.>; <iteraz.>)
    {<corpo del ciclo>}
```

```
1  for(i = 0; i < MAX; i++)
2       {a = a + 1;}
```

While vs for

Dovremmo utilizzare il ciclo for soltanto quando il numero di iterazioni è noto a priori, ed utilizzare il ciclo while negli altri casi.

Costrutto for

Mediante il costrutto **for** possiamo scrivere dei cicli piú compatti. La sintassi di base è:

```
for(<init.>; <terminaz.>; <iteraz.>)
    {<corpo del ciclo>}
```

```
1  for(i = 0; i < MAX; i++)
2       {a = a + 1;}
```

While vs for

Dovremmo utilizzare il ciclo for soltanto quando il numero di iterazioni è noto a priori, ed utilizzare il ciclo while negli altri casi.

While vs for

```
1  int i;
2
3  //Ciclo while
4  i = 0;
5  while(i < MAX)
6  {
7       a += calcolo(i);
8       i++;
9  }
10
11 //Ciclo for
12 for(i = 0; i < MAX; i++)
13      {a += calcolo(i);}
```

Definizione di funzione

Per rendere piú leggibile il codice, e per evitare di **"replicarlo"**, possiamo definire delle **funzioni**.

La sintassi di base è:

```
<tiporisultato> <nomefunzione> (<listaparametri>){
    <corpo>
}
```

```c
int somma(int val1, int val2){
    // Variabile locale alla funzione
    int risultato;
    risultato = val1 + val2;
    return risultato;
}
```

Definizione di funzione

Per rendere piú leggibile il codice, e per evitare di **"replicarlo"**, possiamo definire delle **funzioni**.

La sintassi di base è:

```
<tiporisultato> <nomefunzione> (<listaparametri>){
    <corpo>
}
```

```
1 int somma(int val1, int val2){
2     // Variabile locale alla funzione
3     int risultato;
4     risultato = val1 + val2;
5     return risultato;
6 }
```

Definizione di funzione

Per rendere piú leggibile il codice, e per evitare di **“replicarlo”**, possiamo definire delle **funzioni**.

La sintassi di base è:

```
<tiporisultato> <nomefunzione> (<listaparametri>){
    <corpo>
}
```

```
1 int somma(int val1, int val2){
2     // Variabile locale alla funzione
3     int risultato;
4     risultato = val1 + val2;
5     return risultato;
6 }
```

Definizione di funzione

Per rendere piú leggibile il codice, e per evitare di **"replicarlo"**, possiamo definire delle **funzioni**.

La sintassi di base è:

```
<tiporisultato> <nomefunzione> (<listaparametri>){
    <corpo>
}
```

```
1  int somma(int val1, int val2){
2      // Variabile locale alla funzione
3      int risultato;
4      risultato = val1 + val2;
5      return risultato;
6  }
```

Esempio di funzione: i booleani

In C il tipo booleano (true, false) non è nativo del linguaggio.
Possiamo usare gli interi: 0 vale FALSE, tutti gli altri numeri valgono TRUE;
Scriviamo una funzione che verifichi se un numero è pari o meno.

```
1  while(i < MAX)
2      {if( pari(i) )
3          cont = cont + 1;}
4  // MAX e' una macro che indica il numero a cui fermarsi
5
6  int pari(int valore){
7      int ris = 0;
8      if((valore % 2) == 0)
9          ris = 1;
10     return ris
11 }
```

Esempio di funzione: i booleani

In C il tipo booleano (true, false) non è nativo del linguaggio.
Possiamo usare gli interi: 0 vale FALSE, tutti gli altri numeri valgono TRUE;
Scriviamo una funzione che verifichi se un numero è pari o meno.

```
1  while(i < MAX)
2      {if( pari(i) )
3          cont = cont + 1;}
4  // MAX e' una macro che indica il numero a cui fermarsi
5
6  int pari(int valore){
7      int ris = 0;
8      if((valore % 2) == 0)
9          ris = 1;
10     return ris
11 }
```

Esempio di funzione: i booleani

In C il tipo booleano (true, false) non è nativo del linguaggio. Possiamo usare gli interi: 0 vale FALSE, tutti gli altri numeri valgono TRUE;

Scriviamo una funzione che verifichi se un numero è pari o meno.

```
1  while(i < MAX)
2      {if( pari(i) )
3          cont = cont + 1;}
4  // MAX e' una macro che indica il numero a cui fermarsi
5
6  int pari(int valore){
7      int ris = 0;
8      if((valore % 2) == 0)
9          ris = 1;
10     return ris
11 }
```

Esempio di funzione: i booleani

In C il tipo booleano (true, false) non è nativo del linguaggio.
Possiamo usare gli interi: 0 vale FALSE, tutti gli altri numeri valgono TRUE;
Scriviamo una funzione che verifichi se un numero è pari o meno.

```
1  while(i < MAX)
2      {if( pari(i) )
3          cont = cont + 1;}
4  // MAX e' una macro che indica il numero a cui fermarsi
5
6  int pari(int valore){
7      int ris = 0;
8      if((valore % 2) == 0)
9          ris = 1;
10     return ris
11 }
```

Esempio di funzione: i booleani

In C il tipo booleano (true, false) non è nativo del linguaggio.
Possiamo usare gli interi: 0 vale FALSE, tutti gli altri numeri valgono TRUE;
Scriviamo una funzione che verifichi se un numero è pari o meno.

```
1  while(i < MAX)
2      {if( pari(i) )
3          cont = cont + 1;}
4  // MAX e' una macro che indica il numero a cui fermarsi
5
6  int pari(int valore){
7      int ris = 0;
8      if((valore % 2) == 0)
9          ris = 1;
10     return ris
11 }
```

Funzioni di I/O

Per relazionarci con l'esterno avremo bisogno delle funzioni di **input/output**.
Possiamo utilizzare quelle giá definite dalla libreria standard di C, contenute dentro stdio.h (Standard Input Output).

```c
#include <stdio.h>

int scanf(const char *format, ...);
int printf(const char *format, ...);
```

Le stringhe di Formato

Entrambi queste funzioni vogliono in input, oltre alle variabili da leggere/stampare, una stringa di formato

Funzioni di I/O

Per relazionarci con l'esterno avremo bisogno delle funzioni di **input/output**.

Possiamo utilizzare quelle giá definite dalla libreria standard di C, contenute dentro stdio.h (Standard Input Output).

```
1 #include <stdio.h>
2
3 int scanf(const char *format, ...);
4 int printf(const char *format, ...);
```

Le stringhe di Formato

Entrambi queste funzioni vogliono in input, oltre alle variabili da leggere/stampare, una stringa di formato

Funzioni di I/O

Per relazionarci con l'esterno avremo bisogno delle funzioni di **input/output**.
Possiamo utilizzare quelle giá definite dalla libreria standard di C, contenute dentro stdio.h (Standard Input Output).

```
1 #include <stdio.h>
2
3 int scanf(const char *format, ...);
4 int printf(const char *format, ...);
```

Le stringhe di Formato

Entrambi queste funzioni vogliono in input, oltre alle variabili da leggere/stampare, una stringa di formato

Funzioni di I/O

Per relazionarci con l'esterno avremo bisogno delle funzioni di **input/output**.
Possiamo utilizzare quelle giá definite dalla libreria standard di C, contenute dentro stdio.h (Standard Input Output).

```
1  #include <stdio.h>
2
3  int scanf(const char *format, ...);
4  int printf(const char *format, ...);
```

Le stringhe di Formato

Entrambi queste funzioni vogliono in input, oltre alle variabili da leggere/stampare, una stringa di formato

Funzioni di I/O

Per relazionarci con l'esterno avremo bisogno delle funzioni di **input/output**.
Possiamo utilizzare quelle giá definite dalla libreria standard di C, contenute dentro stdio.h (Standard Input Output).

```
1  #include <stdio.h>
2
3  int scanf(const char *format, ...);
4  int printf(const char *format, ...);
```

Le stringhe di Formato

Entrambi queste funzioni vogliono in input, oltre alle variabili da leggere/stampare, una stringa di formato

Esempio di uso di printf/scanf

```c
#include <stdio.h>

int main(){
    int a;
    scanf("%d", &a);
    /* Vedremo dopo perche' abbiamo messo & davanti alla
       variabile */

    printf("Il numero che hai inserito e': %d\n", a);
    printf("Il valore di %d modulo 2 e':\t %d\n", a, a%2);

    if(pari(a))
        {printf("Il numero e' pari\n");}
    else
        {printf("Il numero e' dispari\n");}

    return 0;
}
```

Segnaposti di printf/scanf

Nelle stringhe di formato possiamo utilizzare i seguenti **segnaposto**.

`%d`	Valore di tipo intero
`%c`	Valore di tipo carattere
`%s`	Valore di tipo stringa
`%o`	Valore ottale
`%x`	Valore esadecimale
`%p`	Valore di una locazione di memoria
`\n`	Newline
`\t`	Tabulazione

Segnaposti di printf/scanf

Nelle stringhe di formato possiamo utilizzare i seguenti **segnaposto**.

`%d`	Valore di tipo intero
`%c`	Valore di tipo carattere
`%s`	Valore di tipo stringa
`%o`	Valore ottale
`%x`	Valore esadecimale
`%p`	Valore di una locazione di memoria
`\n`	Newline
`\t`	Tabulazione

Segnaposti di printf/scanf

Nelle stringhe di formato possiamo utilizzare i seguenti **segnaposto**.

`%d`	Valore di tipo intero
`%c`	Valore di tipo carattere
`%s`	Valore di tipo stringa
`%o`	Valore ottale
`%x`	Valore esadecimale
`%p`	Valore di una locazione di memoria
`\n`	Newline
`\t`	Tabulazione

Strumenti di supporto allo sviluppo

Utilizzo del man

Uno strumento molto utile è il **man**.
Specialmente le sezioni 2 e 3. Dopo aver installato il pacchetto **manpages-dev** lo si puó invocare cosí:

```
bash:~$ man 3 printf

PRINTF(3)                 Linux Programmer's Manual                 PRINTF(3)

NAME
       printf,  fprintf,  sprintf,  snprintf,  vprintf,  vfprintf,  vsprintf,
       vsnprintf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
...
```

Proviamo a vedere cosa succede se invochiamo **man 3 stdio**

Utilizzo del man

Uno strumento molto utile è il **man**.

Specialmente le sezioni 2 e 3. Dopo aver installato il pacchetto **manpages-dev** lo si puó invocare cosí:

```
bash:~$ man 3 printf

PRINTF(3)                 Linux Programmer's Manual                 PRINTF(3)

NAME
       printf,  fprintf,  sprintf,  snprintf,  vprintf,  vfprintf,  vsprintf,
       vsnprintf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
...
```

Proviamo a vedere cosa succede se invochiamo **man 3 stdio**

Utilizzo del man

Uno strumento molto utile è il **man**.
Specialmente le sezioni 2 e 3. Dopo aver installato il pacchetto **manpages-dev** lo si puó invocare cosí:

```
bash:~$ man 3 printf

PRINTF(3)                 Linux Programmer's Manual                 PRINTF(3)

NAME
       printf,  fprintf,  sprintf,  snprintf,  vprintf,  vfprintf,  vsprintf,
       vsnprintf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
...
```

Proviamo a vedere cosa succede se invochiamo **man 3 stdio**

Utilizzo del man

Uno strumento molto utile è il **man**.
Specialmente le sezioni 2 e 3. Dopo aver installato il pacchetto **manpages-dev** lo si puó invocare cosí:

```
bash:~$ man 3 printf

PRINTF(3)                  Linux Programmer's Manual                 PRINTF(3)

NAME
       printf,  fprintf,  sprintf,  snprintf,  vprintf,  vfprintf,  vsprintf,
       vsnprintf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
...
```

Proviamo a vedere cosa succede se invochiamo **man 3 stdio**

Utilizzo del man

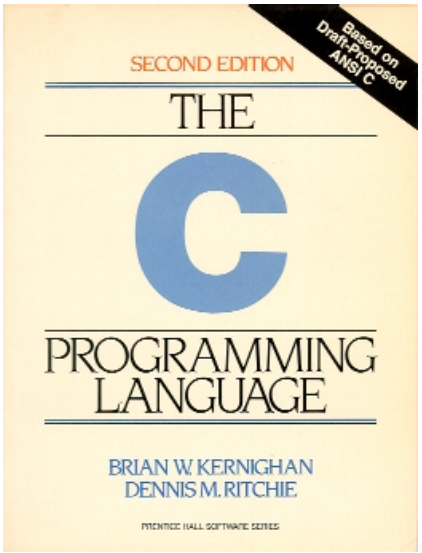
Uno strumento molto utile è il **man**.
Specialmente le sezioni 2 e 3. Dopo aver installato il pacchetto **manpages-dev** lo si puó invocare cosí:

```
bash:~$ man 3 printf

PRINTF(3)                  Linux Programmer's Manual                 PRINTF(3)

NAME
       printf,  fprintf,  sprintf,  snprintf,  vprintf,  vfprintf,  vsprintf,
       vsnprintf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
...
```

Proviamo a vedere cosa succede se invochiamo **man 3 stdio**

Il K&R

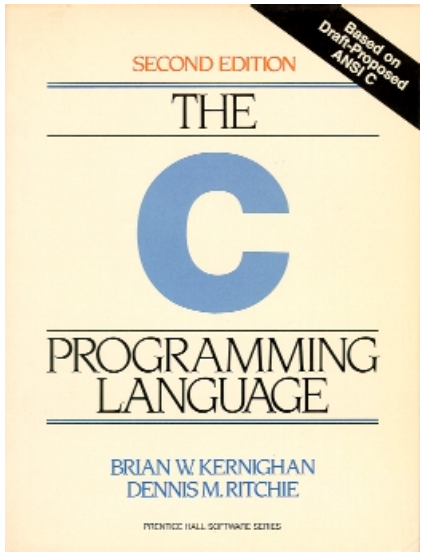
Un altro strumento di riferimento molto importante è **Il Linguaggio C** di Kernighan e Ritchie.



Manuale che dovrebbe essere sempre a portata di mano di ogni programmatore C.
Contiene tutta la definizione del linguaggio e la descrizione di tutta la libreria standard C.

Il K&R

Un altro strumento di riferimento molto importante è **Il Linguaggio C** di Kernighan e Ritchie.

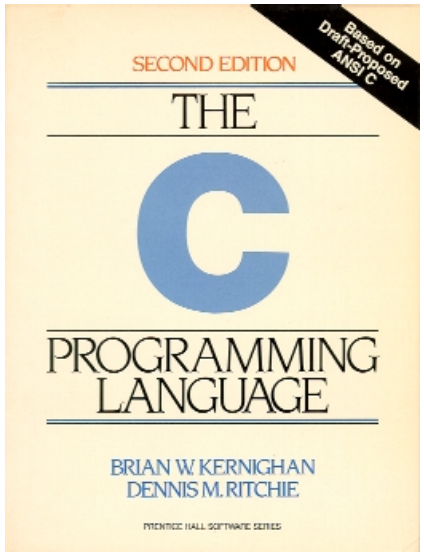


Manuale che dovrebbe essere sempre a portata di mano di ogni programmatore C.

Contiene tutta la definizione del linguaggio e la descrizione di tutta la libreria standard C.

Il K&R

Un altro strumento di riferimento molto importante è **Il Linguaggio C** di Kernighan e Ritchie.



Manuale che dovrebbe essere sempre a portata di mano di ogni programmatore C.
Contiene tutta la definizione del linguaggio e la descrizione di tutta la libreria standard C.

gdb e ddd

gdb è il debugger di C del progetto GNU. Ci permette di vedere i valori delle variabili a runtime.

Va invocato su un file compilato con l'opzione **-g**.

Va utilizzato da terminale. Altrimenti possiamo usare **ddd** come interfaccia grafica.

gdb e ddd

gdb è il debugger di C del progetto GNU. Ci permette di vedere i valori delle variabili a runtime.

Va invocato su un file compilato con l'opzione **-g**.

Va utilizzato da terminale. Altrimenti possiamo usare **ddd** come interfaccia grafica.

gdb e ddd

gdb è il debugger di C del progetto GNU. Ci permette di vedere i valori delle variabili a runtime.

Va invocato su un file compilato con l'opzione **-g**.

Va utilizzato da terminale. Altrimenti possiamo usare **ddd** come interfaccia grafica.

Strutture e tipi complessi

I vettori

In C è possibile definire delle **variabili strutturate**, un esempio semplice sono i **Vettori**

I vettori rappresentano una lista di elementi di dimensione **statica**, ció vuol dire che una volta definiti non possono essere ridimensionati.

La sintassi base per definire un vettore è:

```
<tipo> <nomevettore>[<dimensione>];
```

Possono essere utilizzati per memorizzare N elementi:

I vettori

In C è possibile definire delle **variabili strutturate**, un esempio semplice sono i **Vettori**

I vettori rappresentano una lista di elementi di dimensione **statica**, ció vuol dire che una volta definiti non possono essere ridimensionati.

La sintassi base per definire un vettore è:

```
<tipo> <nomevettore>[<dimensione>];
```

Possono essere utilizzati per memorizzare N elementi:

I vettori

In C è possibile definire delle **variabili strutturate**, un esempio semplice sono i **Vettori**

I vettori rappresentano una lista di elementi di dimensione **statica**, ció vuol dire che una volta definiti non possono essere ridimensionati.

La sintassi base per definire un vettore è:

```
<tipo> <nomevettore>[<dimensione>];
```

Possono essere utilizzati per memorizzare N elementi:

I vettori

In C è possibile definire delle **variabili strutturate**, un esempio semplice sono i **Vettori**

I vettori rappresentano una lista di elementi di dimensione **statica**, ció vuol dire che una volta definiti non possono essere ridimensionati.

La sintassi base per definire un vettore è:

```
<tipo> <nomevettore>[<dimensione>];
```

Possono essere utilizzati per memorizzare N elementi:

I vettori

In C è possibile definire delle **variabili strutturate**, un esempio semplice sono i **Vettori**

I vettori rappresentano una lista di elementi di dimensione **statica**, ció vuol dire che una volta definiti non possono essere ridimensionati.

La sintassi base per definire un vettore è:

```
<tipo> <nomevettore>[<dimensione>];
```

Possono essere utilizzati per memorizzare N elementi:

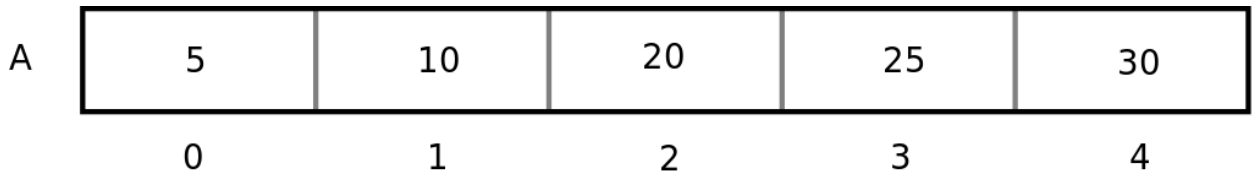
Esempio con un vettore

```
1  int vector[10];
2
3  vector[0] = 123;
4  /* ... */
5  vector[9] = 789;
6
7  for(i = 0; i < n; i++)
8    printf("%d", vector[i]);
```

Esempio con un vettore

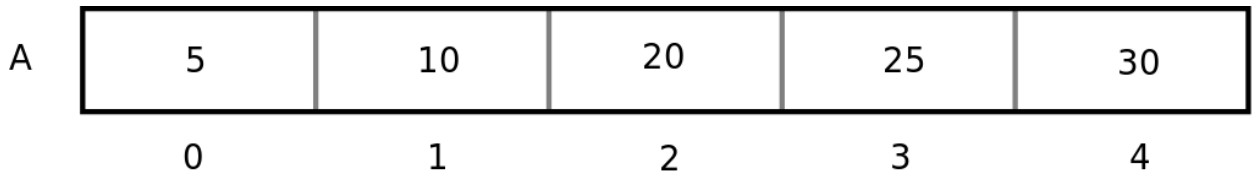
```
1  int vector[10];
2
3  vector[0] = 123;
4  /* ... */
5  vector[9] = 789;
6
7  for(i = 0; i < n; i++)
8    printf("%d", vector[i]);
```

Gli indici di un vettore



Un vettore di `n` elementi, ha gli indici che vanno da `0` a `n-1`

Gli indici di un vettore



Un vettore di `n` elementi, ha gli indici che vanno da `0` a `n-1`

Vediamo qualche **Esempio**...

Le stringhe

Se vogliamo memorizzare parole o frasi, dobbiamo utilizzare il tipo **Stringa**.
Il tipo stringa non è peró definito nativamente nel linguaggio C, dobbiamo utilizzare un **vettore di caratteri**.

```
1 char stringa[10];
2 stringa = "hello";
3 printf("%c",stringa[2]);
```

Il terminatore

Ogni stringa deve includere un carattere in piú che si chiama *terminatore* (e vale \0), altrimenti possiamo avere dei comportamenti inattesi *(Buffer overrun)*

Le stringhe

Se vogliamo memorizzare parole o frasi, dobbiamo utilizzare il tipo **Stringa**.

Il tipo stringa non è peró definito nativamente nel linguaggio C, dobbiamo utilizzare un **vettore di caratteri**.

```
1 char stringa[10];
2 stringa = "hello";
3 printf("%c",stringa[2]);
```

Il terminatore

Ogni stringa deve includere un carattere in piú che si chiama *terminatore* (e vale \0), altrimenti possiamo avere dei comportamenti inattesi *(Buffer overrun)*

Le stringhe

Se vogliamo memorizzare parole o frasi, dobbiamo utilizzare il tipo **Stringa**.
Il tipo stringa non è peró definito nativamente nel linguaggio C, dobbiamo utilizzare un **vettore di caratteri**.

```
1 char stringa[10];
2 stringa = "hello";
3 printf("%c",stringa[2]);
```

Il terminatore

Ogni stringa deve includere un carattere in piú che si chiama *terminatore* (e vale \0), altrimenti possiamo avere dei comportamenti inattesi *(Buffer overrun)*

Le stringhe

Se vogliamo memorizzare parole o frasi, dobbiamo utilizzare il tipo **Stringa**.
Il tipo stringa non è peró definito nativamente nel linguaggio C, dobbiamo utilizzare un **vettore di caratteri**.

```
1 char stringa[10];
2 stringa = "hello";
3 printf("%c",stringa[2]);
```

Il terminatore

Ogni stringa deve includere un carattere in piú che si chiama *terminatore* (e vale \0), altrimenti possiamo avere dei comportamenti inattesi *(Buffer overrun)*

Le stringhe

Se vogliamo memorizzare parole o frasi, dobbiamo utilizzare il tipo **Stringa**.
Il tipo stringa non è peró definito nativamente nel linguaggio C, dobbiamo utilizzare un **vettore di caratteri**.

```
1 char stringa[10];
2 stringa = "hello";
3 printf("%c",stringa[2]);
```

Il terminatore

Ogni stringa deve includere un carattere in piú che si chiama *terminatore* (e vale \0), altrimenti possiamo avere dei comportamenti inattesi *(Buffer overrun)*

I puntatori

In C possiamo avere accesso agli **indirizzi di memoria** dove sono memorizzate le variabili.

Possiamo avere puntatori che puntano ad ogni tipo di variabile.

Per definirli usiamo la sintassi:

```
<tipo> * <nomepunt>;
```

Per utilizzarli usiamo gli operatori `&` e `*`.

La cosa è piú semplice di quanto sembra...

Facciamo qualche esempio...

I puntatori

In C possiamo avere accesso agli **indirizzi di memoria** dove sono memorizzate le variabili.

Possiamo avere puntatori che puntano ad ogni tipo di variabile.

Per definirli usiamo la sintassi:

```
<tipo> * <nomepunt>;
```

Per utilizzarli usiamo gli operatori `&` e `*`.

La cosa è piú semplice di quanto sembra...

Facciamo qualche esempio...

I puntatori

In C possiamo avere accesso agli **indirizzi di memoria** dove sono memorizzate le variabili.

Possiamo avere puntatori che puntano ad ogni tipo di variabile.

Per definirli usiamo la sintassi:

```
<tipo> * <nomepunt>;
```

Per utilizzarli usiamo gli operatori `&` e `*`.

La cosa è piú semplice di quanto sembra...

Facciamo qualche esempio...

I puntatori

In C possiamo avere accesso agli **indirizzi di memoria** dove sono memorizzate le variabili.

Possiamo avere puntatori che puntano ad ogni tipo di variabile.

Per definirli usiamo la sintassi:

```
<tipo> * <nomepunt>;
```

Per utilizzarli usiamo gli operatori & e *.

La cosa è piú semplice di quanto sembra...

Facciamo qualche esempio...

I puntatori

In C possiamo avere accesso agli **indirizzi di memoria** dove sono memorizzate le variabili.

Possiamo avere puntatori che puntano ad ogni tipo di variabile.

Per definirli usiamo la sintassi:

```
<tipo> * <nomepunt>;
```

Per utilizzarli usiamo gli operatori `&` e `*`.

La cosa è piú semplice di quanto sembra...

Facciamo qualche esempio...

I puntatori

In C possiamo avere accesso agli **indirizzi di memoria** dove sono memorizzate le variabili.

Possiamo avere puntatori che puntano ad ogni tipo di variabile.

Per definirli usiamo la sintassi:

```
<tipo> * <nomepunt>;
```

Per utilizzarli usiamo gli operatori `&` e `*`.

La cosa è piú semplice di quanto sembra...

Facciamo qualche esempio...

I puntatori

In C possiamo avere accesso agli **indirizzi di memoria** dove sono memorizzate le variabili.

Possiamo avere puntatori che puntano ad ogni tipo di variabile.

Per definirli usiamo la sintassi:

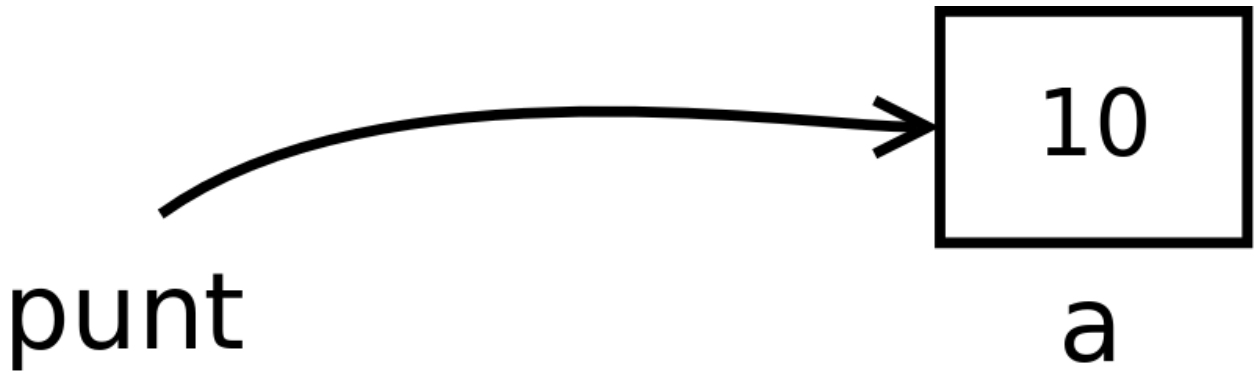
```
<tipo> * <nomepunt>;
```

Per utilizzarli usiamo gli operatori `&` e `*`.

La cosa è piú semplice di quanto sembra...

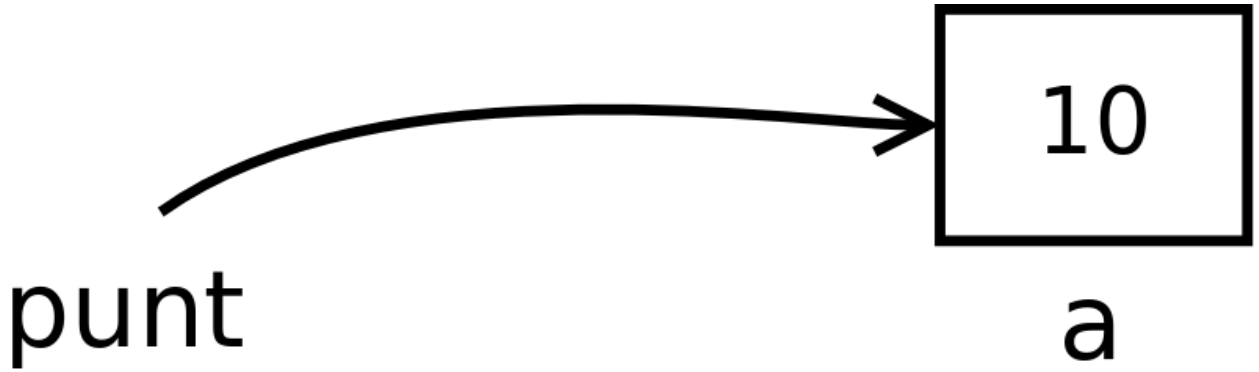
Facciamo qualche esempio...

Esempio sui puntatori (1)




```
1       int * punt;
2       int a = 3;
3
4       punt = &a;
5       printf("%p valore %d\n", punt, *punt);
6
7       // punt punta alla variabile a
8       // il valore di *punt e' il valore di a
```

Esempio sui puntatori (1)




```
1       int * punt;
2       int a = 3;
3
4       punt = &a;
5       printf("%p valore %d\n", punt, *punt);
6
7       // punt punta alla variabile a
8       // il valore di *punt e' il valore di a
```

Esempio sui puntatori (2)

Attenzione!

Anche i vettori e le stringhe sono dei puntatori!

```
1  int p[10];
2
3  int somma (int *a, int n){
4
5      int sum = 0, i;
6      for(i = 0; i < n; i++)
7          sum += *(a + i);
8
9          return sum
10     }
```

Esempio sui puntatori (2)

Attenzione!

Anche i vettori e le stringhe sono dei puntatori!

```
1  int p[10];
2
3  int somma (int *a, int n){
4
5      int sum = 0, i;
6      for(i = 0; i < n; i++)
7          sum += *(a + i);
8
9          return sum
10     }
```

Esempio sui puntatori (2)

Attenzione!

Anche i vettori e le stringhe sono dei puntatori!

```
1  int p[10];
2
3  int somma (int *a, int n){
4
5      int sum = 0, i;
6      for(i = 0; i < n; i++)
7          sum += *(a + i);
8
9          return sum
10     }
```

Passaggio dei parametri

Possiamo utilizzare i puntatori anche per il passaggio dei parametri per simulare un passaggio per **riferimento**.
In parole povere, possiamo utilizzare i puntatori per scrivere funzioni che **modificano i loro parametri**.

```
1 int a = 10;
2 void addhalf (int * number){
3     *number = *number + (*number / 2);
4 }
5 addhalf(&a);
6 //La funzione void non ritorna niente
```

Questo è il motivo per cui nella scanf si usa &nomevar

Passaggio dei parametri

Possiamo utilizzare i puntatori anche per il passaggio dei parametri per simulare un passaggio per **riferimento**.

In parole povere, possiamo utilizzare i puntatori per scrivere funzioni che **modificano i loro parametri**.

```
1 int a = 10;
2 void addhalf (int * number){
3     *number = *number + (*number / 2);
4 }
5 addhalf(&a);
6 //La funzione void non ritorna niente
```

Questo è il motivo per cui nella scanf si usa &nomevar

Passaggio dei parametri

Possiamo utilizzare i puntatori anche per il passaggio dei parametri per simulare un passaggio per **riferimento**.
In parole povere, possiamo utilizzare i puntatori per scrivere funzioni che **modificano i loro parametri**.

```
1 int a = 10;
2 void addhalf (int * number){
3     *number = *number + (*number / 2);
4 }
5 addhalf(&a);
6 //La funzione void non ritorna niente
```

Questo è il motivo per cui nella scanf si usa &nomevar

Passaggio dei parametri

Possiamo utilizzare i puntatori anche per il passaggio dei parametri per simulare un passaggio per **riferimento**.
In parole povere, possiamo utilizzare i puntatori per scrivere funzioni che **modificano i loro parametri**.

```
1 int a = 10;
2 void addhalf (int * number){
3     *number = *number + (*number / 2);
4 }
5 addhalf(&a);
6 //La funzione void non ritorna niente
```

Questo è il motivo per cui nella scanf si usa &nomevar

Passaggio dei parametri

Possiamo utilizzare i puntatori anche per il passaggio dei parametri per simulare un passaggio per **riferimento**.
In parole povere, possiamo utilizzare i puntatori per scrivere funzioni che **modificano i loro parametri**.

```
1 int a = 10;
2 void addhalf (int * number){
3     *number = *number + (*number / 2);
4 }
5 addhalf(&a);
6 //La funzione void non ritorna niente
```

Questo è il motivo per cui nella scanf si usa &nomevar

Cenno - Doppi puntatori

In C possiamo anche definire dei **puntatori di puntatori**.
Si usano solo in casi particolari, tipo per rappresentare le matrici di interi, i vettori di interi, oppure per usare funzioni che modificano i puntatori.

```
1       int a[10][10];
2
3       // a e' di tipo int**
4
5       *(*(a + 2) + 3) = 10;
6
7       // Equivale ad a[2][3] = 10
```

Cenno - Doppi puntatori

In C possiamo anche definire dei **puntatori di puntatori**.

Si usano solo in casi particolari, tipo per rappresentare le matrici di interi, i vettori di interi, oppure per usare funzioni che modificano i puntatori.

```
1       int a[10][10];
2
3       // a e' di tipo int**
4
5       *(*(a + 2) + 3) = 10;
6
7       // Equivale ad a[2][3] = 10
```

Cenno - Doppi puntatori

In C possiamo anche definire dei **puntatori di puntatori**.
Si usano solo in casi particolari, tipo per rappresentare le matrici di interi, i vettori di interi, oppure per usare funzioni che modificano i puntatori.

```
1       int a[10][10];
2
3       // a e' di tipo int**
4
5       *(*(a + 2) + 3) = 10;
6
7       // Equivale ad a[2][3] = 10
```

Cenno - Doppi puntatori

In C possiamo anche definire dei **puntatori di puntatori**.
Si usano solo in casi particolari, tipo per rappresentare le matrici di interi, i vettori di interi, oppure per usare funzioni che modificano i puntatori.

```
1       int a[10][10];
2
3       // a e' di tipo int**
4
5       *(*(a + 2) + 3) = 10;
6
7       // Equivale ad a[2][3] = 10
```

Strutture e nuovi tipi

Definiamo adesso dei nuovi tipi di dati, che ci aiutino a rappresentare meglio le informazioni che vogliamo elaborare.

Possiamo utilizzare il costruttore **struct** per creare dei record di informazioni relative ad uno stesso elemento.

Aggreghiamo insieme variabili di tipi differenti, ma logicamente correlate.

La sintassi per definire una struttura è

```
struct <nome> {
    <tipovar> <nomecampo>;
    ...
    };
```

Strutture e nuovi tipi

Definiamo adesso dei nuovi tipi di dati, che ci aiutino a rappresentare meglio le informazioni che vogliamo elaborare.

Possiamo utilizzare il costruttore **struct** per creare dei record di informazioni relative ad uno stesso elemento.

Aggreghiamo insieme variabili di tipi differenti, ma logicamente correlate.

La sintassi per definire una struttura è

```
struct <nome> {
    <tipovar> <nomecampo>;
    ...
    };
```

Strutture e nuovi tipi

Definiamo adesso dei nuovi tipi di dati, che ci aiutino a rappresentare meglio le informazioni che vogliamo elaborare.

Possiamo utilizzare il costruttore **struct** per creare dei record di informazioni relative ad uno stesso elemento.

Aggreghiamo insieme variabili di tipi differenti, ma logicamente correlate.

La sintassi per definire una struttura è

```
struct <nome> {
    <tipovar> <nomecampo>;
    ...
    };
```

Strutture e nuovi tipi

Definiamo adesso dei nuovi tipi di dati, che ci aiutino a rappresentare meglio le informazioni che vogliamo elaborare.

Possiamo utilizzare il costruttore **struct** per creare dei record di informazioni relative ad uno stesso elemento.

Aggreghiamo insieme variabili di tipi differenti, ma logicamente correlate.

La sintassi per definire una struttura è

```
struct <nome> {
    <tipovar> <nomecampo>;
    ...
    };
```

Strutture e nuovi tipi

Definiamo adesso dei nuovi tipi di dati, che ci aiutino a rappresentare meglio le informazioni che vogliamo elaborare.

Possiamo utilizzare il costruttore **struct** per creare dei record di informazioni relative ad uno stesso elemento.

Aggreghiamo insieme variabili di tipi differenti, ma logicamente correlate.

La sintassi per definire una struttura è

```
struct <nome> {
    <tipovar> <nomecampo>;
    ...
    };
```

Esempio di uso delle strutture

```
1  struct record {
2  int matricola;
3  char nome[30];
4  char codfisc[17];
5  };
6  int main(){
7      struct record mario;
8      mario.matricola = 123456;
9      mario.nome = "mario";
10     mario.codfisc = "MRSRSS88A10G702A";
11 }
```

Accediamo ai campi della struttura con l'operatore "."

Esempio di uso delle strutture

```
1  struct record {
2  int matricola;
3  char nome[30];
4  char codfisc[17];
5  };
6  int main(){
7       struct record mario;
8       mario.matricola = 123456;
9       mario.nome = "mario";
10      mario.codfisc = "MRSRSS88A10G702A";
11 }
```

Accediamo ai campi della struttura con l'operatore "."

Esempio di uso delle strutture

```
1  struct record {
2  int matricola;
3  char nome[30];
4  char codfisc[17];
5  };
6  int main(){
7       struct record mario;
8       mario.matricola = 123456;
9       mario.nome = "mario";
10      mario.codfisc = "MRSRSS88A10G702A";
11 }
```

Accediamo ai campi della struttura con l'operatore "."

Problema...

Fino ad ora, abbiamo definito gli array con una dimensione definita a priori...

E se volessimo far decidere la dimensione all'utente...?!?

Non possiamo fare `int a[n]` dopo aver letto n con una `scanf`...

Problema...

Fino ad ora, abbiamo definito gli array con una dimensione definita a priori...

E se volessimo far decidere la dimensione all'utente...?!?

Non possiamo fare `int a[n]` dopo aver letto n con una `scanf`...

Problema...

Fino ad ora, abbiamo definito gli array con una dimensione definita a priori...

E se volessimo far decidere la dimensione all'utente...?!?

Non possiamo fare `int a[n]` dopo aver letto n con una `scanf`...

Problema...

Fino ad ora, abbiamo definito gli array con una dimensione definita a priori...

E se volessimo far decidere la dimensione all'utente...?!?

Non possiamo fare `int a[n]` dopo aver letto n con una `scanf`...

Allocazione di memoria dinamica

Dobbiamo utilizzare l'**allocazione dinamica della memoria**.
Per farlo possiamo utilizzare le seguenti funzioni:

malloc Restituisce il puntatore ad un'area di memoria della dimensione desiderata.

calloc Restituisce il puntatore ad un'area di memoria e la inizializza a 0.

realloc Permette di riallocare un'area di memoria che era giá stata allocata prima.

In caso di fallimento ritornano il puntatore `NULL`.

Allocazione di memoria dinamica

Dobbiamo utilizzare l'**allocazione dinamica della memoria**.
Per farlo possiamo utilizzare le seguenti funzioni:

malloc Restituisce il puntatore ad un'area di memoria della dimensione desiderata.

calloc Restituisce il puntatore ad un'area di memoria e la inizializza a 0.

realloc Permette di riallocare un'area di memoria che era giá stata allocata prima.

In caso di fallimento ritornano il puntatore `NULL`.

Allocazione di memoria dinamica

Dobbiamo utilizzare l'**allocazione dinamica della memoria**.
Per farlo possiamo utilizzare le seguenti funzioni:

malloc Restituisce il puntatore ad un'area di memoria della dimensione desiderata.

calloc Restituisce il puntatore ad un'area di memoria e la inizializza a 0.

realloc Permette di riallocare un'area di memoria che era giá stata allocata prima.

In caso di fallimento ritornano il puntatore `NULL`.

Allocazione di memoria dinamica

Dobbiamo utilizzare l'**allocazione dinamica della memoria**.
Per farlo possiamo utilizzare le seguenti funzioni:

malloc Restituisce il puntatore ad un'area di memoria della dimensione desiderata.

calloc Restituisce il puntatore ad un'area di memoria e la inizializza a 0.

realloc Permette di riallocare un'area di memoria che era giá stata allocata prima.

In caso di fallimento ritornano il puntatore `NULL`.

Allocazione di memoria dinamica

Dobbiamo utilizzare l'**allocazione dinamica della memoria**.
Per farlo possiamo utilizzare le seguenti funzioni:

malloc Restituisce il puntatore ad un'area di memoria della dimensione desiderata.

calloc Restituisce il puntatore ad un'area di memoria e la inizializza a 0.

realloc Permette di riallocare un'area di memoria che era giá stata allocata prima.

In caso di fallimento ritornano il puntatore `NULL`.

Allocazione di memoria dinamica

Dobbiamo utilizzare l'**allocazione dinamica della memoria**.
Per farlo possiamo utilizzare le seguenti funzioni:

malloc Restituisce il puntatore ad un'area di memoria della dimensione desiderata.

calloc Restituisce il puntatore ad un'area di memoria e la inizializza a 0.

realloc Permette di riallocare un'area di memoria che era giá stata allocata prima.

In caso di fallimento ritornano il puntatore `NULL`.

Allocazione di memoria dinamica

Dobbiamo utilizzare l'**allocazione dinamica della memoria**.
Per farlo possiamo utilizzare le seguenti funzioni:

malloc Restituisce il puntatore ad un'area di memoria della dimensione desiderata.

calloc Restituisce il puntatore ad un'area di memoria e la inizializza a 0.

realloc Permette di riallocare un'area di memoria che era giá stata allocata prima.

In caso di fallimento ritornano il puntatore `NULL`.

Uso della malloc

La **malloc** è una funzione contenuta nell'header `stdlib.h`.

Il suo prototipo è:

```
void * malloc(size_t size);
```

La funzione restituisce un puntatore generico, e vuole un parametro `size` che rappresenta la dimensione.

Possiamo usare **sizeof** per conoscere la dimensione dei vari tipi di dato.

Uso della malloc

La **malloc** è una funzione contenuta nell'header `stdlib.h`.

Il suo prototipo è:

```
void * malloc(size_t size);
```

La funzione restituisce un puntatore generico, e vuole un parametro `size` che rappresenta la dimensione.

Possiamo usare **sizeof** per conoscere la dimensione dei vari tipi di dato.

Uso della malloc

La **malloc** è una funzione contenuta nell'header `stdlib.h`.

Il suo prototipo è:

```
void * malloc(size_t size);
```

La funzione restituisce un puntatore generico, e vuole un parametro `size` che rappresenta la dimensione.

Possiamo usare **sizeof** per conoscere la dimensione dei vari tipi di dato.

Uso della malloc

La **malloc** è una funzione contenuta nell'header `stdlib.h`.

Il suo prototipo è:

```
void * malloc(size_t size);
```

La funzione restituisce un puntatore generico, e vuole un parametro `size` che rappresenta la dimensione.

Possiamo usare **sizeof** per conoscere la dimensione dei vari tipi di dato.

Uso della malloc

La **malloc** è una funzione contenuta nell'header `stdlib.h`.

Il suo prototipo è:

```
void * malloc(size_t size);
```

La funzione restituisce un puntatore generico, e vuole un parametro `size` che rappresenta la dimensione.

Possiamo usare **sizeof** per conoscere la dimensione dei vari tipi di dato.

Esempio di uso della malloc

```
1  int * voti;
2  struct record * studenti;
3  // Dichiaro 2 array
4
5  scanf("%d", &n);
6  // Leggo in input il numero degli studenti
7
8  voti = malloc(n * sizeof(int));
9  studenti = malloc(n * sizeof(struct sutdenti));
10 // Alloco dinamicamente i 2 array
11
12 voti[0] = 25;
13 studenti[0].nome = "Tizio";
```

Definizione di nuovi tipi

Possiamo usare **typedef** per rinominare i tipi di dato.

Ci puó tornare molto utile con i puntatori:

```
1 typedef int * puntatore;
2
3 int * a;
4 puntatore b;
5 //Sono 2 variabili dello stesso tipo
```

Definizione di nuovi tipi

Possiamo usare **typedef** per rinominare i tipi di dato.

Ci puó tornare molto utile con i puntatori:

```
1 typedef int * puntatore;
2
3 int * a;
4 puntatore b;
5 //Sono 2 variabili dello stesso tipo
```

Definizione di nuovi tipi

Possiamo usare **typedef** per rinominare i tipi di dato.

Ci puó tornare molto utile con i puntatori:

```
1 typedef int * puntatore;
2
3 int * a;
4 puntatore b;
5 //Sono 2 variabili dello stesso tipo
```

Definizione di nuovi tipi

Possiamo usare **typedef** per rinominare i tipi di dato.

Ci puó tornare molto utile con i puntatori:

```
1 typedef int * puntatore;
2
3 int * a;
4 puntatore b;
5 //Sono 2 variabili dello stesso tipo
```

Le liste

Mediante le liste possiamo creare delle strutture che siano dinamiche; possiamo liberarci dal vincolo della dimensione imposto dagli array, e creare strutture che **aumentano e diminuiscono a piacere.**

Basta aggiungere un puntatore ad una struttura dello stesso tipo, che punti al successivo:

```
1 struct nodo {
2     int valore;
3     struct nodo * next;
4 };
```

Le liste

Mediante le liste possiamo creare delle strutture che siano dinamiche; possiamo liberarci dal vincolo della dimensione imposto dagli array, e creare strutture che **aumentano e diminuiscono a piacere.**

Basta aggiungere un puntatore ad una struttura dello stesso tipo, che punti al successivo:

```
1 struct nodo {
2     int valore;
3     struct nodo * next;
4 };
```

Le liste

Mediante le liste possiamo creare delle strutture che siano dinamiche; possiamo liberarci dal vincolo della dimensione imposto dagli array, e creare strutture che **aumentano e diminuiscono a piacere.**

Basta aggiungere un puntatore ad una struttura dello stesso tipo, che punti al successivo:

```
1 struct nodo {
2     int valore;
3     struct nodo * next;
4 };
```

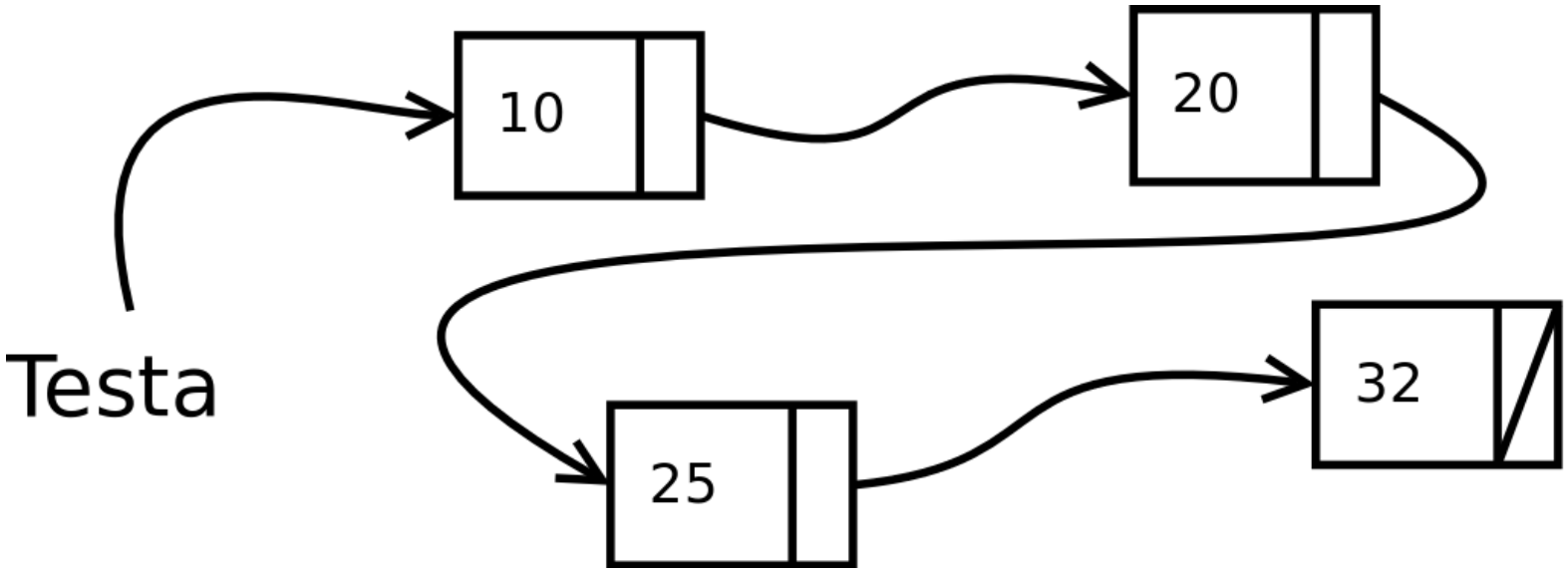
Le liste

Mediante le liste possiamo creare delle strutture che siano dinamiche; possiamo liberarci dal vincolo della dimensione imposto dagli array, e creare strutture che **aumentano e diminuiscono a piacere.**

Basta aggiungere un puntatore ad una struttura dello stesso tipo, che punti al successivo:

```
1 struct nodo {
2     int valore;
3     struct nodo * next;
4 };
```

Rappresentazione grafica di una lista



Esempio di uso delle liste

```
1  struct nodo {
2  int valore;
3  struct nodo * next;
4  };
5
6  typedef struct nodo elemento;
7  typedef elemento * lista;
8
9  int main(){
10
11      lista testa = NULL;
12      testa = malloc(sizeof(elemento));
13
14      testa->valore = 0;
15      testa->next = NULL;
16
17 }
```

Note sulle liste

Usiamo l'operatore "->" per accedere ai campi di una struttura di cui abbiamo il puntatore.

Prima usavamo l'operatore "." dato che potevamo operare direttamente sulla struttura.

Le seguenti espressioni sono equivalenti:

```
lista -> elem;
(*lista).elem;
```

Note sulle liste

Usiamo l'operatore "->" per accedere ai campi di una struttura di cui abbiamo il puntatore.

Prima usavamo l'operatore "." dato che potevamo operare direttamente sulla struttura.

Le seguenti espressioni sono equivalenti:

```
lista -> elem;
(*lista).elem;
```

Note sulle liste

Usiamo l'operatore "->" per accedere ai campi di una struttura di cui abbiamo il puntatore.

Prima usavamo l'operatore "." dato che potevamo operare direttamente sulla struttura.

Le seguenti espressioni sono equivalenti:

```
lista -> elem;
(*lista).elem;
```

Note sulle liste

Usiamo l'operatore "->" per accedere ai campi di una struttura di cui abbiamo il puntatore.

Prima usavamo l'operatore "." dato che potevamo operare direttamente sulla struttura.

Le seguenti espressioni sono equivalenti:

```
lista -> elem;
(*lista).elem;
```

Note sulle liste

Inoltre usiamo il valore **NULL**, per indicare l'elemento nullo (usato per esempio per la fine della lista).

Se vogliamo effettuare una scansione di una lista, utilizzeremo un puntatore di appoggio, e lo faremo avanzare elemento per elemento.

Ci dobbiamo fermare appena il puntatore di appoggio diventa **NULL**.

Attenzione!

Accedere ai campi di una puntatore NULL, provoca un errore di **Segmentation Fault**

Note sulle liste

Inoltre usiamo il valore **NULL**, per indicare l'elemento nullo (usato per esempio per la fine della lista).

Se vogliamo effettuare una scansione di una lista, utilizzeremo un puntatore di appoggio, e lo faremo avanzare elemento per elemento.

Ci dobbiamo fermare appena il puntatore di appoggio diventa **NULL**.

Attenzione!

Accedere ai campi di una puntatore NULL, provoca un errore di **Segmentation Fault**

Note sulle liste

Inoltre usiamo il valore **NULL**, per indicare l'elemento nullo (usato per esempio per la fine della lista).

Se vogliamo effettuare una scansione di una lista, utilizzeremo un puntatore di appoggio, e lo faremo avanzare elemento per elemento.

Ci dobbiamo fermare appena il puntatore di appoggio diventa **NULL**.

Attenzione!

Accedere ai campi di una puntatore NULL, provoca un errore di **Segmentation Fault**

Note sulle liste

Inoltre usiamo il valore **NULL**, per indicare l'elemento nullo (usato per esempio per la fine della lista).

Se vogliamo effettuare una scansione di una lista, utilizzeremo un puntatore di appoggio, e lo faremo avanzare elemento per elemento.

Ci dobbiamo fermare appena il puntatore di appoggio diventa **NULL**.

Attenzione!

Accedere ai campi di una puntatore NULL, provoca un errore di **Segmentation Fault**

Note sulle liste

Inoltre usiamo il valore **NULL**, per indicare l'elemento nullo (usato per esempio per la fine della lista).

Se vogliamo effettuare una scansione di una lista, utilizzeremo un puntatore di appoggio, e lo faremo avanzare elemento per elemento.

Ci dobbiamo fermare appena il puntatore di appoggio diventa **NULL**.

Attenzione!

Accedere ai campi di una puntatore NULL, provoca un errore di **Segmentation Fault**

Free

Per liberare lo spazio allocato in precedenza con una malloc si usa la funzione `free`.

```
void free(void *ptr);
```

La free vuole in input il puntatore dell'area di memoria da liberare, e non restituisce niente.

Il garbage

È sempre buona norma deallocare le strutture che non vengono piú usate, e ridurre cosí il *garbage*. Possiamo usare delle utility come **valgrind** o **mtrace** che tengono traccia della memoria allocata per vedere cosa ci siamo dimenticati.

Free

Per liberare lo spazio allocato in precedenza con una malloc si usa la funzione `free`.

```
void free(void *ptr);
```

La free vuole in input il puntatore dell'area di memoria da liberare, e non restituisce niente.

Il garbage

È sempre buona norma deallocare le strutture che non vengono piú usate, e ridurre cosí il *garbage*. Possiamo usare delle utility come **valgrind** o **mtrace** che tengono traccia della memoria allocata per vedere cosa ci siamo dimenticati.

Free

Per liberare lo spazio allocato in precedenza con una malloc si usa la funzione `free`.

```
void free(void *ptr);
```

La free vuole in input il puntatore dell'area di memoria da liberare, e non restituisce niente.

Il garbage

È sempre buona norma deallocare le strutture che non vengono piú usate, e ridurre cosí il *garbage*. Possiamo usare delle utility come **valgrind** o **mtrace** che tengono traccia della memoria allocata per vedere cosa ci siamo dimenticati.

Free

Per liberare lo spazio allocato in precedenza con una malloc si usa la funzione `free`.

```
void free(void *ptr);
```

La free vuole in input il puntatore dell'area di memoria da liberare, e non restituisce niente.

Il garbage

È sempre buona norma deallocare le strutture che non vengono piú usate, e ridurre cosí il *garbage*. Possiamo usare delle utility come **valgrind** o **mtrace** che tengono traccia della memoria allocata per vedere cosa ci siamo dimenticati.

Free

Per liberare lo spazio allocato in precedenza con una malloc si usa la funzione `free`.

```
void free(void *ptr);
```

La free vuole in input il puntatore dell'area di memoria da liberare, e non restituisce niente.

Il garbage

È sempre buona norma deallocare le strutture che non vengono piú usate, e ridurre cosí il *garbage*. Possiamo usare delle utility come **valgrind** o **mtrace** che tengono traccia della memoria allocata per vedere cosa ci siamo dimenticati.

Domande...?

Slides realizzate da:

Nicola Corti - corti.nico [at] gmail [dot] com

Michael Sanelli - michael [at] sanelli [dot] org

Slides realizzate con LaTeX Beamer, ed utilizzando interamente software libero.

La seguente presentazione è rilasciata sotto licenza **Creative Commons - Attributions, Non Commercial, Share-alike**.