## TECNICHE DI OTTIMIZZAZIONE DEL CODICE

- CONSTANT FOLDING

Anticipa i valori costanti presenti nel codice. (anche ND DO!)

```
int f() {
    return 8+(4/2)*3;   ⇒   return 23;
}
```

- CONSTANT PROPAGATION

I valori costanti vengono propagati nel codice dove possibile

```
int f() {
    int x = 8;                  ② CP          ② CP
    return x-2;   ⇒   return 8-2;   ⇒   return 6;
}
```

- COMMON SUBEXPRESSION ELIMINATION

Sostituisce occorrenze multiple della stessa sottoespressione

```
int f(int x, int y) {
    return (x+y)*(x+y);   ⇒   int z = x+y;
                              return z*z;
}
```

- LOOP INVARIANT CODE MOTION

Sposta fuori da un ciclo le espressioni che non dipendono dal ciclo stesso.

```
int f(int x, int n) {                          int f(int x, int n) {
    int s=1;                                       int s=1;
    for ( ; n>0; n--)                              int z=7*x;
        s+= 7*x;   ⇒                               for ( ; n>0; n--)
                                                       s+= z;
    return s;                                      return s;
}                                              }
```

sottoespressione che non dipende dal ciclo.

ATTENZIONE: NON SEMPRE APPLICABILE !!!

```
int f(int x, int n) {
    int s=1;
    for ( ; n>0; n--)
        s+= g(x);
    return s;
}
int g(int k) {
    printf("PIPPO\n");
    return k+1;
}
```

NON OTTIMIZZABILE CON "-O1 -fN"

- LOOP UNROLLING

Data un ciclo con numero di iterazione noto a tempo di compilazione, riduce o elimina le iterazioni.

```
int f(){                                       int f(){
    int a=0;                                       int a=0;
    for (int i=0; i<40; i++)   ⇒                   a++;
        a++;                                       a++;
    return a;                                      ⋮ (×40)
}                                                  a++;
                                               return a;
                                           }
```

Come facciamo se il numero di iterazioni è noto solo in esecuzione?

```
int sum (int*v, int n) {                       int sum (int*v, int n) {
    int i, s=0;                                    int i, s=0;
    for (i=0; i<n; i++)   ⇒                        for (i=0; i<(n/4); i+=4) {
        s+= v[i];                                      s+= v[i];
    return s;                                          s+= v[i+1];
}                                                      s+= v[i+2];
                                                   }
                                                   for ( ; i<n; i++)
                                                       s+= v[i];
                                               return s;
                                           }
```

nota lo stampo di esecuzione

Esegue ricorda la iterazione per [4/4] volte

Esegue ricorda di iterazione e per max k-1 volte

k-1

k

k operazioni per iterazione

- REGISTER ALLOCATION

Massimizza l'uso dei registri.

ATTENZIONE ottimizzazione piattaforma dependent

- OPERATOR STRENGHT REDUCTION

Sostituisce una istruzione con altra equivalente ma meno costosa

```
x/8          x>>3
x*66   ⇒     x<<6
x+2          x+x
x*15         (x<<4) - x
```

```
void f(int*v, int n, int k) {                  void f(int*v, int n, int k) {
    int i;                                         int i, c=0;
    for (i=0; i<n; i++)   ⇒                        for (i=0; i<n; i++) {
        v[i]= i*k;                                     v[i]= c;
}                                                      c+= k;
                                                   }
                                               }
```

- DEAD CODE ELIMINATION

Elimina il codice non raggiungibile

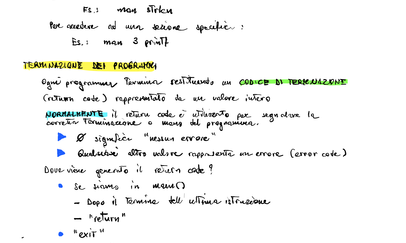
- RIDUZIONE SPAZIO DI ALLOCAZIONE

Riorganizzazione dei campi di una struct (del più grande al più piccolo) per minimizzare il padding.

LA DEVE APPLICARE IL PROGRAMMATORE

- ALLINEAMENTO IN MEMORIA

Ogni dato di dimensione S byte viene caricato in memoria ad indirizzo X multiplo di S.




DOCUMENTAZIONE POSIX / LIBC

- Sul web → sito del "open group"
- comando "man"

Man prende diverse sezioni tra cui:

① COMANDI DEL TERMINALE (SHELL)
  Es.: man ls

② SYSTEM CALL
  Es.: man fork

③ FUNZIONI LIBC
  Es.: man strlen

Per cercare ed una sezione specifica:
  Es.: man 3 printf

## TERMINAZIONE DEI PROGRAMMI

Ogni programma termina restituendo un CODICE DI TERMINAZIONE (return code) rappresentato da un valore intero

NORMALMENTE il return code è utilizzato per segnalare la corretta terminazione o meno del programma.

- ▶ 0 significa "nessun errore"
- ▶ Qualunque altro valore rappresenta un errore (error code)

Dove viene generato il return code?

- Se siamo in main()
  - Dopo il termine dell'ultima istruzione
  - "return"
- "exit"
- "abort"

## GESTIRE LA TERMINAZIONE DI UN PROGRAMMA

È possibile configurare nel programma una o più funzioni di gestione della terminazione (EXIT HANDLER)

```
#include <stdlib.h>
int atexit ( void (*gestore)(void)) ;
```

① ② ③

Es.:

```
void f() {
    ⋮
}
main() {
    ⋮
    atexit(f);
    ⋮
}
```

## TIPOLOGIE DI ERRORE

① ERRORI NON RECUPERABILI

Questi errori portano alla terminazione immediata del programma. abort() genera un errore non recuperabile

② ERRORI RECUPERABILI

Questi errori pongono il programma in uno stato che potrebbe o meno richiedere la sua terminazione (la scelta è del programmatore)

Per restituire il codice di errore spesso le librerie scrivono il codice di errore nella variabile globale ERRNO

## PATTERN DI GESTIONE DEGLI ERRORI

```
int mia_funzione() {
 int res, *buffer = NULL;
 ...
 buffer = malloc(...);
 if (buffer == NULL) goto cleanup; // errore nella chiamata
 ...
 res = chiamata(...);
 if (res != 0) goto cleanup; // errore nella chiamata
 ...
 free(buffer);
 return 0;

cleanup:
 ...
 if (buffer != NULL) free(buffer); // deallocazione buffer
 return -1;
}
```