DIPARTIMENTO
DI INFORMATICA
SISTEMISTICA
E COMUNICAZIONE

Linguaggi di Programmazione 2022-2023
Introduzione al C e C++ (2)

Marco Antoniotti
Gabriella Pasi
Fabio Sartori

Introduzione al C (2ª parte)

- Ulteriori elementi di C (e C++)
  - Consigli per lo sviluppo di programmi
    - Il compilatore, il pre-processore ed il "linker"
    - Come organizzare il codice e librerie (header files)
  - La memoria dinamica (heap)

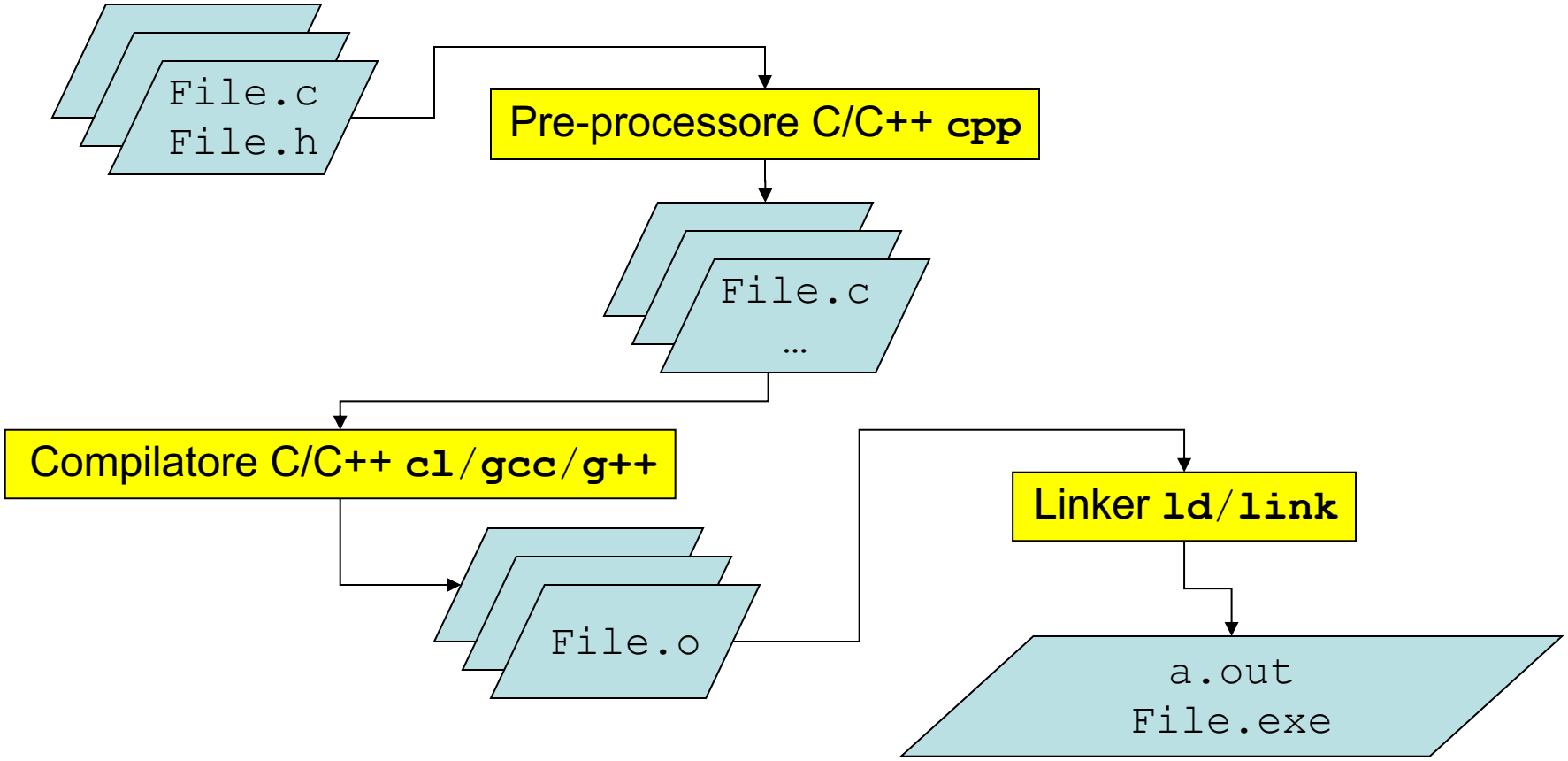
La compilazione dei programmi C/C++

- Assumiamo una "*command line interface*"
- Per compilare il programma, si invoca il compilatore

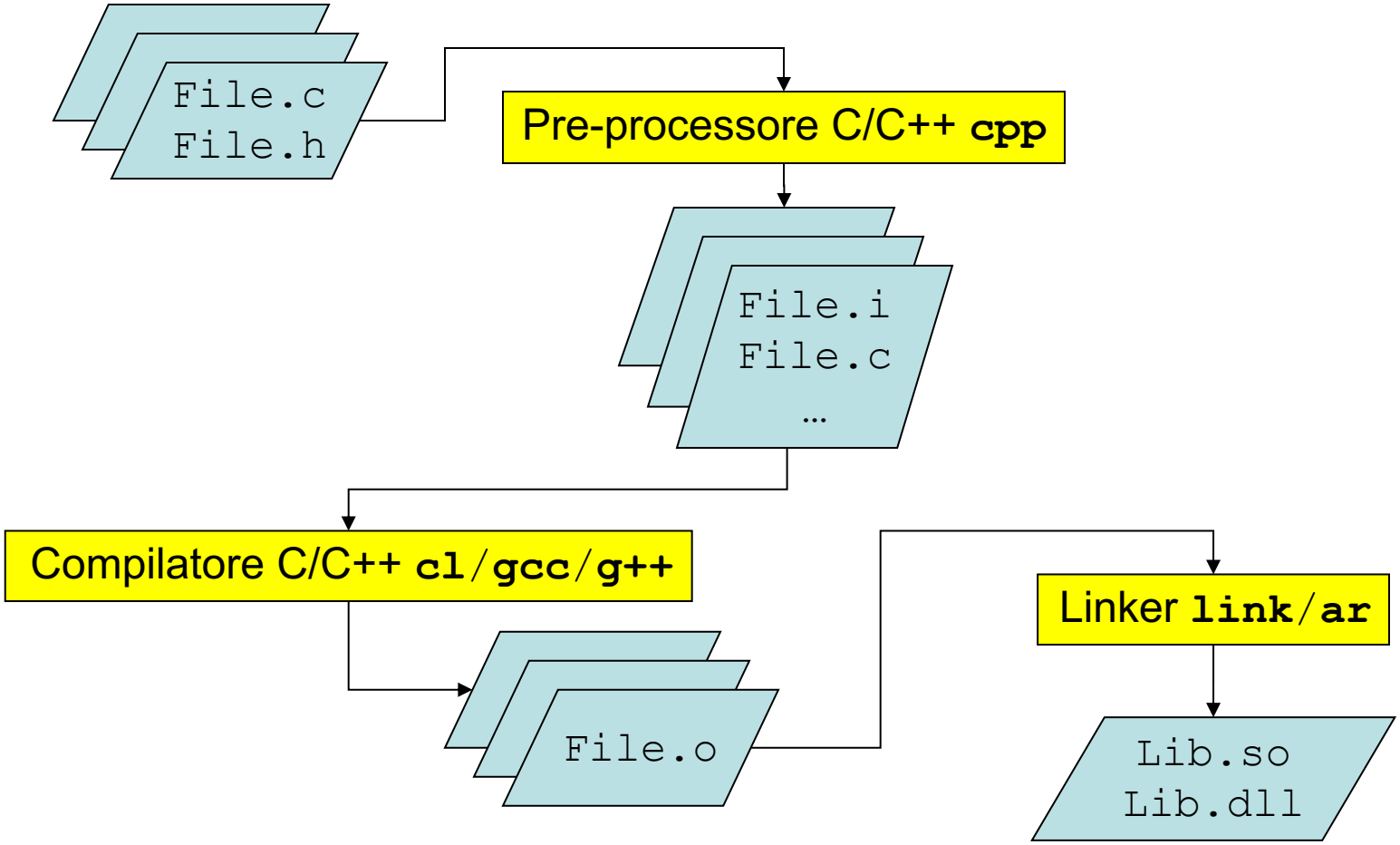
```
prompt$ gcc hello.c
```

- Questo nei casi semplici, in cui abbiamo solo un file, ma normalmente ciò non accade.
- Il codice sorgente di applicazioni e librerie è distribuito su un insieme di files e directories
  - La loro organizzazione è fondamentale al fine di costruire sistemi estensibili e mantenibili
- I programmi C/C++ si basano sulla distinzione tra
  - Header files (con estensione **`.h`** o **`.hpp`**)
  - File di implementazione (con varie estensioni: **`.c`**, **`.cc`**, **`.C`**, **`.cpp`**)
- Questa distinzione fa leva sul pre-processore C/C++

La compilazione dei programmi C/C++: produzione di eseguibili



La compilazione dei programmi C/C++: produzione di librerie



La compilazione dei files C/C++

- Chiamare il compilatore dalla linea di comando fa sì che venga chiamato anche il preprocessore
  - Esistono – di solito – delle opzioni che permettono di richiamare solo il compilatore o solo il preprocessore
    - Ad esempio, `-E` con `gcc`

- Chiamare il compilatore dalla linea di comando fa sì che venga chiamato anche il linker
  - Anche in questo caso è possibile chiamare selettivamente questi programmi
    - Ad esempio, `-c` (produci solo file “oggetto”) su molte piattaforme C/C++

cpp: il preprocessore C/C++

- Il preprocessore C/C++ è un programma che trasforma testo
- Non è necessario che il testo sia un programma C/C++

- Il preprocessore C/C++ opera sulla base di direttive
- Le più utili sono di tre tipi
  - Inclusione di testo
  - Definizione di “macro”
  - Condizionali
- Tutte queste direttive funzionano assieme per permetterci di costruire programmi modulari

cpp: il preprocessore C/C++

- Direttiva di inclusione

```
#include ”file.h”
```

oppure

```
#include <file.h>
```

include tutto il file nel sorgente

cpp: il preprocessore C/C++

- Direttiva di definizione

```
#define PI 3.14L
```

oppure

```
#define max(x, y) ((x) < (y) ? (y) : (x))
```

- Tutte le istanze della stringa `PI` o di stringhe `max(a, b * 42)` vengono sostituite
  - `PI` **==>** `3.14L`
  - `max(a, b*42)` **==>** `((a) < (b*42)) ? (b*42) : (a))`

cpp: il preprocessore C/C++

- Condizionali

```
#ifdef PI
     …
#else
     …
#endif
```

oppure

```
#ifndef PI
     …
#else
     …
#endif
```

- Il testo compreso tra `#ifdef` e `#endif` viene incluso o meno se la macro è definita (la parte `#else` è opzionale)

cpp: il preprocessore C/C++

- Usare il preprocessore era l'unico modo di introdurre "costanti" simboliche in C
- Buona parte degli "header" files di sistema (e no) contengono parecchie definizioni di costanti che vengono processate dal preprocessor C/C++
- **Esempio**: di solito, su una piattaforma simil-UNIX, il file `limits.h` viene incluso con la direttiva

```
#include <limits.h>
```

e contiene, tra l'altro, le seguenti definizioni (su Mac OS X)

```
#define CHAR_MAX    127
#define CHAR_MIN    (-128)
#define UINT_MAX    0xffffffff
#define INT_MAX     2147483647
#define INT_MIN     (-2147483647-1)
```

Compilazione separata e “header” files

- Ogni programma di una certa dimensione dovrebbe essere modularizzato in maniera appropriate
- La modularizzazione di un programma corrisponde alloperazione di compilazione separata
- Un compilatore C/C++ agisce (di solito) su un solo file; il risultato è un file “oggetto” contente dei riferimenti irrisolti a codice non direttamente disponibile
- Il linker ha il compito di risolvere questi riferimenti e, nel caso, segnalare degli errori qualora qualche riferimento rimanga irrisolto

Compilazione separata

- Consideriamo il seguente file **`use-foo.c`**

```
#include <stdio.h>
int foo(int); /* Una dichiarazione di funzione. */

int main() {
     printf("foo(42) == %d\n", foo(42));  /* Il suo uso. */
}
```

- Proviamo a compilarla (su Mac OS X)

```
prompt$ gcc use-foo.c
ld: Undefined symbols:
_foo
```

- Dato che `foo` non è definita, ma solo dichiarata, il linker **`ld`** (che sta cercando di creare un eseguibile, ma trova un simbolo non definito) segnala un errore
  - Il prefisso '_' denota solitamente i simboli che vengono manipolati dal linker

Compilazione separata

- Definiamo foo nel file **`foo.c`**

```
int foo(int x) {        /* La definizione di `foo'. */
      return x == 42 ? 1 : 0;
}
```

- Proviamo a compilarla (su Mac OS X)

```
prompt$ gcc foo.c
ld: Undefined symbols:
_main
```

- Dato che `foo.c` non contiene un programma completo – manca `main` – il linker **`ld`** (che sta cercando di creare un eseguibile, ma trova un simbolo non definito) segnala un errore

Compilazione separata

- Come possiamo evitare questi errori?
- Si usa l'opzione `-c`

```
prompt$ gcc -c foo.c
prompt$ gcc -c use-foo.c
```

(ordine ininfluente)

- Il risultato è la costruzione di due files, `use-foo.o` e `foo.o`
  - Simile alla creazione di files **`.class`** in Java
  - Ciò non è sufficiente per creare un eseguibile (che cosa possiamo eseguire?)

- La creazione di un eseguibile (se confrontata con Java) richiede un passo ulteriore: i files `use-foo.o` e `foo.o` devono essere linkati assieme

```
prompt$ gcc use-foo.o foo.o
prompt$ a.out
foo(42) == 1
```

  - Nota bene: il compilatore ha richiamato automaticamente il linker

Compilazione separata e “header” files

- Regole per definizioni e dichiarazioni distribuite su files multipli (e no)
  - Tutte le dichiarazioni di variabili e funzioni devono essere consistenti per tipo
  - Ogni oggetto può essere definito una volta sola
    - Il linker segnala un errore in caso contrario

Compilazione separata e “header” files

- Nell’esempio precedente abbiamo due files:
  - **`use-foo.c`** che “usa” una funzione definita (implementata) altrove
  - **`foo.c`** che contiene l’implementazione della funzione suddetta

- Un modo di separare l’interfaccia di un “modulo” (**`foo`** in questo caso) dalla sua implementazione, evitando al tempo stesso il pericolo di ridefinizioni incontrollate consiste nell’usare attentamente gli *header files*

- Riscriviamo il file **`use-foo.c`** nel seguente modo

```
#include <stdio.h>
#include "foo.h"        /* Contenente la dichiarazione della
                           funzione `foo'. */

int main() {
      printf("foo(42) == %d\n", foo(42));  /* Il suo uso. */
}
```

Compilazione separata e “header” files

- E riscriviamo il file **`foo.h`** nel seguente modo

```
extern int foo(int);   /* La dichiarazione della funzione `foo'. */
```

- Il file **`foo.h`** contiene l’interfaccia del modulo **`foo`**
- La compilazione di **`use-foo.c`** include (tramite il preprocessore) il file **`foo.h`** e quindi rende disponibile la dichiarazione
- Il file **`foo.h`** rimane a disposizione per essere incluso in altri files
- Per modificare l’interfaccia del modulo **`foo`** dobbiamo modificare un solo file

Compilazione separata e “header” files

- Il caso del modulo `foo` è fin troppo semplice
- Consideriamo un caso più complicato con due files che vogliono “usare” dei numeri complessi
  - **`complex-use-1.c`**
  - **`complex-use-2.c`**

```
/* complex-use-1.c */
struct complex {float im; float re;}
struct complex c_mult(struct complex*, struct complex*);
```

```
/* complex-use-2.c */
struct complex {float angle; float magnitude};
struct complex c_mult(struct complex* c1, struct complex* c2) {
   …
}
```

- In questi due casi abbiamo un conflitto nel contenuto delle due strutture (desumibile dai nomi dei campi)
- La concentrazione di queste definizioni in un solo file **`complex.h`** evidenzierebbe questo problema specifico

Compilazione separata e "header" files

- Il mantenimento della consistenza tra l'interfaccia dichiarata in un header file ed il contenuto del file di implementazione è un altro problema da affrontare

- Consideriamo il caso di **`foo.h`** e **`foo.c`**

```
/* foo.h */
extern int foo(int);
```

```
/* foo.c */
int foo(int x) {        /* La definizione di `foo'. */
    return x == 42 ? 1 : 0;
}
```

- Cosa succede se modifichiamo **`foo.c`**?

```
 /* foo.c modificato */
int foo(char x) {       /* La definizione di `foo'. */
    return x == 'x' ? 42 : 0;
}
```

Compilazione separata e “header” files

- Dipende dal grado di sofisticazione del compilatore
- Siccome i due files `foo.c` e `use-foo.c` sono completamente separati è difficile che un compilatore od un linker siano in grado di controllare queste interdipendenze
- Quindi abbiamo bisogno di imporre all’implementazione di un modulo il rispetto della sua interfaccia pubblica
- La cosa più semplice da fare è di includere l’header file nel file di implementazione stesso

```
/* foo.c modificato */

#include ”foo.h”

int foo(char x) {       /* La definizione di `foo’. */
      return x == ‘x’ ? 42 : 0;
}
```

Compilazione separata e “header” files: inclusioni multiple

- Se vogliamo includere **`foo.h`** in un header file **`baz.h`** per poi includere sia **`foo.h`** che **`baz.h`** in un file **`use-foo-baz.c`** dobbiamo mettere in conto la possibilità di definizioni multiple: strutture, tipi (classi in C++)

```
/* baz.h */

#include ”foo.h”

struct zot {char* zut; double qd[42]; };
extern void baz(int z, struct zot*);
```

```
/* use-foo-baz.c */

#include ”foo.h”
#include ”baz.h”

/* … codice che usa sia foo che baz … */
```

Compilazione separata e “header” files: inclusioni multiple

- La soluzioni in questo caso è di utilizzare le direttive condizionali del preprocessore

```
/* foo.h */

#ifndef _FOO_H
#define _FOO_H

extern int foo(int);

#endif
```

```
/* baz.h */

#ifndef _BAZ_H
#define _BAZ_H

#include ”foo.h”

struct zot {char* zut; double qd[42]; };
extern void baz(int z, struct zot*);

#endif
```

Compilazione separata e “header” files: inclusioni multiple

```
/* use-foo-baz.c */

#include ”foo.h”
#include ”baz.h”    /* foo.h non viene re-incluso! */

/* … codice che usa sia foo che baz … */
```

Come organizzare le librerie in C

- Che cos'è una "libreria"?
  - È un file in un particolare formato che può essere manipolato dal linker (sia staticamente che dinamicamente)
  - Come già accennato, una libreria (dinamica) ha l'estensione **`.so`** in Linux/Unix (**`.dylib`** in Mac OS X) e **`.dll`** in Windows

- Una libreria è essenzialmente una collezione di file oggetto con un indice associato che permette al linker (od al programma in esecuzione) di andare a caricare il codice corrispondente ad un dato "entry point" (ovvero ad una funzione)

Esempio: UNION-FIND come libreria

- Consideriamo una libreria chiamata QUICK-UNION-PCP (che implementa l'algoritmo UNION-FIND)
- Potremmo mettere tutto il codice in un solo file con la funzione `main` (o altre) che usa le funzioni **`find`** e **`unite`**
- Invece, per prima cosa scorporiamo l'interfaccia della libreria QUICK-UNION-PCP

```
#ifndef _QUPCP_H
#define _QUPCP_H

extern const int N; /* Inizializzazione non desiderata! */
extern bool find(int, int);
extern void unite(int, int);
extern void init_quick_find_pcp();

#endif
```

e la mettiamo in un file **`QUPCP.h`**

UNION-FIND come libreria

- Questa definizione non è ancora ottimale, ma dà un'idea della struttura della libreria
- Note
  - Il parametro `N` non è visibile all'esterno dell'implementazione
  - L'unico elemento non parametrico è il tipo (`int`) della rappresentazione usata per codificare gli elementi

- A questo punto potremmo cambiare completamente l'implementazione nel file **`QUPCP.cc`** (il file contenente la definizione di funzioni, costanti e parametri)

Potenziali conflitti sui “nomi”

- La libreria QUPCP “esporta” una constante chiamata `N`
- La scelta di questo nome non è particolarmente felice, dato che il nome `N` potrebbe essere specificato da molte altre librerie, nonché dal programma principale
- In C è assolutamente necessario essere molto disciplinati nella scelta dei nomi
  - In C++ la cosa è molto facilitata dalla presenza del concetto di “**namespace**”
- La soluzione più semplice consiste nel prefissare ogni nome della libreria visibile all’esterno con un identificatore univoco
  - Nel nostro caso il prefisso `qupcp_` è più che sufficiente

Potenziali conflitti sui “nomi”

- La libreria QUPCP quindi esporterà i seguenti identificatori

```
#ifndef _QUPCP_H
#define _QUPCP_H

extern const int qupcp_N;
extern bool qupcp_find(int, int);
extern void qupcp_unite(int, int);
extern void qupcp_init_quick_find_pcp();

#endif
```

Potenziali conflitti sui “nomi” (C++)

- In C++ la libreria QUPCP potrebbe essere scritta nel modo seguente

```
#ifndef _QUPCP_H
#define _QUPCP_H

namespace QUPCP {
    extern const int N; // Check this.
    bool find(int, int);
    void unite(int, int);
    void init_quick_find_pcp();
}

#endif
```

Costruzione effettiva di librerie

- Dipende dalla piattaforma
- Su sistemi di derivazione UNIX si usa il programma **`ar`** (o simili)
- Su Windows si usa il programma **`link`**
- Per altri sistemi si faccia riferimento agli appositi manuali

Memoria dinamica in C/C++

- Finora abbiamo visto solo dichiarazioni e definizioni di oggetti C/C++ che vengono allocati sullo stack di sistema (automatic memory in C/C++)

- L'esempio UNION-FIND, di fatto utilizzava una "memoria statica" (l'array di dimensioni fisse) per rappresentare in memoria gli elementi degli insiemi e le loro relazioni di appartenenza

- Il C ed il C++ ci obbligano a gestire esplicitamente la memoria dinamica (free store o heap in C/C++)
  - Non esiste la nozione di "garbage collection" (raccolta di rifiuti) come in Java, Lisp, ecc. ecc.
    - *C/C++ programmers think that memory management is too important to be left to the computer*
    - *Lisp (and Java, and R, and Python, and Ruby, and Go, and …) programmers think that memory management is too important to be left to the programmer*

Memoria dinamica in C/C++

- In C la memoria dinamica viene allocata e de-allocata usando la coppia di funzioni `malloc` e `free` (e derivati)

```
int *p = (int*) malloc(10 * sizeof (int));
int i;
*p = 1; *(p + 1) = 42; *(p + 2) = 0;
for (i = 0; i < 10; i++) printf("%d\n", *(p + i));
free(p);
```

Questo frammento alloca un puntatore a 10 interi nel free store, assegna i numeri 1, 42 e 3 nelle prime tre posizioni e poi stampa i 10 elementi.
Alla fine, la memoria viene riposta nel free store con la chiamata a **`free`**.

  - Il frammento contiene almeno un errore di prassi che vedremo successivamente

- In Java, od in qualunque altro linguaggio con *garbage collection*, **`free`** sarebbe una no-op

Memoria dinamica in C/C++

- La funzione **`malloc`** restituisce un puntatore di tipo `void*` ad una zona di memoria nel free store
  - Quindi è necessario inserire tutte le necessarie conversioni di tipo per evitare problemi con il compilatore
  - Le dimensioni del blocco di memoria ritornato dipendono dal parametro passato alla funzione
  - Se non vi è memoria disponibile, **`malloc`** restituisce un puntatore nullo; quindi bisogna sempre controllare il risultato di un'allocazione nel free store

Memoria dinamica in C/C++

- In C++ si usano gli operatori **`new`** e **`delete`** per manipolare il free store
- Il frammento precedente diventa

```
int *p = (int*) new int[10];
*p = 1; *(p + 1) = 42; *(p + 2) = 0;
for (int i = 0; i < 10; i++) cout << *(p + i) << endl;
delete [] p;
```

- Si noti la presenza di **`[]`** per denotare la deallocazione di un array
- Anche in questo caso è necessario controllare il valore restituito da **`new`**
- **`new`** e **`delete`** funzionano anche con i costruttori di istanze

Conclusioni

- Organizzazione del codice
- Compilazione separata
- Il preprocessore C
- Header files ed il preprocessore C
- Organizzazione di librerie

- Introduzione alla gestione della memoria
- Prossima lezione

  – Modifiche alle dichiarazioni
    - Costanti e, per il C++, `const`
    - Direttive per il linking (`extern`) e linking

  – Input/Output e files