



Master.Mind

In Linguaggio C

ABSTRACT

Progetto per lo sviluppo di un programma in linguaggio C del gioco Mastermind, giocabile da console, necessario al superamento dell'esame di Programmazione del corso di laurea SSRI di UniMi.

Emanuele Piras

matricola: 000001

1 Tavola dei Contenuti

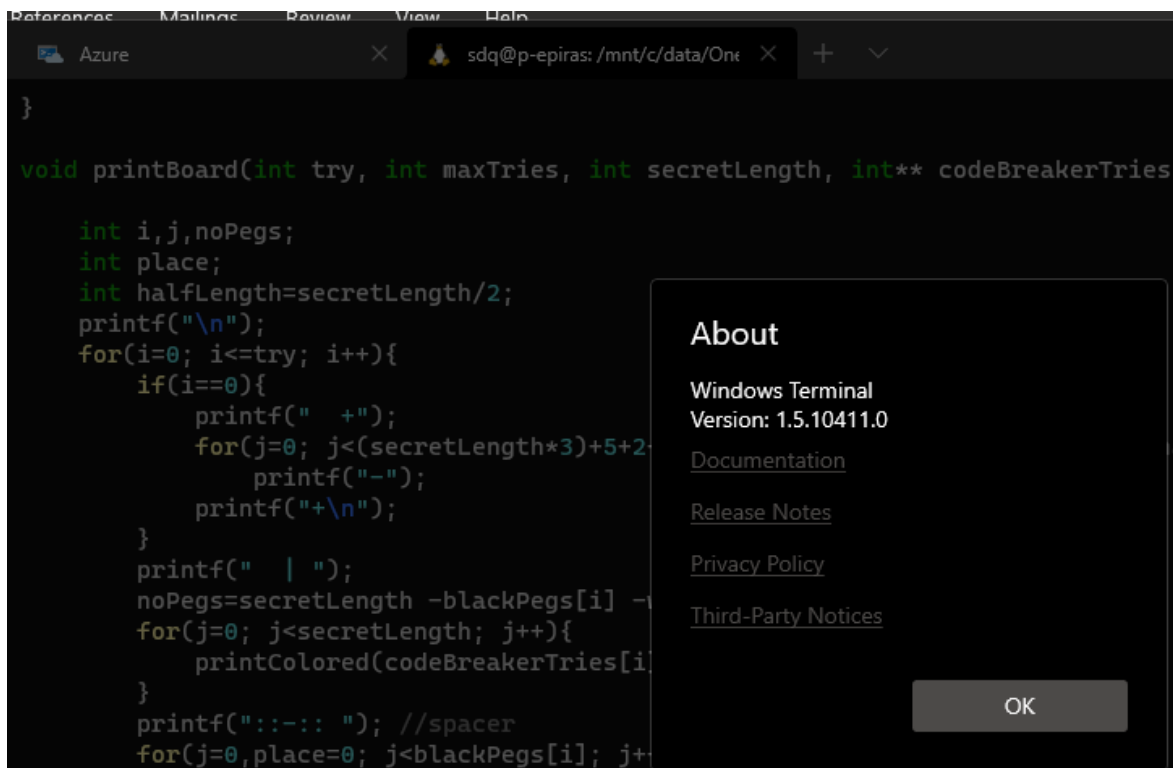
1	Tavola dei Contenuti	2
2	Scopo.....	3
3	Premessa	3
4	Specifiche	4
5	Progettazione	5
5.1	Menu principale	5
5.2	Interfaccia	6
5.3	Struttura dati.....	7
5.4	Funzioni di supporto	7
5.5	Funzioni intermedie	7
5.6	Programma principale.....	7
5.7	Modalità debug.....	7
5.8	Struttura dei file	8
5.8.1	Makefile	8
6	Modellazione	9
6.1	Flow-chart	9
6.2	Pseudocodice	11
7	Codifica.....	12
7.1	Sistema di selezione dei menu.....	12
7.2	Struttura dati.....	12
7.3	Schema della struttura dati.....	14
7.4	Funzioni di utilità.....	15
7.5	Funzioni intermedie	15
7.6	Programma principale.....	16
8	Test & debug	17
8.1	Verifica	17
8.2	Correzione.....	18

2 Scopo

Il progetto prevede lo sviluppo di un programma in linguaggio C del gioco Mastermind, giocabile da console, necessario al superamento dell'esame di Programmazione del corso di laurea SSRI di UniMi.

3 Premessa

Il programma è stato sviluppato su un *PC Windows 10*, utilizzando **Windows Terminal** in ambiente **Windows Subsystem for Linux (Ubuntu 20.04.x LTS)**



The screenshot shows a Windows Terminal window with a dark theme. The terminal displays C code for a function named `printBoard`. The code includes variables for `try`, `maxTries`, `secretLength`, and `codeBreakerTries`. It uses `printf` for output and `printColored` for colored text. An `About` dialog box is overlaid on the right side of the terminal. The dialog box has a title bar and contains the following text:

```
}

void printBoard(int try, int maxTries, int secretLength, int** codeBreakerTries

    int i,j,noPegs;
    int place;
    int halfLength=secretLength/2;
    printf("\n");
    for(i=0; i<=try; i++){
        if(i==0){
            printf("  +");
            for(j=0; j<(secretLength*3)+5+2; j++){
                printf("-");
            }
            printf("+\n");
        }
        printf(" | ");
        noPegs=secretLength -blackPegs[i] -whitePegs[i];
        for(j=0; j<secretLength; j++){
            printColored(codeBreakerTries[i][j], "red", "white");
        }
        printf("::--:: "); //spacer
        for(j=0,place=0; j<blackPegs[i]; j++){
            printf("black ");
        }
        for(j=0,place=0; j<whitePegs[i]; j++){
            printf("white ");
        }
    }
}
```

About
Windows Terminal
Version: 1.5.10411.0
[Documentation](#)
[Release Notes](#)
[Privacy Policy](#)
[Third-Party Notices](#)
OK

Trattandosi della mia prima esperienza nella programmazione, il progetto ha avuto un percorso piuttosto accidentato.

Ho iniziato a lavorare con una struttura dati di matrici, vettori e variabili, che si è complicata con l'avanzare del progetto, fino a rendermi conto che sarebbe stato preferibile l'utilizzo di una *struct* contenente gli elementi descritti sopra. Questo intervento è stato oggetto di una delle revisioni principali, nel passaggio dalla major release v0.x alla v1.x.

Nella release v1.2 riorganizzato il codice in header file per clausole *#define* e definizione delle funzioni, e file .c dedicati funzioni di supporto e strutture dati, snellendo il file principale *mastermind.c* e orientando maggiormente la *main function* al flusso logico del programma.

Ho preferito far uso di strutture dinamiche, soprattutto a fini didattici, pur potendo lavorare direttamente con i valori massimi scelti per le opzioni di gioco, viste le ridotte dimensioni delle strutture dati.

4 Specifiche

Il committente richiede un programma per computer, per sistema operativo non specificato, che riproduca il gioco Mastermind, con le seguenti caratteristiche:

- giocabile da console
- che gestisca i codici con ripetizioni di colori/numeri
- che consenta su richiesta di visualizzare il codice segreto per la verifica funzionale
- scritto in linguaggio C, nel rispetto delle regole della programmazione strutturata

Per migliorare il prodotto ho introdotto:

- un'interfaccia utente più completa rispetto al semplice *command prompt*, con colori e qualche animazione
- un menu di istruzioni con le regole del gioco

- un menu impostazioni, per variare la lunghezza del codice segreto, il numero e l'opzione di ripetizione dei colori, a partire dai quali il programma adegua autonomamente il numero di tentativi. Per maggiore usabilità, il menu *Settings* consente di confermare con un semplice invio quei valori che non si vogliono cambiare.
- la riproduzione del tavolo da gioco, che si adatta ai parametri scelti dall'utente
- la possibilità di terminare anticipatamente la partita in corso, tornando al menu principale, senza interrompere l'intero programma con *Ctrl+c*
- una gestione dell'input utente con un buon grado di tolleranza per gli errori di digitazione e limitazione dei messaggi di avviso ai casi significativi, a vantaggio dell'usabilità.

5 Progettazione

Ho iniziato l'elaborazione del programma con un approccio misto *Top-Down / Bottom-Up*, scomponendo il problema iniziale in problemi più semplici e, contemporaneamente, individuando delle funzionalità di base richieste dagli algoritmi e utilizzate ripetutamente, facilmente implementabili, che ho isolato in blocchi di codice, ad esempio per la stampa della legenda. Per la naming convention ho fatto uso di *camel case* e *hungarian notation*.

5.1 Menu principale

con relativi sottomenu, che consente di scegliere tra

- Giocare una partita
- Cambiare le impostazioni di gioco per
 - Lunghezza del codice segreto
 - Numero di colori
 - Numero di tentativi
 - Ripetizioni ammesse o meno
- Stampare le istruzioni di gioco
- Uscire dal gioco

5.2 Interfaccia

per l'interazione con l'utente, che deve

- Ricevere istruzioni sulle azioni utente necessarie
- Inserire delle digitazioni per attuare le proprie decisioni
- Ricevere un riscontro alle proprie azioni
- Rappresentare lo stato della partita
- Comunicare l'esito della partita

Per avere una buona usabilità e leggibilità anche a seguito di eventuali messaggi d'errore e per consentire di avere tutte le giocate precedenti sempre a disposizione, ho scelto di stampare l'intero stato della partita dopo ogni tentativo dell'utente. Questo è visualizzato tramite un'interfaccia che richiama l'aspetto del gioco da tavolo e adatta dinamicamente le proprie dimensioni ai parametri selezionati a *run-time* dall'utente.

```
MasterMind -> PLAY

Secret code length      [ 4]
Number of colors        [ 6]
Number of attempts     [10]
Duplicated colors       [YES]
○      right color and position
●      right color in wrong position
×      missing color
Q      to quit the match
Colors  1 2 3 4 5 6

+-----+
| 1 6 5 2 ::-:: ● ● |
9 | ● ● ● ● ::-:: ● × |
+-----+
| 2 3 6 4 ::-:: ○ ○ |
8 | ● ● ● ● ::-:: × × |
+-----+

Guess the secret code:
```

L'interfaccia è popolata con pioli e numeri colorati da un lato, e con pioli neri, bianchi e la segnalazione di quelli mancanti, dall'altro.

Ho preferito adottare la doppia segnalazione di pioli e numeri, rispetto ai numeri *unicode* in negativo, per la maggiore leggibilità.

La soluzione ha richiesto una struttura dati adeguata.

5.3 Struttura dati

Ho pensato ad una struttura dati che potesse contenere l'intero stato del gioco in ogni istante della partita, in modo da poterla disaccoppiare dalle funzioni che ne fanno uso, per interfacciarmi più comodamente.

La struttura è dinamica, per motivi didattici e per correttezza concettuale, anche se le dimensioni richieste sono contenute e avrebbe potuto essere facilmente statica.

L'accesso alla struttura dati sarà sempre per indirizzo.

5.4 Funzioni di supporto

Una serie di funzioni al servizio delle componenti più complesse del programma, a vantaggio della leggibilità del flusso. Vanno dall'allocazione e de-allocazione della struttura dati, all'acquisizione dell'input, fino alla stampa dei messaggi.

5.5 Funzioni intermedie

Si occupano di compiti più complessi, come il conteggio dei pioli, la stampa dello stato della partita o la gestione degli eventi della partita.

5.6 Programma principale

Contenente poco codice, implementa poche funzioni essenziali del menu principale, come l'uscita dal programma, l'help e lo *switch* di *debug*.

5.7 Modalità debug

Per consentire al committente di visualizzare il codice segreto per le verifiche funzionali, senza richiedere modifiche al codice rispetto alla normale modalità utente, che deve tenere il codice nascosto, ho previsto uno *switch* con cui lanciare il programma in modalità *debug*, preferita ad una normale impostazione che avrebbe confuso l'interfaccia utente. Durante le fasi di gioco, un *prompt* dedicato rende facilmente riconoscibile la modalità *debug*.

5.8 Struttura dei file

Il progetto è organizzato in

- *mastermind.c*
che contiene pochi include e la *main function*
- *mastermind.h*
file condiviso che contiene clausole di *define*, prototipi di funzioni, definizioni di *struct* e *type*.
- *mastermind_data_struct.c*
che contiene le definizioni delle funzioni necessarie alla gestione della struttura dati
- *mastermind_functions.c*
che contiene la maggior parte del codice con la definizione di tutte le restanti funzioni.

Un *Makefile* semplifica la compilazione ed un *README.txt* contiene le istruzioni essenziali per la compilazione e l'esecuzione del programma.

5.8.1 Makefile

```
CC      := gcc
CCFLAGS := -Wall -Wextra -Werror

TARGET:= mastermind
MAINS  := $(addsuffix .o, $(TARGET) )
SRCS   := $(wildcard *.c)
OBJ     := $(SRCS:%.c=%.o)
DEPS    := $(wildcard *.h)

.PHONY: all clean

all: $(TARGET)

clean:
    rm -f $(TARGET) $(OBJ)

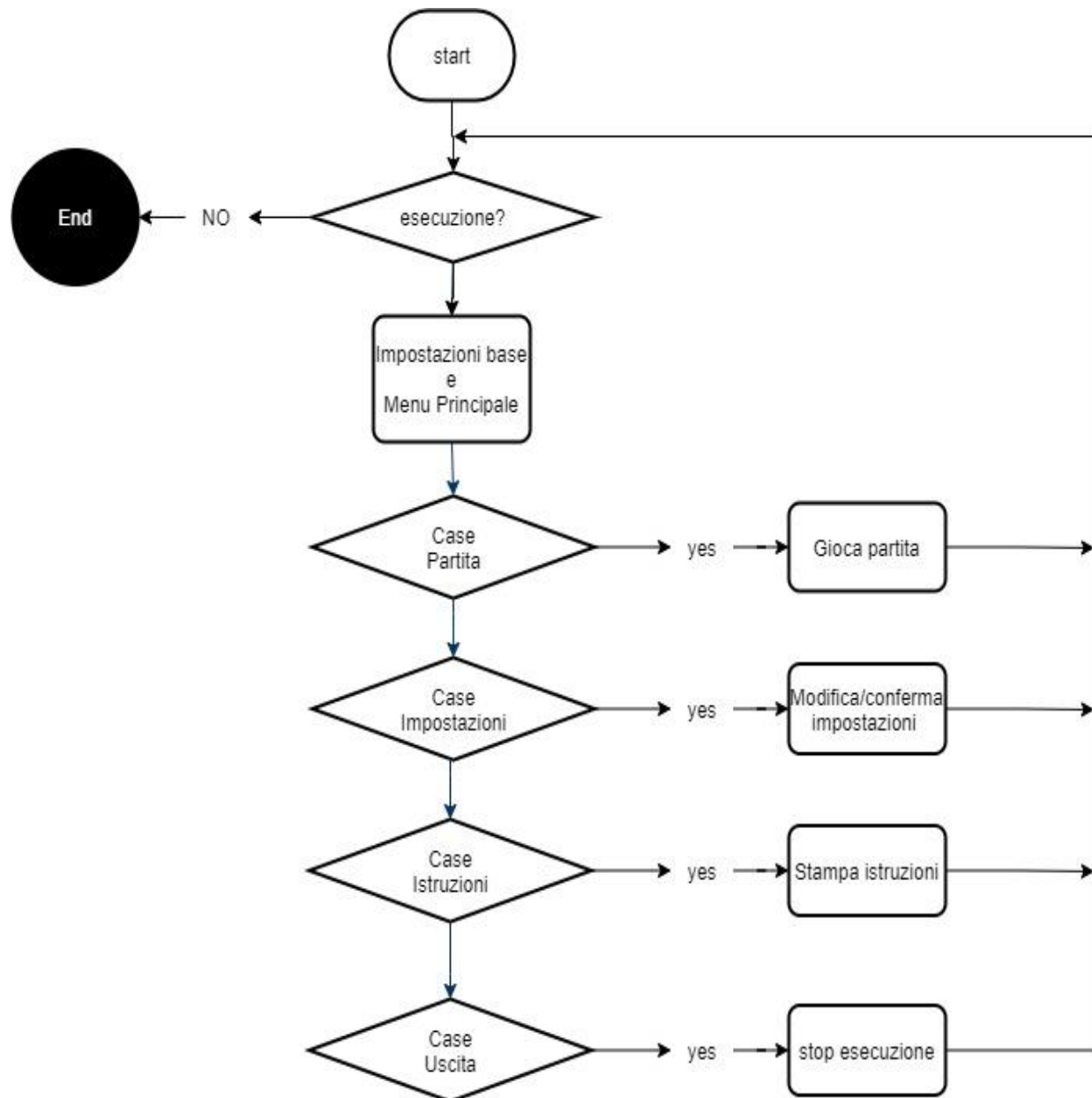
$(OBJ): %.o : %.c $(DEPS)
    $(CC) -c -o $@ $< $(CCFLAGS)

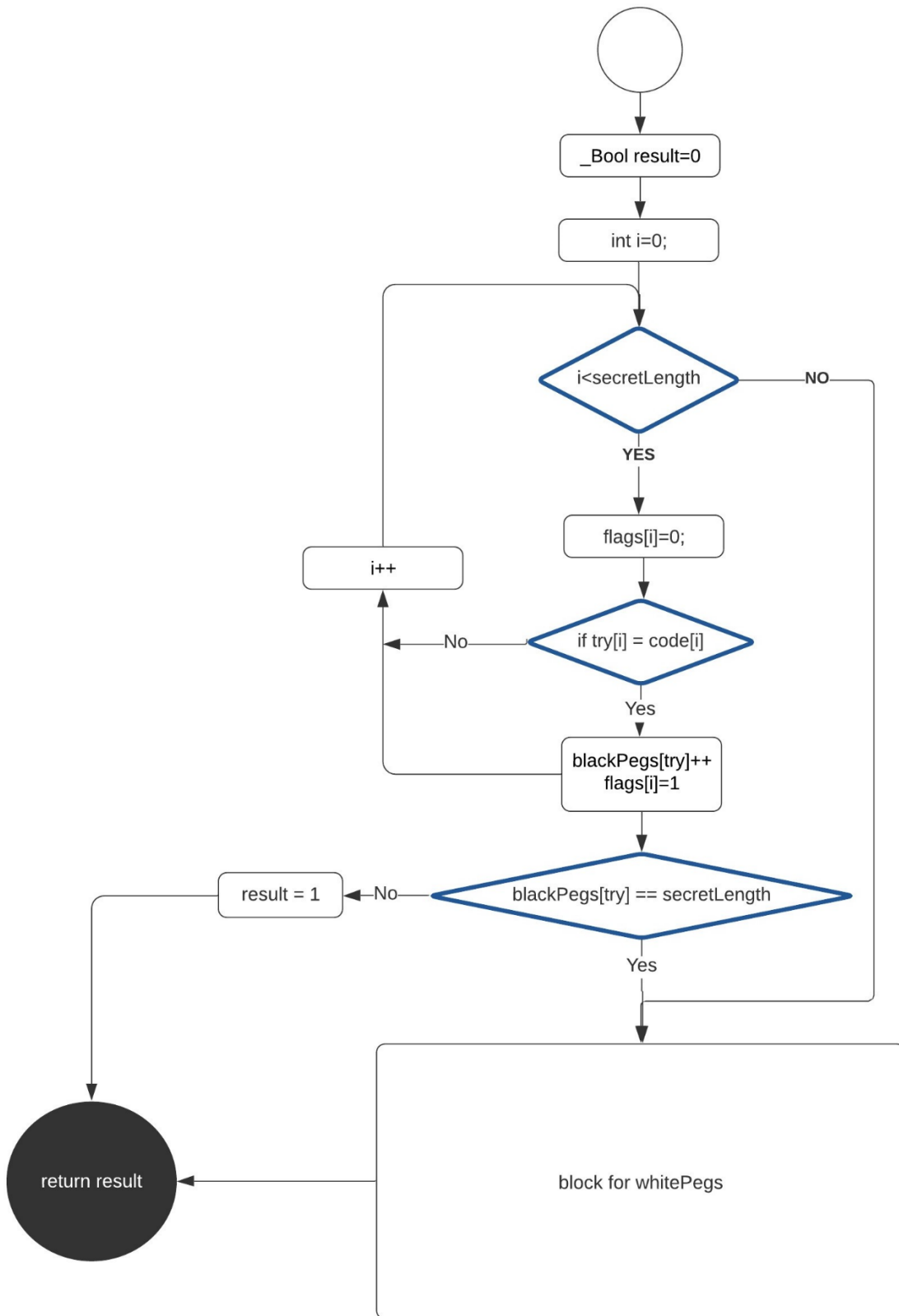
$(TARGET): % : $(filter-out $(MAINS), $(OBJ)) %.o
    $(CC) -o $@ $^ $(CCFLAGS)
```

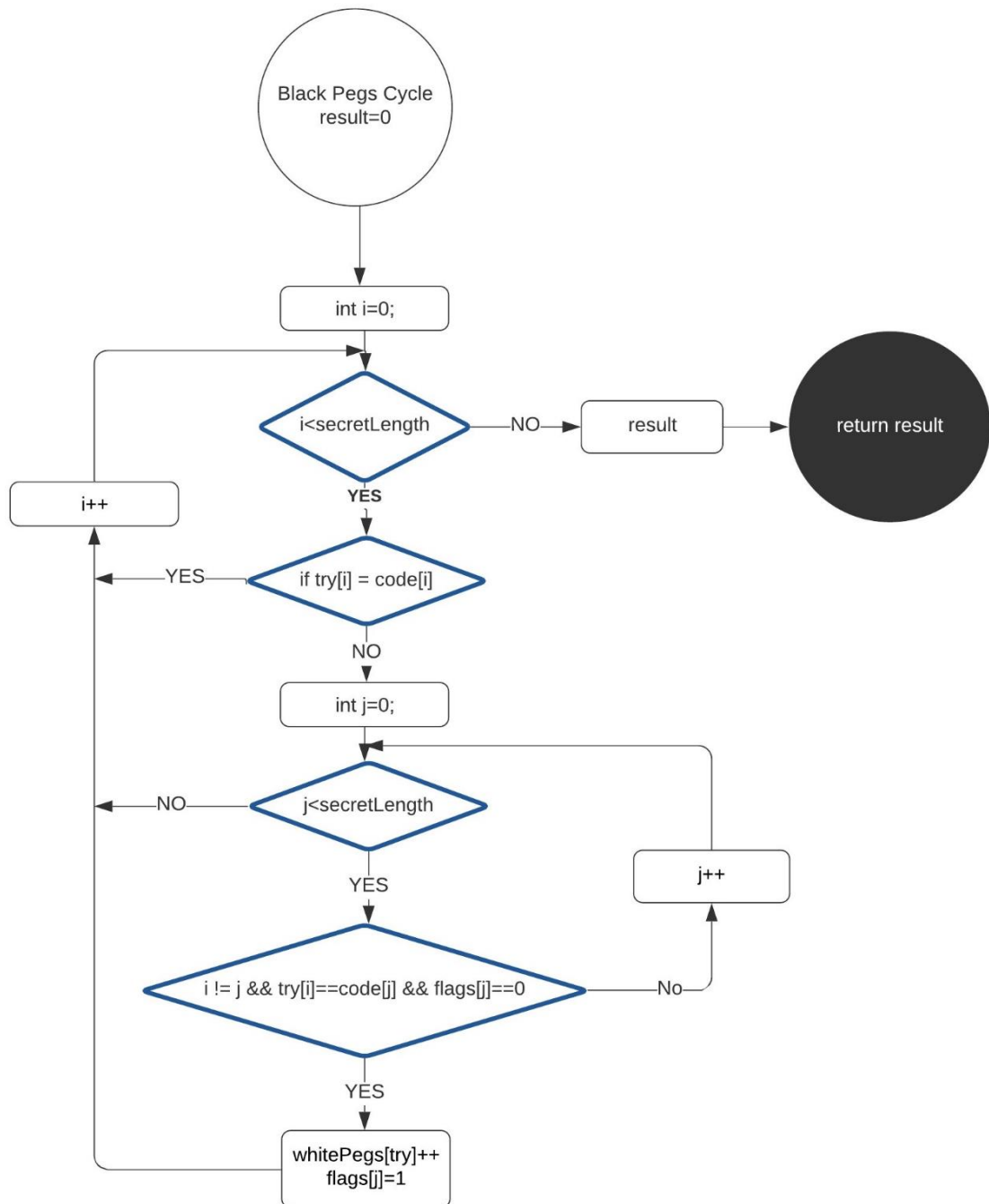

6 Modellazione

6.1 Flow-chart

Ho utilizzato i *flow-chart* a partire dalla struttura più generale del programma, per poi descrivere gli algoritmi complessi, riservando lo *pseudocodice* per definire alcuni dettagli più delicati.







6.2 Pseudocode

Parte del blocco dell'algoritmo di verifica dei pioli bianchi:

```

If ( tryElement=codeElement AND CodeElementFlag=false ) {
    CodeElementFlag <- true;
    increment WhitePeg counter;
    exitLoop; }
  
```

7 Codifica

7.1 Sistema di selezione dei menu

Consiste in un costrutto *switch case* che prende in input la selezione dell'utente e si sviluppa nelle rispettive funzionalità descritte nei requisiti (gioco, impostazioni, istruzioni, uscita).

```
MasterMind

1 - Play
2 - Settings
3 - Instructions
4 - Quit

Make your choice:
```

7.2 Struttura dati

Vi sono memorizzati i dati della partita in corso e una struttura di supporto.

<pre>struct SGame { _Bool bDups; _Bool* prgcCountFlags; int iSecretLength; int iColorsNum; int iMaxTries; int* prgiBlackPegs; int* prgiWhitePegs; char* prgcCode; char** prgcCodeBreakerTries; };</pre>	<ul style="list-style-type: none">- elementi duplicati nel codice- struttura di supporto per verifica pioli- lunghezza del codice segreto- numero di colori nel codice- massimo numero di tentativi- pioli neri per maxTries tentativi- pioli bianchi per maxTries tentativi- codice segreto- matrice tentativi utente
---	--

sulla quale è definito un tipo *SGame*.

La struttura dati dinamica a supporto degli algoritmi, si adatta alle impostazioni scelte dall'utente a *run-time*, ed utilizza:

- **malloc**, quando gli algoritmi che utilizzano quella struttura devono comunque inizializzarne gli elementi per un corretto funzionamento.
- **calloc**, quando gli elementi possono essere acceduti in lettura per verificare che il valore iniziale (0) degli elementi non sia stato modificato, anche prima di un'eventuale scrittura.

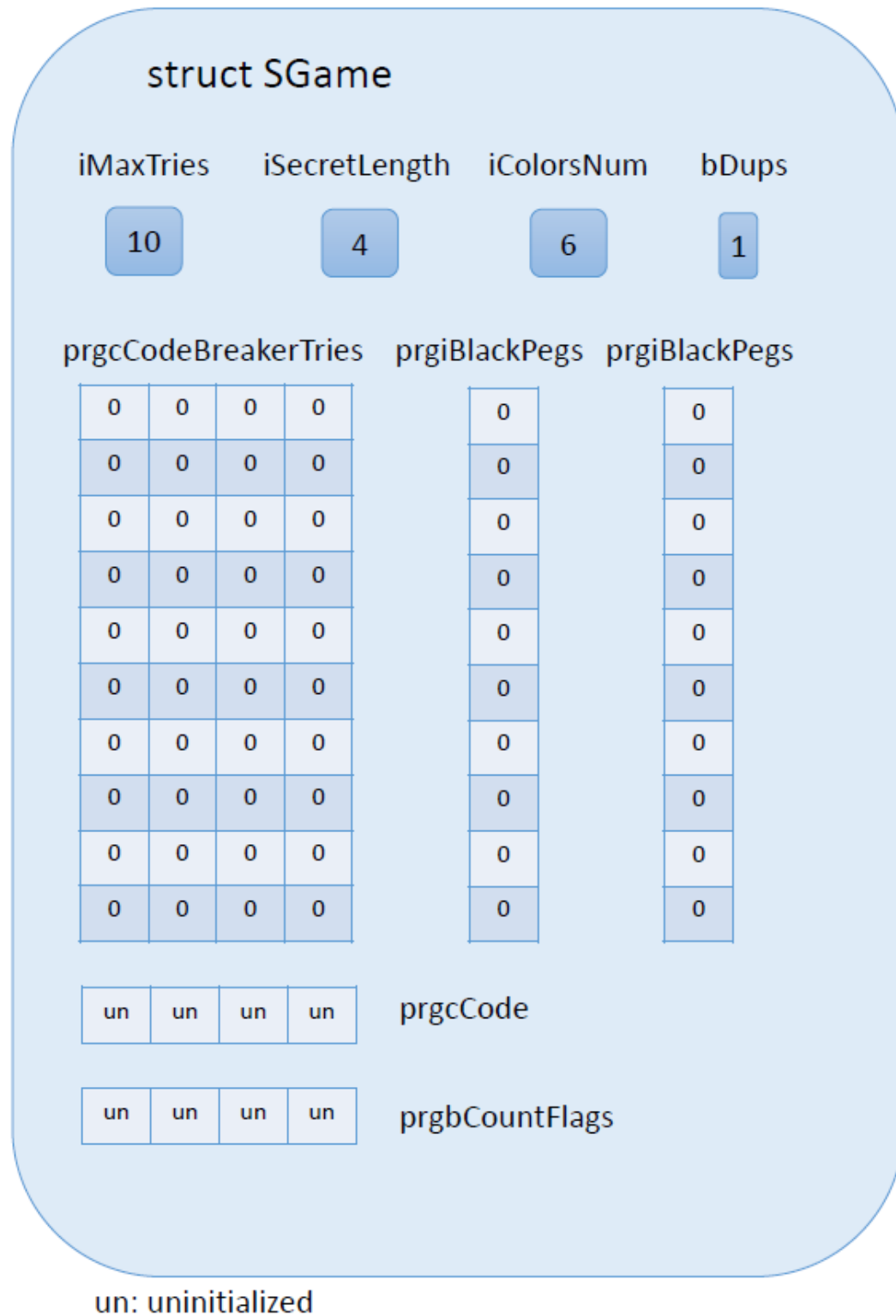
Elementi della struct SGame:

- *iSecretLength*: variabile *int* che stabilisce la massima lunghezza del codice segreto, tra 3 e 5, con default a 4.
- *iColorsNum*: variabile *int* che stabilisce il numero dei colori, tra 6 e 8, con default a 6.
- *iMaxTries*: variabile *int* che stabilisce il massimo numero di tentativi, tra 7 e 18, con default a 10, calcolata dal sistema in funzione della lunghezza del codice segreto e del numero dei colori.
- *prgcCode*: array di *char*, di lunghezza variabile pari a *iSecretLength*
- *prgcCodeBreakerTries*[][]: matrice di n righe e m colonne, dove n e m variano in funzione della lunghezza di *iSecretLength* e m in funzione di *iMaxTries*
- *prgiBlackPegs*[]): array di *int* di lunghezza variabile pari a *iMaxTries*
- *prgiBwhitePegs*[]): array di *int* di lunghezza variabile pari a *iMaxTries*
- *prgbCountFlags*[]): array di *_Bool* di lunghezza variabile pari a *iSecretLength*
- *bDups*: variabile *_Bool* per gestire l'opzione dei colori duplicati

Ho trovato più efficiente de-allocare la struttura dati al termine della partita, per inizializzarla nuovamente in caso di una nuova partita, piuttosto che gestirne un'eventuale modifica a seguito di un cambiamento delle impostazioni.

7.3 Schema della struttura dati

inizio gioco, con impostazioni di default:



7.4 Funzioni di utilità

per le operazioni semplici e ricorrenti, come la stampa di messaggi utente o la composizione dei diversi menu.

Per queste ho utilizzato delle funzioni (es. *celebrateGoodTimes*) e delle clausole *define*, ad esempio per gestire i colori e i caratteri *unicode*, attraverso i quali ho cercato di mettere in risalto le informazioni più significative per l'utente.

7.5 Funzioni intermedie

implementano compiti più articolati, come ad esempio

- *genSecret*: generazione del codice
Genera un array di char compresi tra '6' e il massimo numero di colori scelto, gestendo la possibilità di duplicati o meno. L'implementazione utilizza la funzione *rand()*, che genera un numero pseudo-randomico, che diventa randomico con un seme generato tramite una chiamata *srand(time(NULL))*, a partire dal tempo all'istante della chiamata.
- *getString*: acquisizione stringa per *getChoice* e *getTry*
la funzione acquisisce una stringa da standard input e la memorizza come codice ASCII in un array di *char* all'interno della *struct*. La funzione consente un certo grado di libertà nell'acquisizione dell'input, eliminando i caratteri non ammessi, senza generare errori. I valori ammessi sono nessun carattere o i caratteri ASCII tra [0-9] e [q,Q]. Le funzioni chiamanti eseguono ulteriori controlli in base al contesto.
- *getChoice*: acquisizione delle scelte dell'utente
Acquisisce un intero (ASCII) da *getString* e lo verifica contro la condizione che vuole il valore acquisito compreso tra un min e un max, argomenti della funzione. Tramite un parametro *_Bool bZeroLength* è consentito il new line tramite invio, usato per una veloce conferma delle impostazioni del menu Settings che non si intendono modificare.
- *getTry*: acquisizione del tentativo dell'utente
Acquisisce un array di caratteri contenente il tentativo di indovinare il codice segreto, verificando i valori provenienti da *getString* contro le condizioni imposte dalle impostazioni del gioco, generando messaggi d'errore per codice

segreto di lunghezza errata, utilizzo di colori non consentiti, utilizzo di duplicati se non consentiti.

Superati questi controlli, i valori sono memorizzati in una matrice di m righe (numero di tentativi) e n colonne (codice inserito dall'utente) all'interno della struttura dati.

I valori ASCII 'Q' e 'q' sono utilizzati per consentire l'uscita anticipata da una partita senza interrompere l'intero programma con la combinazione *Ctrl+c*.

- *checkTry*: verifica del tentativo utente contro il codice segreto

Un algoritmo che, per ogni tentativo e fino al massimo numero di tentativi consentiti, verifica gli elementi dell'array code contenente il codice segreto e i corrispondenti della riga della matrice *prgcCodeBreakerTries*, contenente il codice inserito dall'utente, aumentando il contatore dei pioli neri ad ogni corrispondenza.

Se gli elementi non sono tutti uguali, passa ad esaminare la corrispondenza di ogni elemento del tentativo rispetto a tutte le posizioni del codice.

In caso di corrispondenza completa dei codici, ritorna *true*. In caso contrario, al raggiungimento del numero massimo di tentativi, ritorna *false*.

L'array *prgbCountedFlags* è utilizzato come bandierina/segnaposto per l'identificazione dei pioli bianchi.

- *printGame*: stampa dello stato della partita e del tavolo di gioco

Ad ogni chiamata, utilizzando i valori registrati nella *struct* per stampare l'esito del tentativo e lo stato del gioco.

- *playGame*: gestione della partita

Il ciclo che parte dal primo tentativo a quello corrente, facendo le chiamate alle diverse funzioni, gestisce il flusso, costruisce l'intero tavolo da gioco, arrivando all'esito finale della partita.

7.6 Programma principale

che gestisce prevalentemente

- Gli *switch -d* e *-h* del programma *mastermind*
- Il menu principale
- Pochi messaggi di interazione e chiamate alle funzioni di supporto
- L'uscita dal gioco

8 Test & debug

8.1 Verifica

I test sono stati condotti per componenti e funzionalità, riguardando quattro macroaree:

- Struttura dati e gestione errori
- Acquisizione dell'input utente
- Usabilità e leggibilità dell'interfaccia
- Esattezza degli algoritmi

La struttura dati è stata verificata in particolare per l'allocazione dinamica della memoria e per la gestione di eventuali errori, simulando delle *failure* in questa fase e verificando i codici d'uscita.

L'acquisizione dell'input utente è stata testata inserendo un gran numero di combinazioni in ingresso, cercando di coprire anche situazioni di inserimento di stringhe estremamente improbabili. Sono stati verificati in particolare tutti i casi di errori gestiti.

Usabilità e leggibilità dell'interfaccia sono state migliorate facendo testare il gioco a diversi utenti e recependone feed-back e suggerimenti che hanno aiutato, ad esempio, a stabilire massimi e minimi per numero di colori e lunghezza del codice, o a rendere più agile il menu delle impostazioni.

Una parte importante del lavoro è stata impiegata nel tentativo di riproduzione di un tavolo da gioco in grado di adattarsi alle impostazioni. Anche la scelta di caratteri *unicode* ha richiesto molto tempo e tutt'ora la corretta visualizzazione dipende dal terminale usato e dall'impostazione dei font.

L'esattezza degli algoritmi è stata testata provando casistiche per tutte le possibili configurazioni di gioco.

I maggiori problemi sono stati riscontrati nella modellazione ed implementazione dell'algoritmo che verifica il tentativo dell'utente e conta i pioli, quando sono permessi i colori duplicati.

8.2 Correzione

Le componenti che hanno subito correzioni rilevanti sono

- La struttura dati, per un errore nella lunghezza dei vettori di white e black pegs, uguale a *iSecretLength* invece che a *iMaxTries*, causando un *core dump* per l'utilizzo di aree di memoria al di fuori del range assegnato agli array.
- gli algoritmi di generazione del codice segreto per la gestione di elementi duplicati o meno
- l'algoritmo di verifica dei pioli è stato oggetto di particolari attenzioni e diverse revisioni
- la funzione di stampa dello stato del gioco è stata corretta più volte

Una fase abbastanza onerosa è stata la ricerca e la riscrittura di tutti i blocchi di loop che, pur funzionando correttamente, facevano uso improprio delle istruzioni *continue* e *break*, in particolare nei cicli *for*.

I tanti errori commessi a causa dell'inesperienza nelle fasi precedenti, durante le fasi di codifica e test mi hanno costretto diverse volte a tornare alla modellazione e, in qualche caso, anche ad aggiustamenti nella fase di progettazione, arrivando, come detto in premessa, a ripensare completamente le strutture dati, ristrutturare profondamente *main function*, funzioni di supporto e utilità, fino a riorganizzare il progetto con *Makefile*, *.h* file e *.c* file, dovendo investire molto più tempo di quanto non sarebbe stato necessario con il corretto approccio.