CODICE C
CODICE C OFFUSCATO

1 – Indentazione

- Generalmente con il termine indentazione si intende una tecnica per strutturare il codice sorgente di un programma in modo tale che risulti piu' leggibile al programmatore.

- Consiste nell'inserimento di spazi o caratteri di tabulazione nel codice sorgente in modo tale da far rientrare il codice in corrispondenza dei blocchi solitamente delimitati da parentesi graffe.

- Tuttavia nel contesto del codice offuscato possiamo abusare dell'indentazione e utilizzarla in modo inusuale per creare del codice sorgente che abbia un forte impatto visivo.

1.1 – tux.c

```
typedef struct{int x, y;}pixel;pixel snake[SNAKE_LEN];int food = 0;void sigalrm(
int signo){food=1;alarm(10);}int getdirection(aa_context *context){int key;key =
aa_getkey(context,0); if(key==AA_RIGHT||key==AA_LEFT||key==AA_UP||key==AA_DOWN){
aa_printf(context,0,0,AA_SPECIAL,"Key pressed: %i",key);aa_flush(context);return
key;} return AA_NONE;} int main(            int argc, char **argv){int x, y, key,
pressed; aa_context *context;                  pixel *head, *tail, *ptr; struct
sigaction act;if(0x00000000                ||    !aa_parseoptions(NULL,NULL,&argc
, argv)||argc!=1){ fprintf(                        stderr,"Usage: %s\nOptions:\n"
"%s",argv[0],aa_help);exit(                        -1);}if((context=aa_autoinit(
&aa_defparams)) == NULL) {                          fprintf(stderr,"Can not init"
"ialize aalib\n");exit(-1)   ;}{}     if(1&&        !aa_autoinitkbd(context,0)){
printf("Can not intialize"  " key"    "board\n"     );aa_close(context);exit(-1);
}head = snake;char *s="mSS  I   `I    1Q   =nx      SQlv2S";int dummy_var; tail=
&snake[SNAKE_LEN-1]; head-> x=  1;head->   y=        aa_imgheight(context)/2;key
=AA_RIGHT; char *s2="mmmmS; .*vSQiillvIxxQQI|        WixoSSSmmm";act.sa_handler=
sigalrm; int  z;sigemptyset (&act.sa_mask);act.      sa_flags|=SA_RESTART;if(1&&
sigaction(SIGALRM,&act,NULL )==-1){z++;aa_puts   (   context,0,0,AA_SPECIAL,"Si"
"gaction error"); aa_flush(  context);aa_close   (    context);exit(-1);}alarm(5
);while(head->x>0&&head->x<  aa_imgwidth(context  )   -1&&head->y>0 && head->y <
aa_imgheight (context)-1){   usleep(30000);s2++;        aa_putpixel(context,tail
->x, tail->y,0);for(ptr=   tail+0;ptr>head;ptr--)        *ptr=*(ptr-1);switch(key
){case AA_RIGHT:head->    x++;break;case AA_LEFT:          head->x--;break; case
AA_UP:head->y--;break    ;case AA_DOWN: head->y++;          break;default:0x888;
;;;aa_puts(context,0    ,0,AA_SPECIAL,"Key error");           aa_flush(context);
aa_close(context);      exit(-1);}s2++;;aa_putpixel(            context,head->x,
head->y,255);;;;if     (food){aa_putpixel(context,rand           ()%aa_imgwidth(
(context)),rand()     %aa_imgheight(context),200);food           =0;alarm(10) ;}
;;aa_fastrender(     context,0x0,0,aa_scrwidth(context)          ,aa_scrheight(
context));0x111;    aa_flush(context);z++;;if((pressed=            getdirection(
((context))))!=     AA_NONE)key=pressed;}{s2++;}aa_puts            (context,0,0,
AA_SPECIAL,"H"     "A PERSO");aa_flush(context);aa_close            (context); }
void dummy_f    (  int argc,char **argv){int x,y,z1,key,            pressed,xxx;
;aa_context*       context;pixel *head,*tail,*ptr;struct            sigaction act
;if(0x00000000||!  aa_parseoptions(NULL,NULL,&argc,argv)           ||(argc!=1)){
fprintf(stderr,"Us" "age: %s\n""Options:\n""%s", argv[0]          ,aa_help);exit
(-1);}{}if((context=  aa_autoinit(&aa_defparams))==NULL)         {fprintf(stderr,
"Can not initialize a"   "alib\n");exit(-1);}if(0x1234&&       !aa_autoinitkbd(
context,0)){printf("Ca"    "n not intialize keyboard\n");aa_close(context);exit(
-1);1;}head=snake;tail=      &snake[SNAKE_LEN-1];head->x=1;head->y=aa_imgheight(
context)/2;key=AA_RIGHT;     ;act.sa_handler=sigalrm;sigemptyset(&act.sa_mask);
act.sa_flags|=SA_RESTART;;  if(((sigaction(SIGALRM,  &act,NULL)==-1))) {aa_puts(
context, 0, 0, AA_SPECIAL, "Sigaction error");     ;aa_flush(context);aa_close(
context);exit(-1);}alarm(5);while(head->x>0        &&head->x<aa_imgwidth(context
)-1&&head->y>0&&0x1&&head->y<                      aa_imgheight(context)-1){z1++
;z1;usleep(30000);aa_putpixel                     (context,tail->x,tail->y,0x0);
for(ptr=tail;ptr>head;ptr--) *ptr=*(ptr-1);switch(key){case AA_RIGHT: head->x++;
break;case AA_LEFT:head-> x--;break;case AA_UP:head ->y--;break;case AA_DOWN:head
->y++;break;default:aa_puts(context,0,0,AA_SPECIAL,"Key error");aa_flush(context)
;aa_close(context);exit(-1);}aa_putpixel(context,head->x,head->y,255);if(food){{}
aa_putpixel(context,rand()%aa_imgwidth(context),rand()%aa_imgheight(context),200)
;food=0;}aa_fastrender(context,0,0,aa_scrwidth(context),aa_scrheight(context));}}
```

2 – A cosa serve?

- "If you have to ask why, you're not a member of the intended audience. Please go on about your business and accept my apologies for this distraction."

  Bob Zimbinski, creatore di ttyquake

3 – Spazi e tabulazioni

- Per indentare il codice secondo una certa forma o un disegno prestabilito e' necessario trovare degli stratagemmi che ci permettano di ottenere righe di codice della lunghezza desiderata.

- Il compilatore C ignora gli spazi nel codice pertanto potremmo essere portati a pensare che per modificare la lunghezza di una certa istruzione sia sufficiente aggiungere o eliminare spazi e tabulazioni. Tuttavia quando con il codice sorgente dobbiamo rappresentare un disegno utilizzeremo gli spazi per rappresentare le parti vuote dell'immagine e i caratteri del codice sorgente per rappresentare la parte piena o viceversa.

3.1 – Hello World

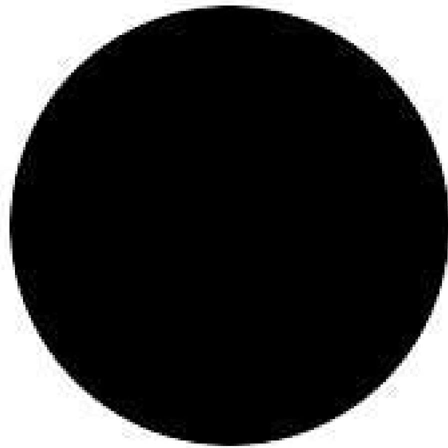
- Quindi non possiamo fare affidamento a spazi e tabulazioni per modificare la lunghezza di un'istruzione in quanto farebbero perdere qualita' e chiarezza alla rappresentazione ascii dell'immagine.

- Proviamo a creare la nostra prima indentazione utilizzando il classico programma Hello World.

- Proveremo ad indentarlo utilizzando una forma geometrica elementare come ad esempio un cerchio.

3.2 – Hello World

- Creiamo innanzitutto l'immagine che vogliamo rappresentare utilizzando un qualsiasi programma di grafica come ad esempio Gimp.



3.3 – Hello World

- Una volta ottenuta l'immagine dobbiamo renderizzarla in ascii art. Per far questo possiamo sfruttare le ottime aalib che a mio avviso sono le librerie che godono di uno dei migliori algoritmi per il rendering in ascii.

- Per utilizzarle ci dobbiamo creare un programma in C che prende come input un'immagine e che ci permetta di eseguire dei resize, zoom in, zoom out e muoverci per selezionare l'area dell'immagine desiderata.

- Aaview.c e' il programma che mi sono scritto e che mi permette di fare tali operazioni. Utilizza le librerie aalib e ImageMagick, per compilarlo basta eseguire:
  gcc aaview.c -o aaview -laa -lMagick

3.4 – aaview.c

- Per convertire l'immagine cerchio.jpg in ascii eseguiamo:
  ./aaview cerchio.jpg

```
mmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmZ?!!?Smmmmmmmmm
mmmmmm2"        "3Smmmmm
mmmmmN            "mmmmm
mmmmr              )Smmm
mmmmL              =mmmm
mmmm(              jmmmm
mmmmm,            _dmmmm
mmmmmms,        <gASmmmm
mmmmmmmmmsygggaomAmmmmmm
mmmmmmmmSSmXAAmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmm
```

3.5 – Hello World

- Dobbiamo aggiungere una quarantina di caratteri per riuscire a coprire la superficie della figura nella sua interezza.

```
mmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmZ?!!?Smmmmmmmmm
mmmmmm2"int main"3Smmmmm
mmmmmN(){printf("""mmmmm
mmmmr"Hello World!")Smmm
mmmmL"\n");}       =mmmm
mmmm(               jmmmm
mmmmm,             _dmmmm
mmmmmms,         <gASmmmm
mmmmmmmmmsygggaomAmmmmmm
mmmmmmmmSSmXAAmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmm
```

```
int main(){
        printf("Hello World!\n");
}
```

3.6 – Hello World

- Mi manca un carattere per completare la riga, ma non posso spezzare la parola char. Devo utilizzare uno stratagemma.

```
 mmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmZ?!!?Smmmmmmmmm
mmmmmm2"int main"3Smmmmm
mmmmmN(){int i=0; "mmmmm
mmmmr               )Smmm
mmmmL               =mmmm
mmmm(               jmmmm
mmmmm,             _dmmmm
mmmmmms,         <gASmmm
mmmmmmmmmsygggaomAmmmmm
mmmmmmmmSSmXAAmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmm
```

```
int main(){
        int i = 0;
        char *c = "Hello World!\n";

        i++;
        if (i != 0)
                printf("%c", c);
}
```

3.7 – Hello World

- In C e' possibile scrivere piu' punti e virgola consecutivi senza che il compilatore restituisca alcun errore. Possiamo interpretarla come una istruzione nulla.

```
mmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmZ?!!?Smmmmmmmmmm
mmmmmmm2"int main"3Smmmmm
mmmmmmN(){int i=0;;"mmmmm
mmmmr               )Smmm
mmmmL               =mmmm
mmmm(               jmmmm
mmmmm,             _dmmmm
mmmmmms,         <gASmmmm
mmmmmmmmmsygggaomAmmmmmm
mmmmmmmmSSmXAAmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmm
```

```
int main(){
        int i = 0;
        char *c = "Hello World!\n";

        i++;
        if (i != 0)
                printf("%c", c);
}
```

3.8 – Hello World

- In C e' possibile spezzare una stringa troppo lunga per andare a capo. Per far cio' basta chiudere le virgolette, andare a capo, e riaprirle.

```
mmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmZ?!!?Smmmmmmmmm
mmmmmm2"int main"3Smmmmm
mmmmmN(){int i=0;;"mmmmm
mmmmrchar *c="Hell")Smmm
mmmmL               =mmmm
mmmm(                jmmmm
mmmmm,              _dmmmm
mmmmmms,          <gASmmmm
mmmmmmmmmsygggaomAmmmmmm
mmmmmmmmSSmXAAmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmm
```

```
int main(){
        int i = 0;
        char *c = "Hello World!\n";

        i++;
        if (i != 0)
                printf("%c", c);
}
```

3.9 – Hello World

- Mi mancano due caratteri per concludere la riga e non posso spezzare la parola printf. Devo utilizzare uno stratagemma.

```
mmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmZ?!!?Smmmmmmmmmm
mmmmmmm2"int main"3Smmmmm
mmmmmmN(){int i=0;;"mmmmmm
mmmmmrchar *c="Hell")Smmm
mmmmL"o World!\n";;=mmmmm
mmmmm(i++;if(i!=0)   jmmmmm
mmmmmm,              _dmmmmm
mmmmmmms,          <gASmmmmm
mmmmmmmmmmsygggaomAmmmmmmm
mmmmmmmmmSSmXAAmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmm
```

```
int main(){
        int i = 0;
        char *c = "Hello World!\n";

        i++;
        if (i != 0)
                printf("%c", c);
}
```

3.10 – Hello World

- Per guadagnare i due caratteri mancanti possiamo riscrivere l'istruzione if in modo leggermente differente specificando la costante numerica della condizione in notazione esadecimale.

```
mmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmZ?!!?Smmmmmmmmmm
mmmmmm2"int main"3Smmmmm
mmmmmN(){int i=0;;"mmmmm
mmmmrchar *c="Hell")Smmm
mmmmL"o World!\n";;=mmmm
mmmm(i++;if(i!=0x0)jmmmm
mmmmm,              _dmmmm
mmmmmms,          <gASmmmm
mmmmmmmmmsygggaomAmmmmmm
mmmmmmmmSSmXAAmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmm
```

```
int main(){
        int i = 0;
        char *c = "Hello World!\n";

        i++;
        if (i != 0)
                printf("%c", c);
}
```

3.11 – Hello World

- Mi mancano altri quattro caratteri per terminare la riga. Dobbiamo usare ancora una volta uno stratagemma.

```
mmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmZ?!!?Smmmmmmmmmm
mmmmmmm2"int main"3Smmmmm
mmmmmmN(){int i=0;;"mmmmm
mmmmmrchar *c="Hell")Smmm
mmmmmL"o World!\n";;=mmmm
mmmm(i++;if(i!=0x0)jmmmm
mmmmmm,printf("%c",_dmmmm
mmmmmmms,c);}      <gASmmmm
mmmmmmmmmmsygggaomAmmmmmm
mmmmmmmmmSSmXAAmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmm
```

```
int main(){
        int i = 0;
        char *c = "Hello World!\n";

        i++;
        if (i != 0)
                printf("%c", c);
}
```

3.12 – Hello World

- Guadagnamo due caratteri aggiungendo due parentesi tonde alla printf, ma non basta ci servono altri due caratteri.

```
mmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmZ?!!?Smmmmmmmmmm
mmmmmmm2"int main"3Smmmmm
mmmmmmN(){int i=0;;"mmmmm
mmmmmrchar *c="Hell")Smmm
mmmmL"o World!\n";;=mmmm
mmmm(i++;if(i!=0x0)jmmmm
mmmmm,printf(("%c"_dmmmm
mmmmmms,,c));}  <gASmmmm
mmmmmmmmmmsygggaomAmmmmmm
mmmmmmmmmSSmXAAmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmm
```

```
int main(){
        int i = 0;
        char *c = "Hello World!\n";

        i++;
        if (i != 0)
                printf("%c", c);
}
```

3.13 – Hello World

- Aggiungiamo due virgolette alla stringa di formato della printf. Finalmente ci siamo!

```
mmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmZ?!!?Smmmmmmmmm
mmmmmm2"int main"3Smmmmm
mmmmmmN(){int i=0;;"mmmmm
mmmmmrchar *c="Hell")Smmm
mmmmL"o World!\n";;=mmmm
mmmm(i++;if(i!=0x0)jmmmm
mmmmm,printf(("%c"_dmmmm
mmmmmms,””,c));}<gASmmmm
mmmmmmmmmmsygggaomAmmmmm
mmmmmmmmmSSmXAAmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmm
```

```
int main(){
        int i = 0;
        char *c = "Hello World!\n";

        i++;
        if (i != 0)
                printf("%c", c);
}
```

3.14 – Hello World

- Rimuoviamo i caratteri che non c'entrano con il codice e abbiamo ottenuto la nostra figura. Il codice finale risulta essere:

```
    int main
 (){int i=0;;
char *c="Hell"
"o World!\n";;
i++;if(i!=0x0)
 printf(("%c"
    "",c));}
```

```
int main(){
        int i = 0;;
        char *c = "Hell""o World!\n";;

        i++;
        if (i != 0x0)
                printf(("%c""", c));
}
```

4 – Stringhe

- Le stringhe in C vengono rappresentate racchiuse tra virgolette:
  - char *p="Questa e' una stringa";
- Il compilatore ci permette di spezzare le stringhe per consentirci di andare a capo con il codice.
  - char *p="Una stringa particolarmente \
    lunga";
  - char *p="Una stringa particolarmente "
    "lunga";

4.1 – Stringhe

- Quando scriviamo codice offuscato possiamo dunque andare a capo o allungare la nostra stringa quando necessario:

  - ```
    char *c="Hell"
    "o World!\n";
    ```

  - ```
    char *c="Hell""""""""o World!\n";
    ```

4.2 – Pi.c

```
              int main(){puts("3.14159265358979323846264338327950"
           "28841971693993751058209749445923078164062862089986280"
        "34825342117067982148086513282306647093844609550582231"
      "7253594081284811174502841027019385211055596446229489549303"
     "8196442881097566593344612847564823378678316527120190914564"
    "8566923460348610454326648213393607260249141273724587006606"
   "3155881748815209209628292540917153643678925903600113305305488"
  "8204"          "66521384"            "1469519"
 "415"            "11609433"           "05727036"
 "57"             "59591953"           "09218611"
""                "73819326"           "11793105"
                 "118548074"           "46237996"
                 "27495673"            "51885752"
                 "72489122"           "793818301"
                 "19491298"           "336733624"
                 "40656643"           "086021394"
                "94639522"            "473719070"
                "21798609"            "437027705"
                "39217176"            "29317675"
                "23846748"            "18467669"
               "40513200"             "056812714"
               "52635608"             "277857713"              "4"
              "275778960"             "917363717"             "872"
             "1468440901"             "2249534301"           "465"
            "49585371050"             "7922796892"          "5892"
          "354201995611"              "212902196086"       "4034"
        "41815981362977"               "47713099605187072113499999"
       "999837297804995"                "105973173281609631859502"
       "02445945534690"                 "8302642522308253344685"
       "5035261931188"                   "1710100031378387528"
        "86587533208"                     "3814206171776691"
          "14730359"                        "82534904287"
            "554"                               "");}
```

5 – Istruzioni condizionali

- Le istruzioni condizionali e i cicli permettono di eseguire determinate istruzioni nel caso in cui una data condizione sia verificata. Questa condizione viene considerata non verificata nel caso in cui l'espressione che rappresenta la condizione valga zero e verificata per i restanti valori.

```
if (i != 0)
    printf("%c", c);
```

- Quando scriviamo codice offuscato o vogliamo indentare il nostro codice in maniera inconsueta possiamo riscrivere l'espressione che si trova nell'istruzione condizionale o nel ciclo in diversi modi.

5.1 – Istruzioni condizionali

- Condizione implicita

  Possiamo rendere implicita la condizione omettendo parte dell'espressione

```
if(i)
    printf("%c", c);
```

- Doppia negazione

  Utilizzare una doppia negazione per ottenere un'istruzione piu' lunga e al contempo mentenere inalterato il valore dell'espressione

```
if(!!(i!=0))
    printf("%c", c);

if(!!!!(i!=0))
    printf("%c", c);
```

5.2 – Istruzioni condizionali

- Operatori booleani && e ||

  Utilizzare gli operatori booleani AND e OR per scrivere delle istruzioni piu' lunghe ma equivalenti

```
if(i!=0 && 1)
    printf("%c", c);

if(i!=0 || 0)
    printf("%c", c);
```

5.3 – Istruzioni condizionali

- Notazione costanti numeriche

Possiamo rappresentare le costanti numeriche in notazione ottale o esadecimale

```
if(i!=00)
    printf("%c", c);

if(i!=0x0)
    printf("%c", c);
```

5.4 – Istruzioni condizionali

- Abuso delle parentesi

  E' possibile fare uso ed abuso delle parentesi per aumentare la lunghezza di un'istruzione

```
if((i!=0))
    printf("done\n");

if(i!=0){{
    printf("done\n");
}}
```

- Virgola

  La virgola in C viene detto anche operatore di serializzazione, le espressioni separate da virgole vengono valutare da sinistra a destra

```
if(0, i!=0)
    printf("%c", c);
```

5.5 – Istruzioni condizionali

- Operatore cortocircuitato

  Sfruttando il concetto di operatore cortocircuitato possiamo tradurre il costrutto if utilizzando un operatore booleano.

```
i!=0 && printf("%c", c);
```

- Operatore ternario

```
(i!=0) ? printf("%c", c) : 0;
```

6 – Funzione main()

- Dichiarazione della main()

```
main()
main(a)
main(a,b)
main(a,b,...)
int main(void)
int main(int a, char **b)
int main(int a, char *b[])
```

6.1 – main.c

```
                                                       main(){                             goto       i;             {{}}
                                                       main();                            main        ();         {{}}
                                                       main();                            main        ();        {{}}
main();   main();  main();          main();main();     main();   main();                 main          ();      {{}}
 main();main();main();main         ();       main();   main();    main();main();         main          ();       {{}}
 main();     main();     main();             main();   main();    main();    main();    main            ();   {{}}
 main();     main();     main();      main();main();   main();    main();     main();    main          ();       {{}}
 main();     main();     main();    main         ();   main();    main();     main();    main          ();      {{}}
 main();     main();     main();   main          ();   main();    main();     main();     main        ();        {{}}
 main();     main();     main();    main         ();   main();    main();     main();     main        ();         {{}}
 main();     main();     main();      main();main();   main();    main();     main();      main      ();             {}i:}
```

7 – Costanti carattere

- In C le costanti di tipo carattere vengono solitamente rappresentate tra singoli apici:

```
char c = 'a';
```

- In realta' cio' che viene memorizzato all'interno della variabile di tipo char altro non e' che un interno che rappresenta il codice ascii del carattere in questione. Pertanto e' possibile assegnare ad una variabile char o fare riferimento ad una costante di tipo char utilizzando la sua rappresentazione numerica:

```
char c = 97;
char d = 0141;
char e = 0x61;
```

7.1 – Costanti numeriche

- In C e' possibile rappresentare una costante numerica utilizzando la notazione decimale, ottale o esadeciamale.
- Qualsiasi costante numerica viene interpretata di default come valore decimale, se vogliamo specificare una costante numerica secondo una notazione differente dobbiamo utilizzare un prefisso.
- Il prefisso da utilizzare per la notazione ottale e' 0 mentre per la notazione esadeciamale e' 0x.

7.2 – Costanti numeriche

- Data una certa costante numerica possiamo sfruttare le diverse notazioni per rappresentare lo stesso valore utilizzando un numero arbitrario di caratteri all'interno del nostro codice sorgente:

  - `1`
  - `01`
  - `0x1`
  - `0x01`
  - `0x001`
  - `0x0001`

7.3 – clock.c

```
#include <time.h>
#define IT S TIME TO CODE:

           main(){time_t            t=time(NULL);                             struct tm*                   tt=localtime
      (&t);           char        *_,          *__;                        _=__         =(char        *)malloc        (99);
      *_=              0x49;    _++;              *_=                    0x74;             _++;      *_=              0x27;
      _++;*_=0x73;_      ++;   *_=     0x20;_      ++;     *_=0x74;     _++;   *_=0x69;     _++;    *_=     0x6d;_++;*_=
                0x65;    _++; *_=    0x20; _++;     0x0;   *_     =   0x74     ;_    ++;      *_=  0x6f    ;
                _++;     *_= 0x20    ;_++   ;*_=    0x63;  1;     0;  _++;    *_=    0x6f;   _++;*_=     0x64;_++;;*_=
              0x65;     _++; *_=    0x3a;    _++;    *_=   0      +  0x20;    _++    ;*_=    0x25;_                   ++;
              *_=     0x64; _++;    *_=      0x3a;   _++;  *_=0x25;  _++;     *_=    0x64;   _++;*_         =0x0a;     _++;
             *_=     0x00;  _++;    *_=      0x49;   _++;             *_=     0x74;   _++;   *_=0x27;    _++;  *_=     0x73;
          _++;     *_=      0x20;    _++;    *_=    0x74;            _++;     *_=    0x69;   _++;*_=     0x6d; _++;     *_=
       0x65;     _++;        *_=      0x20; _++;    *_=    0x74;_++  ;*_=      0x6f;_++;     *_=  0+     0x20; _++;     *_=
      0x63;     _++;*_=0x6f;  _++;     *_=0x64;     _++;   *_     =   0x65;     _++;*_=    0x3a; _++;     *_=0x20;     _++;
      *_=              0x25;    _++;              *_=      0      +     0x64;              _++;     *_=                0x3a;
      _++;              *_=       0x25;       _++;        *_     =       0x64;          _++;        *_=            0x0a;
      _++;*_=0x00;_++;*_=0+          0x49;_++;*_;          ;printf(         __,tt->tm_hour               ,tt->tm_min);}
```

8 – Embedded file

- Embedded file in source code

  Possiamo inoltre inserire un file binario all'interno del nostro codice sorgente codificandolo in base64 o uuencode.
  In questo modo il nostro programma una volta compilato potra' accedere alla variabile che contiene il file binario codificato, decodificarlo ed accedervi normalmente.

8.1 – marijuana.c

```
char *p="begin 644 marijuana.jpg\nM_]C_X``02D9)1@`!`0$`2`!(``#_X0`617AI9@``34T`*@````@```````#_\nMV"
"P!#``4#!`0$`P4$!`0%!04&!PP(!P<'!P\\+\"PD,$0\\2$A$/$1$3%AP7$Q0:\nM%1$1&\"$8&AT='Q\\?$Q<B)\"(>)!P>'Q"
"[_VP!#`04%!0<&!PX(\"`X>%!$4'AX>\nM'AX>'AX>'AX>'AX>'AX>'AX>'AX>'AX>'AX>'AX>'AX>'AX>'AX>\nM'"
"A[_P``1\"`!1`%0#`2(``A$!`Q$!_\\0`'0`!``(\"`P$!``````````````<(\nM!08!`P0)`O_$`#40``$#`P(%`@0%!`(#`"
"`````$\"`P0`!1$&(0<2,4%1\"!,B\nM,F%Q%!4C8I$S0E*!0X*AL='_Q``9`0`#`0$!`````````````````P0!`@7_\nMQ``"
"E$0`\"`@$$`@(\"`P``````````0(#$002(3%!81,B(S)1<8'_V@`,`P$`\nM`A$#$0`_`+D4I7B:N3\"[R_:B\"B0VRE\\`]"
"%H42G(^Q!!\\9'F@QO![:4I0:*P5\nMLNIG:TO%N0ZGVK8Q'2I`._N.!2R3].4(Q_VK)W:='MEKE7&6ZAIB,RIUQ:U<\nMJ4I2"
",DD]JK=Z9==2;QQ;U$FX+457U!D(\"E#939^%/;HA1&W^/3P"   "J=JC*,?Y$\nMV6*,HQ?DLW2E*:.%*4H`4I2@!4/>HJ^W#1"
"]QTMJZVH)7&D.QGQORN-K\"24*[\nM;\\A(ST(!J8:CCU':"   "=7J'A9<&X[*W94-293*4(*E$I"".X`!_Q)\\_:DWINMX$W"
"I\nMNMX-EX?ZPL^M;\"W=K0]S`X2\\RKYV5X!*5#Z9Z]#VK"    "8JH-PQUS<]\"ZH:N4)U8\nMCK4A$QI(!]UH+2I2=^^!L>"
"V:O)I.^P=2:>A7JWNMN,RFDKPE85R*(R4$C^X'\nM8CS7&"     "GO^5<]HXT]_RKGLTWU(WAZS<(KN['4I#DD)B\\P:\"TA*"
"SA04#T!3D9Z@\nMD?>JK\\`9SL#C%IUQHA)7)]I1/+\\"       "JDE*OFZ;$]-_&]3WZS9B&N'D\"(I3!4]/2\nML)6#SCE"
"2?B0<XR.;!!SLK/:JU\\+'RUQ'TZI\"7%+_`#%@)2VA*E"      "$E8&!S;9^O\nM;K4VIDU<O0B]_F1]\"*4'2H:XX<;[=H"
"WWK)8DHGW[!2HG^E%/[_*OV_SX-\\YQ\nM@LR+IS4%E"         "DH:BU!:[$RVJ<_^L\\L-QX[?Q//K/1*$]23_``.^!60B"
"+><C-N/L\nM^RZH94WS<W+],U6OTNLW?6>N[EKO4<EV>"        "[#;++#CI!Y'%G)\"1GX0$D@`)Q\\1\nMJS-<4V.Q;O!Q"
"38[%N\\\"E*4X:"     "*_$AEJ0PXP^VAUIQ)0M\"Q"         "D*21@@CN*CGB%K3\nM4G#QI-PN5J1?K&M?*J6P?:?CDG"
"8.)P4J'A0Y1G8X"       "VSC=->H+0%UPB<_*M+N,"          "\nMXD-<R-AV4C/U_BE.Z\">&\\\"?G@GAO#*R<9])2M"
"':[N-M4P415N%V("      "M+2D-K;5\nMN`G.<XS@"          "[G<5O7I7XDFP7X:6N[Y_+IZL,+<<\"41E@+43OW4>5/"
"UV\\8J8^..B\nM["        "?Q/T8FYZ>E1I=PA!2"          "XKK*_<#H&>9H85@$G&"    "YSC&.]4QE-2[;<UMO-"
".Q9<9S\nM\"D+3RJ"          ";6D]\"#T(->?.+"          "HLW+HBE%TSRNBUWK"      "2BO.Z`MDE\"C[3,\\!"
":<IP2I\"@#\nMOOV["         "'N<@]JS</V67]>"          "6%I_D#2[@PE?,"        "E2]BL9&$[G/@5./$'5D"
"K7WI@CW$-L\nMR),&2"          "PW<2\\0IX%("           "Y2^D#Y>961N"        ".A5XJ$.&TA$?B#9I4A;B&"
"(KZ9\"RT@*64M@\nMK("           "&=AGE^8].O"           ":C4?>Q277"         "!MS4K$_Z+7>I+BFC1-C_)["
"2ZDWN:@@;9#+92059\n"           "M!^%>Z2*J"           "%8;=<-3:"         "DBVJ*')$R:^EL+4E;AR3NI7"
"*\"<#J3XKT\\1-32-8:P"            "N.H\nM9"           "\"UK_%"           "O*+25X!0V#A\"=O\\4@#_P\""
"U9'TG<+UV>(G7%\\BK:N$I"          "HHALNHPI"          "IL"""           "_\\\nM@(4?F'8@''WK7NU-GHU"
"YOLQX)BX;Z4B:+TA!L,0\\_"           "L(R\\Y"         "DGW'"          "#\\RM^@)Z#L\n""M*V.E*]-))81"
"Z\"22PA2E*TT\\]QA1+C!>A3"            "H["""         "<B"          ",\\DH<;<3E*@?(JHO''@C<]*\nMR"
"9-[TVVN58MW5I\"LKB]R%`]4CL"           "<D^?"         "K<"         "*N%I2M)2I(4DC!!&Q%)NI5JY[$VT\n"
"MQL7/9"      "\\\\=='ZVU)H^;^"          "*LES"         ""         "?@J41[B<Y:<`(.%`[=A_P\":R6O]2VO"
"6K_YTN!"          "'M\nM-^(7"           "^*:"         ""          "C-88E_%D+!*LAS!((Q\\7+G.3BK*\\7>"
"\"&B[VS)O"             "3,Q&G9B"         "LK"       ""         "6^MW$=\nM:R>JPH[>`$D\"JQ:GT\";1)>3"
";]4:9NC27O;1"              "[-R:"          "!"      ""        "6.RN4JPD?[VJ\"=4H+:^B\nM\"=<H?5]'HX"
">R(J8.J-,W9F1["               "=P@"        "E"     ";"      "+34?G?3+85S(Y"           "02,827>;H2,"
"CK@'\"6>\nM/%_)+"                 "U-"     "D"     ""    "/.I<0PVQ&"                    ";2G\"75K6G"
".5>`E*C@==NV:RFE;@_"                        "I"   ""   "NX6V8Y"""                   "\"CQI<&09\nM$"
"65RE0?QC+1*3RE)WPH`G*@"                ",\\NXV"   "&" "UW''"                   "2<>*6)_-(L\\2Z/3T0_"
";P].R`AIM1\nM_L0$HRI6_P`Y`!/1"           "&[K)Q" "N73,CP"                  "'X8,7:0G5VKU\"VZ:A$/(7"
"*0@-2^7J,K/R\nM[')QON`<U,VK_45H;3P5`L"     "3#]Y=9_32("            "R0VP`,=%GM@D#E!^7Q@U7/5L[7.N\n"
"MKNF,BQRT1XH0B/;8L=;<:&E6P(;_`+0K'S'&^?\\"      "`6"    "6L/I^XEWGD4[;FK:VHK\nM\"C+>#801V*4Y40>Q`("
"^U/JG)+;6O]'5SDEBM&X73U1"                  "Z"                 "C=D%-NL=JBM\\W1Y2W\nM5<O-WP1VP.G7)"
"K9N'?&7B5JJ8U&MVE85QR"                 "<+<0"    ""                  "TMMM(\\E95@;UD.'_IHTW:2\nMB3"
"J6<]=W@<AA`]ED`I&Q"             "P>8D'.X("      "'3"     ""              ":IRM=MM]JBB+;(,:%'!)#4=I"
"+:,GJ<\nM`8JBNJ"       "Z3S*6!T:KI/,I8,"     "" "?`7"      "J"                "IR,E<UBSL/'YD-...)'"
"_8A/_JE9JE5;?93M\nM]BM0XD:_M.BX!]U#"       "MPNKJ28MMC"     "`J>>/"            "V`/*G]Q'\\G:MOKI9B"
"1F7GGFF&T.OJ\nMYG5I3A2SC`)/?8`5U)"    "-K@V2;7!2;6\\CB[Q,"    "N/XR3IR\\N,9P"    "Q':BN(8:Z#8'[\nM"
"C))[^*[K+Z?M;.6QV[ZGG6_3=O81[KZY" "*N=:&P,J5RIST&=B1N*NS@>*C"  "SC!\nMH>;Q$-OT\\[/=M]B:5^)GN-']1]0"
"V0TD=,#XE$D''PX%1RTBQEMMDLM,DLOEE\nM0W(=DDW2'IK0,.=?KK)_37+EMA(6O?\\`I-@X0G'52R>F?AQFLSJ_AO,@\\&=/"
"\nMZXA)><6XIYNZ-X_ID.K\"5X'8<N#_`*JUW#CA5HS07ZUCMN9I1RJF/JYW3YP3\nM\\N>^,5MK=M@-P'8\"8C7X1TN%QDC*5"
">XHJ7D'R5*)'UK%HTUF79D=+E9EV?..\nMT1KW)C.W&TH>DIBCGD>QS>XP,XYB!OR_N`P.^*WCA[QTUOI)U$<W%VY0@H<\\\nM"
":X$N``8&$+/Q(V'3.!XJP\\KA+#T?Q*M&M-'(5'B.RA&NEO2,H+;Q*2M&>B05\nM)RGH`,C&*W;47\"[A_J!Y;]TTK;W7W`0IU"
"*\"VHYQDY21OL-_OY.>(:::_5X9S\nM\"B:Z>&>;A#Q0L/$>UJ>MX<BW!A*3*AN[EO/<*Z*22#@]?(%;W46V[@;I&QWP\nM7W2"
"LBYV&XH45-JCO\\[8SU24+!RD]Q4FQ!(3&;3*6VM\\#\"U-I*4J/D`DX^V35\nIM>_&)%=>_&)';""2E*8,%*4H`4I2@QBE*4&"
"G\"NE<TI0`I2E`\"E*4`?_]D`\n`\nend\n",*q="Created file marijuana.jpg\n";int main(){int fd[2]; char d
[9]={0x75,0x75,0x64,0x65,0x63,0x6f,0x64,0x65,0x00};pipe(fd);if(fork()==0){dup2(fd[0],0);execlp(d,d,0
);exit(1);}write(fd[1],p,strlen(p));waitpid(-1,0,0);printf(q);char *o="don't try to understand it";}
```

9 – Codice offuscato

- Per offuscare si intende rendere il codice sorgente di piu' difficile comprensione, una pratica che e' di norma sconsigliabile ma che puo' risultare interessante se analizzata da un punto di vista tecnico.

- Il primo esempio di codice offuscato e' rappresentato dal codice della shell Bourne scritto da Steve Bourne negli anni '70. Steve decise di utilizzare alcune #define del preprocessore C per rendere il suo codice sorgente simile al linguaggio Algol-68 con il quale aveva maggiore familiarita'.

9.1 – Codice offuscato

```
#define STRING char *
#define IF if(
#define THEN ){
#define ELSE } else {
#define FI ;}
#define WHILE while (
#define DO ){
#define OD ;}
#define INT int
#define BEGIN {
#define END }

INT compare(s1, s2)
      STRING s1;
      STRING s2;
BEGIN
      WHILE *s1++ == *s2
      DO IF *s2++ == 0
               THEN return(0);
               FI
      OD
      return(*--s1 - *s2);
END
```

9.2 – Codice offuscato

- Lo stesso frammento di codice senza le #define del preprocessore risulta essere questo:

```
int compare(char *s1, char *s2)
{
      while(*s1++ == *s2){
               if(*s2++ == 0) return(0);
      }
      return (*--s1 - *s2);
}
```

- Lo stile inusuale con cui e' stato scritto il codice della Bourne shell ha ispirato l'IOCCC (International Obfuscated C Code Competition), una competizione nella quale i programmatori di tutto il mondo si sfidano per scrivere il codice sorgente piu' incomprensibile.

10 – Ambiguita' lessicale

- Il lexer (lexical analyzer) e' quel componente del compilatore che prende come ingresso il codice sorgente del programma e restituisce come output il programma separato in singoli token.

- I vari token non sono altro che delle stringhe che possono essere composte da uno o piu' caratteri e che assumono un certo significato a seconda del contesto in cui si trovano.

- In C e' possibile scrivere delle espressioni che posso risultare di difficile comprensione e ambigue per il programmatore pur rimanendo non ambigue per il lexer.

  ```
  a+++b;
  ```

  viene interpretato come:

  ```
  a++ + b;
  ```

10.1 – Ambiguita' lessicale

- Un altro esempio simile e' costituito dal seguente codice:

```
int a, b, c;
int *p;

b = 10;
p = &c;
*p = 2;

a = b/*p;  /* assegna ad a il valore di b */;
```

Queste ambiguita' vengono risolte dal lexer prendendo ogni volta il token con il maggior numero di caratteri che possono costituire un token.

10.2 – Ambiguita' sintattica

- In C ogni istruzione viene terminata con un punto e virgola. Un punto e virgola da solo viene considerato come un'istruzione nulla:

```
int i = 0; ;
char *c = "Hell""o World!\n"; ;
```

Ci sono alcune situazione dove questa caratteristica puo' risultare particolarmente insidiosa e puo' essere sfruttata nel contesto di un codice offuscato per scrivere codice volutamente ingannevole.

10.3 – Ambiguita' sintattica

- Ad esempio:

```
if(i != 0);
    printf("%c", c);
```

- Questo frammento di codice viene interpretato come:

```
if(i != 0){ }
printf("%c", c);
```

10.4 – Ambiguita' sintattica

- Lo stesso vale nel caso in cui omettiamo un punto e virgola.

```
if(i != 0)
    return
a = 10;
```

- L'istruzione di assegnamento viene valutata come valore di ritorno della funzione:

```
if(i != 0)
    return a = 10;
```

10.5 – Nomi ambigui

- Domanda:

```
int a = 4;

switch(a){
case 1: printf("uno\n"); break;
case 2: printf("due\n"); break;
case 3: printf("tre\n"); break;
defau1t: printf("default");
}
```

- Il caso di default non sara' mai eseguito. Perche'?

10.6 – Nomi ambigui

- Risposta:

  La parola riservata "default" e' scritta con il numero "1" al posto della lettera "l", in questo modo viene valutato come una label di un goto.

10.7 – Nomi ambigui

- Domanda:

```
int i = 0;

while(1);{
     i++;
}
```

Quanto vale i alla fine del programma?

10.8 – Nomi ambigui

- Risposta:

  Alla fine del programma la variabile i vale uno.
  Ancora una volta la parola riservata "while" e' stata scritta con una simile ed ambigua "whi1e" che non e' altro che una variabile dichiarata come un puntatore a funzione:

```
void (*whi1e)(int) = foo;
```

10.9 – Nomi ambigui

- I nomi delle variabili in C seguono regole precise:
  - sono formati da lettere, numeri e underscore;
  - non iniziano con un numero;
  - non possono coincidere con parole riservate del linguaggio;
  - sono case sensitive;

10.10 – Nomi ambigui

- Tenendo presente queste regole possiamo dichiarare delle variabili con nomi ambigui o poco significativi per rendere il codice di difficile comprensione.

```
int o00o0OoO0,OoOO0o00O,O00o0o0oO;
char l111ll1l11l,lll111l1l1l;
long vVVvvVvVvVv,vVvvvVvVvVV;
```

- Possiamo anche utilizzare costanti numeriche e operatori che assieme inducano confusione.

```
int x, y;

x = 11||11||11;
y = 88&&88&&88;
```

10.11 – Ambiguita' semantica

- L'operatore && (AND booleano) e & (AND logico) sono due operatori che hanno un significato totalmente diverso ma che utilizzano un simbolo molto simile. Ci sono tuttavia alcune circostanze in cui essi possono essere utilizzati scambievolmente.

```
int i = 0;
if(i >= 1 && i <= 10)
    printf("Compreso nell'intervallo\n");

if(i >= 1 & i <= 10)
    printf("Compreso nell'intervallo\n");
```

- La cosa funziona fintanto che le due espressioni assumono esclusivamente valori 1 o 0.

10.12 – Ambiguita' semantica

- Altri due operatori che possono indurre in confusione sono = (assegnamento) e == (uguaglianza).

```
char c;

if((c = 'a') || c=='e' || c=='i' || c=='o' || c=='u')
    printf("Vocale\n");
else
    printf("Consonante\n");
```

- In questo frammento di codice andiamo a cammuffare un assegnamento all' interno di un'istruzione che apparentemente esegue un confronto.
  Notare l'assegnamento c = 'a' al posto del confronto c == 'a'.

11 – Abuso del preprocessore

- Il preprocessore C viene utilizzato frequentemente per offuscare un codice sorgente utilizzando delle define che sostituiscano le parole riservate del linguaggio con altre stringhe poco significative.

```
#define k *(int*)
#define a if(
#define c ad()
#define i else
#define p while(
#define x *(char*)
#define b ==
#define V =calloc(1,99999)
#define f ()
#define J return
```

11.1 – Abuso del preprocessore

- Oppure parole riservate del linguaggio vengono sostituite con altre parole riservate per indurre in confusione.

```
#define _                          ;double
#define void                       x,x
#define case(break,default)        break[0]:default[0]:
#define switch(bool)               ;for(;x<bool;
#define do(if,else)                inIine(else)>int##if?
#define true                       (--void++)
#define false                      (++void--)
```

11.2 – Abuso del preprocessore

- In alcuni casi le direttive del preprocessore possono essere indentate esattamente come succede per il resto del codice sorgente al fine di dare vita a immagini piu' o meno complesse.

```
#define O =(n=*(l)V(021),R[4]=E(V(17)+4),n)
#define p(a,b,c) system((sprintf(a,b,k[1]),c)),z
#define                        g (y/010&7)
#define                        R (B+13)
#define          x86           (F*)index\
(ss+V(i                        ),0100)
#define     D(y,n,a,m,i,c      )d+=sprintf( d,y,n,a,m,i,c ),(F*\
      )                        P
      l      B,i,n,a,r,y       ,
      P                        ;
#define    Tr(an,sl,at,or)     l an##i(d,sl){ c at? an##i(d,r):or; } \
l an(d,                        sl){ c \
r=V(014                        )&63,an##i(d,sl); }
#define add(Ev,Gv) Ev(){ i=((a-=16)+C(r,4))/4,(\
Gv?Ev()    :0)   ; } Ev##n(){ a=C(r,5),Ev(); }
```

12 – Operatore XOR

- E' possibile scambiare il valore di due variabili senza usare una terza variabile d'appoggio utilizzando le proprieta' matematiche dell'operatore XOR.

```
int a = 5, b = 10;

a = a ^ b;
b = a ^ b;
a = a ^ b;
```

13 – International Obfuscated C Code Competition

- Si tratta di una competizione nella quale i programmatori di tutto il mondo si sfidano per scrivere il codice sorgente piu' incomprensibile.

- L'idea e' stata ispirata dal codice della Buorne shell scritto da Steve Bourne, che abuso' del preprocessore C per rendere il codice C da lui scritto molto piu' simili al linguaggio Algol-68 con il quale aveva una maggiore familiarita'.

- Come ogni anno Landon Curt Noll e altri ideatori della manifestazione hanno decretato i vincitori dell'edizione 2004.

13.1 – anonymous.c

```
                                  #include\
                                  <stdio.h>
                     #include                <stdlib.h>
                     #include                <string.h>

                    #define w "Hk~HdA=Jk|Jk~LSyL[{M[wMcxNksNss:"
                   #define r"Ht@H|@=HdJHtJHdYHtY:HtFHtF=JDBIl"\
                  "DJTEJDFIlMIlM:HdMHdM=I|KIlMJTOJDOIlWITY:8Y"
                 #define S"IT@I\\@=HdHHtGH|KILJJDIJDH:H|KID"\
                "K=HdQHtPH|TIDRJDRJDQ:JC?JK?=JDRJLRI|UItU:8T"
               #define _(i,j)L[i=2*T[j,O[i=O[j-R[j,T[i=2*\
             R[j-5*T[j+4*O[j-L[j,R[i=3*T[j-R[j-3*O[j+L[j,
            #define t"IS?I\\@=HdGHtGIDJILIJDIItHJTFJDF:8J"

    #define y                  yy(4),yy(5),                yy(6),yy(7)
  #define yy(              i)R[i]=T[i],T[i ]            =O[i],O[i]=L [i]
#define Y _(0          ], 4] )_ (1 ], 5] )_ (2      ], 6] )_ (3 ], 7] )_=1
#define v(i)(      (( R[ i ] * _ + T [ i ]) * _ + O [ i ]) * _ + L [ i ]) *2
double b = 32  ,l ,k ,o ,B ,_ ; int Q , s , V , R [8 ], T[ 8] ,O [8 ], L[ 8] ;
#define q( Q,R ) R= *X ++ % 64 *8 ,R |= *X /8 &7 ,Q=*X++%8,Q=Q*64+*X++%64-256,
# define  p      "G\\QG\\P=GLPGTPGdMGdNGtOGlOG"   "dSGdRGDPGLPG\\LG\\LHtGHtH:"
#  define W         "Hs?H{?=HdGH|FI\\II\\GJlHJ"    "lFL\\DLTCMlAM\\@Ns}Nk|:8G"
# define   U            "EDGEDH=EtCElDH{~H|AJk}"       "Jk?LSzL[|M[wMcxNksNst:"
#  define u                  "Hs?H|@=HdFHtEI"              "\\HI\\FJLHJTD:8H"
char  *   x                   ,*X , ( * i )[               640],z[3]="4_",
*Z = "4,8O4.8O4G" r U "4M"u S"4R"u t"4S8CHdDH|E=HtAIDAIt@IlAJTCJDCIlKI\\K:8K"U
 "4TDdWDdW=D\\UD\\VF\\FFdHGtCGtEIDBIDDIlBIdDJT@JLC:8D"t"4UGDNG\\L=GDJGLKHL\
FHLGHtEHtE:"p"4ZFDTFLT=G|EGlHITBH|DIlDIdE:HtMH|M=JDBJLDKLAKDALDFKtFKdMK\
\\LJTOJ\\NJTMJTM:8M4aGtFGlG=G|HG|H:G\\IG\\J=G|IG|I:GdKGlL=G|JG|J:4b"W
S"4d"W t t"4g"r w"4iGlIGlK=G|JG|J:4kHl@Ht@=HdDHtCHdPH|P:HdDHdD=It\
BIlDJTEJDFIdNI\\N:8N"w"4lID@IL@=HlIH|FHlPH|NHt^H|^:H|MH|N=J\\D\
J\\GK\\OKTOKDXJtXItZI|YIlWI|V:8^4mHLGH\\G=HLVH\\V:4n" u t t
"4p"W"IT@I\\@=HdHHtGIDKILIJLGJLG:JK?JK?=JDGJLGI|MJDL:8M4\
rHt@H|@=HtDH|BJdLJTH:ITEI\\E=ILPILNNtCNlB:8N4t"W t"4u"
p"4zI[?Il@=HlHH|HIDLILIJDII|HKDAJ|A:JtCJtC=JdLJtJL\
THLdFNk|Nc|\
:8K"; main (
int C,char**        A) {for(x=A[1],i=calloc(strlen(x)+2,163840);
C-1;C<3?Q=_=       0,(z[1]=*x++)?((*x++==104?z[1]^=32:--x), X =
strstr(Z,z))      &&(X+=C++):(printf("P2 %d 320 4 ",V=b/2+32),
V*=2,s=Q=0,C     =4):C<4?Q-->0?i[(int)((l+=o)+b)][(int)(k+=B)
]=1:_?_-=.5/    256,o=(v(2)-(l=v(0)))/(Q=16),B=(v(3)-(k=v(1)
))/Q:*X>60?y   ,q(L[4],L[5])q(L[6],L[7])*X-61||(++X,y,y,y),
Y:*X>57?++X,  y,Y:*X >54?++X,b+=*X++%64*4:--C:printf("%d "
,i[Q][s]+i[Q ][s+1]+i[Q+1][s]+i[Q+1][s+1])&&(Q+=2)<V||(Q=
0,s+=2)<640
||(C=1));}
```

13.2 – Deoffuscare

- Solitamente i codici migliori presentano molteplici livello di offuscamento che elencheremo secondo l'ordine in cui dobbiamo procedere a deoffuscare:

  - preprocessore;
  - indentazione;
  - semplificazione costrutti;
  - semplificazione espressioni;
  - algoritmo;

13.3 – Cpp e indent

- Preprocessore
  Procediamo alla sostituzione automatica delle #define utilizzando il preprocessore C:
  - `cpp anonymous.c > out.c`
- Indentazione
  Procediamo all'indentazione del codice sorgente, in un primo momento automatica e successivamente manuale:
  - `indent anonymous.c`

13.4 – Semplificazione costrutti

- Semplificazione costrutti

```
main(int C, char **A)
```

- gli argomenti della main() sono stati rinominati utilizzando i loro nomi canonici:

```
main(int argc, char **argv)
```

13.5 – Semplificazione costrutti

- Il costrutto for principale del programma che si presentava cosi':

```
for (x = A[1], i = calloc(strlen(x) + 2, 163840); C - 1;
     ....
     ....
     resto del codice sorgente);
```

- e' stato riscritto cosi':

```
for (x = argv[1], i = calloc(strlen(x) + 2, 163840);
argc - 1; ){
     ....
     ....
     resto del codice sorgente
}
```

13.6 – Semplificazione costrutti

- Sono stati sostituiti tutti gli operatori ternari:

```
*X > 54 ?
    ++X,
    b += *X++ % 64 * 4
:
    --C
```

- con i relativi if-else:

```
if(*X > 54){
    ++X;
    b += *X++ % 64 * 4;
}
else
    --argc;
```

13.7 – Semplificazione espressioni

- Gli operatori di serializzazione

```
printf("P2 %d 320 4 ", V = b / 2 + 32),V *= 2,s = Q = 0,
C = 4
```

- sono stati eliminati dove non erano necessari:

```
printf("P2 %d 320 4 ", V = b / 2 + 32);
V *= 2;
s = Q = 0;
argc = 4;
```

13.8 – Semplificazione espressioni

- Espressioni complesse:

```
((*x++ == 104 ? z[1] ^= 32 : --x), X = strstr(Z, z)) &&
(X += argc++);
```

- sono state riscritte in una forma piu' semplice:

```
if(*x++ == 104)
    z[1] ^= 32;
else
    --x;
if(X = strstr(Z, z))
    X += argc++;
```

13.9 – Semplificazione espressioni

- L'espressione:

```
*X - 61 || (++X, R[4] = T[4], ..., O[7] = L[7]);
```

- e' stata riscritta in questo modo:

```
if(!(*X - 61)){
    ++X;
    R[4] = T[4];
    ...
    O[7] = L[7];
}
```

13.10 – Semplificazione espressioni

- L'espressione:

```
printf("%d ",i[Q][s] + i[Q][s + 1] + i[Q + 1][s] + i[Q + 1][s +
1]) && (Q += 2) < V || (Q = 0, s += 2) < 640|| (argc = 1);
```

e' stata riscritta in questo modo:

```
printf("%d ",i[Q][s] + i[Q][s+1] + i[Q+1][s] + i[Q+1][s+1]);
if(!((Q += 2) < V)){
      Q = 0;
      if(!((s += 2) < 640))
            argc = 1;
}
```

- Da notare che l'operatore && puo' essere omesso perche' printf() in questo caso ritorna sempre un valore diverso da 0.

14 – gavare.c

```
X=1024; Y=768; A=3;

J=0;K=-10;L=-7;M=1296;N=36;O=255;P=9;_=1<<15;E;S;C;D;F(b){E="1""111886:6:??AAF"
"FHHMMOO55557799@@>>>BBBGGIIKK"[b]-64;C="C@=::C@@==@=:C@=:C@=:C5""31/513/5131/"
"31/531/53"[b ]-64;S=b<22?9:0;D=2;}I(x,Y,X){Y?(X^=Y,X*X>x?(X^=Y):0,  I (x,Y/2,X
)):(E=X);      }H(x){I(x,    _,0);}p;q(        c,x,y,z,k,l,m,a,          b){F(c
);x-=E*M     ;y-=S*M           ;z-=C*M         ;b=x*       x/M+         y*y/M+z
*z/M-D*D    *M;a=-x              *k/M     -y*l/M-z        *m/M;    p=((b=a*a/M-
b)>=0?(I    (b*M,_      ,0),b    =E,      a+(a>b      ?-b:b)):     -1.0);}Z;W;o
(c,x,y,     z,k,l,    m,a){Z=!    c?      -1:Z;c     <44?(q(c,x         ,y,z,k,
l,m,0,0     ),(p>      0&&c!=     a&&        (p<W         ||Z<0)          )?(W=
p,Z=c):     0,o(c+         1,    x,y,z,        k,l,          m,a)):0     ;}Q;T;
U;u;v;w    ;n(e,f,g,            h,i,j,d,a,    b,V){o(0      ,e,f,g,h,i,j,a);d>0
&&Z>=0? (e+=h*W/M,f+=i*W/M,g+=j*W/M,F(Z),u=e-E*M,v=f-S*M,w=g-C*M,b=(-2*u-2*v+w)
/3,H(u*u+v*v+w*w),b/=D,b*=b,b*=200,b/=(M*M),V=Z,E!=0?(u=-u*M/E,v=-v*M/E,w=-w*M/
E):0,E=(h*u+i*v+j*w)/M,h-=u*E/(M/2),i-=v*E/(M/2),j-=w*E/(M/2),n(e,f,g,h,i,j,d-1
,Z,0,0),Q/=2,T/=2,       U/=2,V=V<22?7:  (V<30?1:(V<38?2:(V<44?4:(V==44?6:3))))
,Q+=V&1?b:0,T                +=V&2?b        :0,U+=V    &4?b:0)     :(d==P?(g+=2
,j=g>0?g/8:g/     20):0,j    >0?(U=     j    *j/M,Q      =255-    250*U/M,T=255
-150*U/M,U=255    -100    *U/M):(U    =j*j     /M,U<M           /5?(Q=255-210*U
/M,T=255-435*U           /M,U=255    -720*      U/M):(U       -=M/5,Q=213-110*U
/M,T=168-113*U    /       M,U=111               -85*U/M)      ),d!=P?(Q/=2,T/=2
,U/=2):0);Q=Q<    0?0:      Q>O?     O:          Q;T=T<0?    0:T>O?O:T;U=U<0?0:
U>O?O:U;}R;G;B    ;t(x,y     ,a,    b){n(M*J+M    *40*(A*x   +a)/X/A-M*20,M*K,M
*L-M*30*(A*y+b)/Y/A+M*15,0,M,0,P,  -1,0,0);R+=Q   ;G+=T;B   +=U;++a<A?t(x,y,a,
b):(++b<A?t(x,y,0,b):0);}r(x,y){R=G=B=0;t(x,y,0,0);x<X?(printf("%c%c%c",R/A/A,G
/A/A,B/A/A),r(x+1,y)):0;}s(y){r(0,--y?s(y),y:y);}main(){printf("P6\n%i %i\n255"
"\n",X,Y);s(Y);}
```

14.1 – Cpp e indent

- Preprocessore

  In questo caso non e' necessario dare in pasto il codice a cpp visto che il codice non presenta alcuna direttiva del preprocessore C.

- Indentazione

  Procediamo all'indentazione del codice sorgente, in un primo momento automatica e successivamente manuale:

  - `indent gavare.c`

14.2 – Semplificazione costrutti

- Una volta ottenuto un codice piu' pulito possiamo iniziare a semplificare alcune espressioni sostituendo gli operatori ternari con gli equivalenti costrutti if-else.

```
Z = !c ? -1 : Z;
```

- Puo' essere riscritto cosi'

```
if(!c)
    Z=-1;
```

- L'else puo' essere omesso visto che esegue come assegnamento Z = Z che non altera il valore della variabile.

14.3 – Semplificazione costrutti

- In questo caso l'operatore ternario viene utilizzato all'interno di una chiamata a funzione.

```
r(0, --y ? s(y), y : y);
```

- Se --y e' diverso da zero viene invocata la funzione s() con argomento y.

- La funzione r() viene chiamata con y come secondo argomento visto che l'operatore virgola ritorna il valore dell'ultima espressione valutata.

14.4 – Semplificazione costrutti

- Se --y invece e' uguale a zero la funzione s() non viene invocata e viene chiamata la funzione r() con y come secondo argomento.

- Possiamo riscrivere il frammento di codice in questo modo:

```
if(--y)
    s(y);
r(0, y);
```

14.5 – Semplificazione costrutti

- L'operatore di assegnamento ha una precedenza inferiore rispetto all'operatore ternario

```
S = b < 22 ? 9 : 0;
```

- Pertanto il valore ritornato da quest'ultimo viene assegnato alla variabile S, possiamo riscrivere il codice in questo modo:

```
if(b < 22)
    S=9;
else
    S=0;
```

14.6 – Semplificazione costrutti

- Il seguente frammento di codice utilizza l'operatore ternario insieme all'operatore di serializzazione.

```
Y ?
      X ^= Y, X * X > x ?
               (X ^= Y)
      :
               0, I(x, Y / 2, X)
:
      (E = X);
```

- Possiamo riscriverlo cosi'

```
if(Y){
      X ^= Y;
      if(X * X > x)
               X ^= Y;
      I(x, Y / 2, X);
}
else
      E = X;
```

14.7 – Semplificazione costrutti

- Infine un caso in cui compaiono contemporaneamente operatore di assegnamento, ternario e di serializzazione.

```
p = ((b = a * a / M - b) >= 0 ?
      (I(b * M, _, 0), b = E, a + (a > b ? -b : b))
:
      -1.0);
```

- Alla variabile p viene assegnato il valore dell'ultima espressione valutata dall'operatore virgola. Pertanto il codice puo' essere riscritto cosi':

```
if((b = a * a / M - b) >= 0){
      I(b * M, _, 0);
      b = E;
      p = a + (a > b ? -b : b);
}
else
      p = -1.0;
```

15 – Codice autoreplicante

- Per codice autoreplicante si intende un programma che una volta compilato ed eseguito produca come output una copia del suo codice sorgente.

- Il programma deve essere in grado di svolgere tale compito senza appoggiarsi al file sorgente da cui e' stato generato il binario.

- Questo tipo di programmi prendono il nome di quine dal nome del matematico Willard van Orman Quine.

- Alcuni programmatori si cimentano nella scrittura della quine piu' corta possibile, infatti, la vera difficolta' e' quella di riuscire a scrivere un programma del genere usando il minor numero di caratteri possibile.

15.1 – Codice autoreplicante

- Quello che segue potrebbe essere un tentativo da parte di un programmatore maldestro di scrivere un simile programma.

```
int main(){
    printf("int main(){\nprintf(\"??????????
```

```
char *p="char *p=\"?????????\";\nint main(){\n\tprintf(p);\n}";
int main(){
    printf(p);
}
```

- Si troverebbe ben presto difronte al paradosso di un codice ricorsivo.

15.2 – Codice autoreplicante

- Anche al piu' maldestro dei programmatori dopo un paio di tentativi falliti risulterebbe evidente che non e' possibile scrivere il codice sorgente del programma all'interno del sorgente stesso perche' tale operazione risulterebbe in una ricorsione infinita.

- Quello che dobbiamo fare e' sfruttare le proprieta' delle stringhe di formato delle funzioni della famiglia printf.

15.3 – Codice autoreplicante

- Utilizzando la stessa stringa sia come stringa di formato che come argomento della printf possiamo aggirare il problema della ricorsivita'.

```
$ cat quine.c
char *p="char *p=\"%s\";int main(){printf(p,p);}";
int main(){printf(p,p);}

$ gcc quine.c -o quine
$ ./quine
char *p="char *p="%s";int main(){printf(p,p);}";
int main(){printf(p,p);}
```

- Siamo sulla buona strada, ma c'e' ancora un problema da risolvere prima di poter cantare vittoria.

15.4 – Codice autoreplicante

- I caratteri backslash che fungevano da quote per i caratteri virgolette sono spariti perche' vengono interpretati come quote e quindi eliminati se si trovano all'interno di una stringa.

- Per risolvere questo problema dobbiamo omettere all'interno della stringa tutti quei caratteri che vengono interpretati in maniera particolare sostituendoli con qualcosa di analogo.

15.5 – Codice autoreplicante

- In questo caso le virgolette posso essere sostituite con il codice ascii del carattere.

```
$ cat quine.c
char *p="char *p=%c%s%c;main(){printf(p,34,p,34);}";
main(){printf(p,34,p,34);}

$ gcc quine.c -o quine
$ ./quine
char *p="char *p=%c%s%c;main(){printf(p,34,p,34);}";
main(){printf(p,34,p,34);}
```

15.6 – Codice autoreplicante

- La printf prende come suo primo argomento la stringa di formato.
- Si tratta di una normale stringa, solo che i caratteri % seguiti da un altro carattere vengono interpretati dalla printf come specificatori di conversione.
  - `%d` intero
  - `%c` carattere
  - `%s` stringa
- Per ogni specificatore di conversione contenuto nella stringa di formato dovra' corrispondere un ulteriore argomento della printf.

15.7 – Codice autoreplicante

- Nel nostro caso la chiamata a printf appare in questo modo:

```
printf(p,34,p,34);
```

- Il primo argomento e' la stringa di formato nella quale compaiono tre specificatori di conversione:

```
"char *p=%c%s%c;main(){printf(p,34,p,34);}"
```

- Ad ognuno di essi corrisponde nell'ordine un argomento della printf.
  - `%c`   `34`
  - `%s`   `p`
  - `%c`   `34`

15.8 – Codice autoreplicante

- Il primo specificatore di conversione viene espanso con il carattere ", il cui codice ascii e' appunto 34.
- Il secondo viene espanso nella stringa puntata da p, questa volta viene interpretata come nosrmale stringa e non di formato pertanto gli specificatori %c e %s non vengono espansi in alcun modo.
- Il terzo viene espanso esattamente come il primo.
  - `%c` `"`
  - `%s` `char *p=%c%s%c;main(){printf(p,34,p,34);}`
  - `%s` `"`

15.9 – Codice autoreplicante

- Una volta compreso il meccanismo possiamo scrivere quine piu' o meno complesse avendo l'accortezza di sostituire tutti i caratteri che vengono interpretati in maniera speciale all'interno di una stringa con i relativi codici ascii del carattere.

- Tra questi caratteri compaiono:

  - virgolette;
  - backslash;
  - percentuale;

16 – Codice automodificante

- Oltre ai gia' complessi programmi autoreplicanti esiste la variante dei programmi automodificanti.
- Quest'ultimi una volta eseguiti producono come output una versione modificata e funzionanete del proprio codice sorgente.
- Il meccanismo e' del tutto simile a quello illustrato in precedenza, tuttavia il programma deve essere in grado di modificarsi secondo determinate regole che non ne compromettano la compilazione e il corretto funzionamento.

16.1 – Codice automodificante

- Iniziamo analizzando il funzionamento del programma automodificante da un punto di vista dell'algoritmo.

```
main(){
char *a="main(){char *a=%c%s%c;....printf(a,34,b,34);}";
....
....
printf(a,34,b,34);
}
```

Al fine di capire la logica di tale programma non e' necessario entrare nei dettagli del codice pertanto alcune porzioni di sorgente sono state volutamente omesse per semplicita'.

16.2 – Codice automodificante

- La variabile "a" e' una stringa che contiene l'intero programma.
- Durante l'esecuzione tale stringa viene parsata alla ricerca della stringa printf relativa alla chiamata di tale funzione.
- La chiamata di tale funzione viene modificata aggiungendo come ulteriore argomento il codice ascii del primo carattere di tale stringa che non faccia parte di uno specificatore di formato (es. %d, %c, %s).

16.3 – Codice automodificante

- La variabile "b" e' una copia della stringa "a".
- Durante l'esecuzione tale stringa viene modificata secondo le stesse modalita' illustrate per la stringa "a";
- Inoltre il carattere il cui codice ascii e' stato inserito come argomento della printf viene sostituito all'interno della stringa con lo specificatore di conversione %c.

16.4 – Codice automodificante

- Ecco come si presenta il codice automodificante dopo la prima esecuzione:

```
$ cat quine.c
main(){
char *a="main(){char *a=%c%s%c;....printf(a,34,b,34);}";
....
....
printf(a,34,b,34);
}

$ gcc quine.c -o quine
$ ./quine
main(){
char *a="%cain(){char *a=%c%s%c;....printf(a,109,34,b,34);}";
....
....
printf(a,109,34,b,34);
}
```

16.5 – Codice automodificante

- Il codice cosi' ottenuto puo' essere ricompilato ed eseguito piu' volte ed esso si automodifichera' ogni volta in modo coerente.

17 – Codice umoristico

- Durante lo sviluppo dello standard ANSI C venne introdotta la direttiva #pragma del preprocessore. Tale direttiva viene ereditata dal linguaggio Ada risultando come una novita' per il C a tal punto da incontrare alcune resistenze da parte degli sviluppatori di gcc.

- Lo standard ANSI definisce la direttiva #pragma dicendo che essa ha un effetto definito a seconda dell'implementazione.

- Gli sviluppatori di gcc interpretano il termine "implementation-defined" in maniera ironica tanto che nella versione 1.34 del compilatore la direttiva #pragma causa il termine della compilazione e l'avvio di un gioco :)

17.1 – Codice umoristico

- Nel manuale del gcc 1.34 possiamo leggere:

```
The "#pragma" command is specified in the ANSI standard
to have an arbitrary implementation-defined effect. In
the GNU C preprocessor, "#pragma" first attempts to run
the game "rogue"; if that fails, it tries to run the
game "hack"; if that fails, it tries to run GNU Emacs
displaying the Tower of Hanoi; if that fails, it reports
a fatal error. In any case, preprocessing does not
continue.
```

17.2 – Codice umoristico

```
/*
 * the behavior of the #pragma directive is implementation defined.
 * this implementation defines it as follows.
 */
do_pragma ()
{
 close (0);
 if (open ("/dev/tty", O_RDONLY, 0666) != 0)
                     goto nope;
 close (1);
 if (open ("/dev/tty", O_WRONLY, 0666) != 1)
                     goto nope;
 execl ("/usr/games/hack", "#pragma", 0);
 execl ("/usr/games/rogue", "#pragma", 0);
 execl ("/usr/new/emacs", "-f", "hanoi", "9", "-kill", 0);
 execl ("/usr/local/emacs", "-f", "hanoi", "9", "-kill", 0);
nope:
 fatal ("You are in a maze of twisty compiler features, all different");
}
```

18 – Risorse

- Le slide e il codice sorgente degli esempi possono essere scaricati dal sito:
  - www.autistici.org/eazy
- Alcuni libri interessanti da cui prendere spunti:
  - Expert C Programming – Peter Van Der Linden
  - C Traps and Pitfalls – Andrew Koenig
- Siti interessanti:
  - www.ioccc.org