```
int main(int argc, char** argv)
{
  char c = 0;
  char* commands = "ads pq"; // key commands: "left,right,rotate,confirm,pause,quit"
  int speed = 2; // sets max moves per row
  int moves_to_go = 2;
  int full = 0; // whether board is full
  init(); // initialize board an tetrominoes


  cur =
  state[
  state[

  system
  printf

  while
    {
      dr

    // process user action
    c = getchar();   // get new action
    if (c == commands[0] && !intersect(cur, state[0]-1, state[1])) state[0]--; // move left
    if (c == commands[1] && !intersect(cur, state[0]+1, state[1])) state[0]++; // move right
    if (c == commands[2] && !intersect(cur->rotated, state[0], state[1])) cur = cur->rotated;
    if (c == commands[3]) moves_to_go=0;

    // scroll down
    if (!moves_to_go--)
      {
        if (intersect(cur,state[0],state[1]+1)) // if tetromino intersected with sth
          {
            cramp_tetromino();
            remove_complete_lines();
            cur = &tetrominoes[rand() % NUM_POSES];
            state[0] = (WIDTH - cur->width)/2;
```

MAC122 - PRINCÍPIOS DE DESENVOLVIMENTO DE ALGORITMOS

Ponteiros e Funções

Revisão: Tipos de dados nativos

Tipo de dados: Conjunto de valores equipado com operações

tipo	objeto representado	tamanho
`char`	palavra com sinal	1
`unsigned char`	palavra sem sinal	1
`int`	inteiros	4
`unsigned int`	inteiros não-negativos (naturais)	4
`float`	reais	4
`double`	reais	8
`long double`	reais	16

Para saber tamanho (em bytes): `sizeof(tipo)`

Revisão: Tipos de dados inteiros

Tipo	Valores
`char`	-128 a +127
`unsigned char`	0 a 255
`int`	`-INT_MAX` a `INT_MAX-1`
`unsigned int`	0 a `INT_MAX`

O valor de `INT_MAX` depende da arquitetura do sistema e do compilador e é definido na biblioteca `limits.h`

Revisão: Operações aritméticas

```
+ - * / %
```

```
long double op double = long double
double op float = double
float op int = float
int op unsigned int = int
unsigned int op char = int
char op unsigned char = int
```

Revisão: Comparações

`<, <=, >, >=, ==, !=`

Retornam 1 ou 0

```
long double comp double = long double
double comp float = double
float comp int = float
unsigned int comp int = unsigned int
int comp char = int
char comp unsigned char = int
```

Qual o problema?

```c
#include <stdio.h>

int main(void) {
        int i;
        unsigned int limite;

        limite = 0;
        i = -10;
        /* exibe numeros de -10 a 0 */
        while(i <= limite) {
                printf(" %d", i);
                i = i + 1;
        }
        return 0;
}
```

Qual o problema?

Ao comparar `int` e `unsigned`, `int` é demovido a `unsigned`

```
1  #include <stdio.h>
2
3  int main(void) {
4          int i;
5          unsigned int limite;
6
7          limite = 0;
8          i = -10; /* note: (unsigned) i > 0 */
9          /* exibe numeros de -10 a 0 */
10         while(i <= limite) { /*comp. de int e unsigned*/
11                 printf(" %d", i);
12                 i = i + 1;
13         }
14         return 0;
15 }
```

Unsigned

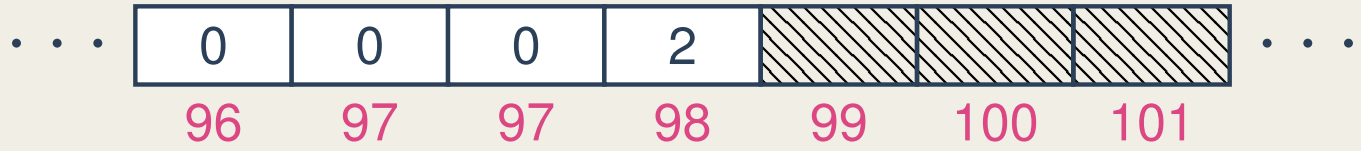
De forma geral, recomenda-se não utilizar tipos `unsigned`; a precisão extra pode ser obtida usando `long` sem correr os riscos introduzidos pela manipulação conjunta de tipos `signed` and `unsigned`

Endereços e Variáveis

- Memória de computador é uma sequência de bytes (palavras)
- Cada byte é acessado através de seu endereço
- Dados ocupam posições contíguas na memória, dependendo de seu tipo (e também do sistema)
  - Ex.: `char` ocupa 1 byte, `int` ocupa 4 bytes etc.
- Uma variável é um abstração do endereço de um valor (de certo tipo) armazenado na memória

Endereços e Variáveis

```
int a=2;
```



- Endereço de variável é obtido pelo operador `&`
- Exemplo:

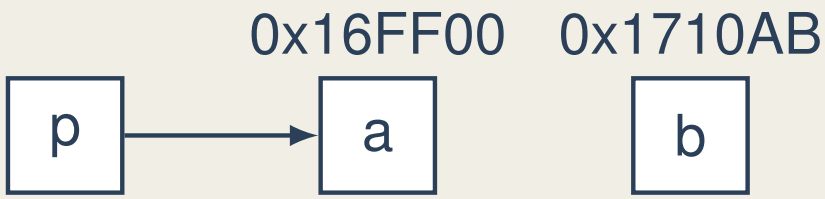
variável	valor
`a`	2
`&a`	96

Nota: Costumamos escrever endereços em base hexadecimal, ex:
`&a = 0x06`

`tipo *nome;`

Tipo especial de variável que armazena (e manipula) endereços de variávies de um determinado tipo; Variável "apontada" é obtida pelo operador `*`

```
1 int a = 2, b = 0;
2 int *p, *q;
3 p = &a;
4 b = *p; *p = 4;
5 q = p+2;
```



variável	valor	variável	valor
`a`	4	`*p`	4
`&a`	0x16FF00	`b`	2
`p`	0x16FF00	`q`	0x16FF02

Ponteiros

Múltiplos ponteiros podem apontar para mesma variável; ponteiro pode apontar para variável nula null;

```
1 int a = 2;
2 int *p, *q, *r;
3 p = &a;
4 q = &a;
5 r = NULL;
```

Ponteiros

- Permitem se referir a uma estrutura de dados grande de forma concisa
- Facilitam compartilhamento de dados entre as partes do programa
- Permitem alocar memória durante a execução do programa
- Permitem criar relações entre variáveis

Qual a saída?

```
1 int a = 2, b;
2 int *p, *q, *r;
3 p = &a;
4 q = p;
5 r = &b;
6 *r = *q;
7 *q += 2;
8 printf("%d %d\n", a, b);
```

Qual a saída?

```
1 int a = 2, b;
2 int *p, *q, *r;
3 p = &a;
4 q = p;
5 r = &b;
6 *r = *q;
7 *q += 2;
8 printf("%d %d\n", a, b);
```

4 2

Funções

- permitem dividir um problema em subproblemas menores
- subproblemas menores são mais fáceis de entender e resolver
- código fica mais portável e reutilizável

```
1  int quad (int n) {
2    return n*n;
3  }
4  int cubo (int n) {
5    return n*n*n;
6  }
7  int main() {
8    int n = 10;
9    printf("%d", quad(n)*cubo(n));
10   return 0;
11 }
```

Funções

`tipo nome (argumentos) bloco`

Executa `bloco` com variáveis definidas em `argumentos` e retorna valor `tipo`

```
1 int fatorial (int n) {
2   int i, f = 1;
3   for (i = 1; i <= n; i++) f *= i;
4   return f;
5 }
```

Use `tipo`/`argumentos`=`void` para funções que não retornam/recebem valores

```
1 void fazNada (void) {
2   return;
3 }
```

Funções

Chamada de funções: `nome` (expressões);

- Expressões são avaliadas antes de serem passadas
- Função não tem acesso a escopo de variáveis na chamada (apenas seus valores)
- Escopo dos argumentos é o bloco (não pode haver re-definição)

Qual a saída?

```
1 int inc (int n) {
2     n = n + 1;
3     return n;
4 }
5 int n;
6 n = 3;
7 inc(n);
8 printf("%d", n);
```

Qual a saída?

```
1 int inc (int n) {
2     n = n + 1;
3     return n;
4 }
5 int n;
6 n = 3;
7 inc(n);
8 printf("%d", n);
```

3

Qual a saída?

```
1 int inc (int n) {
2     n = n + 1;
3     return n;
4 }
5 int n;
6 n = 3;
7 n = inc(n);
8 printf("%d", n);
```

Qual a saída?

```
1 int inc (int n) {
2     n = n + 1;
3     return n;
4 }
5 int n;
6 n = 3;
7 n = inc(n);
8 printf("%d", n);
```

4

Qual o problema?

```
1 /* troca os valores de x e y */
2 void troca (int x, int y) {
3     int tmp; /* variavel dinamica */
4     tmp = x; x = y; y = tmp;
5 }
```

Passagem de variáveis por referência

Permite modificar valor de variáveis no escopo orginal

```
1 void func (int *x, int *y, int *z) {
2   *x = 10; *y = 20; *z = 30;
3 }
4 int x, y, z;
5 fun(&x, &y, &z);
```

Troca de valores

```
1 /* Troca valores de variaveis */
2 void troca (int *p, int *q) {
3   int t;
4   t = *p; *p = *q; *q = t;
5 }
```

Função principal

`int main (void) bloco`

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

ao ser chamado, programa executa função `main`; função deve retornar valor nulo se programa terminou sem problemas e valor não nulo caso contrário.

```
1 int main () {
2     printf ("%d! = %d", 10, fatorial(10));
3     return 0;
4 }
```

Função principal

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

- linha de comando: `prog arg1 arg2 ...  argc`
- `argc` contém o número de argumentos na linha de comando
- `argv` contém um vetor de strings contendo argumentos

```
1 /* Imprime todos os argumentos na linha de comando */
2 int main (int argc, char **argv) {
3   for (i = 0; i < argc; i++)
4       printf ("%s\n", argv[i]);
5   return 0;
6 }
```

Protótipos de funções

Uma função só pode chamar outras funções declaradas anteriormente. Suponha que temos duas funções `f` e `g`: `f` chama `g` em sua definição e `g` chama `f`

```
1  /* ERRO */
2  int f (int x) {
3    if (x>0) return g(x-1);
4    else return 0;
5  }
6
7  int g (int x) {
8    if (x>0) return f(x/2);
9    else return 0;
10 }
```

Protótipos de funções

Solução: declarar protótipo de funções antes de suas definições

```
1  int f (int x); /* declaracao de f */
2  int g (int x); /* declaracao de g */
3  /* definicao de f */
4  int f (int x) {
5    if (x>0) return g(x-1);
6    else return 0;
7  }
8  /* definicao de g */
9  int g (int x) {
10   if (x>0) return f (x/2);
11   else return 0;
12 }
```

Protótipos de funções

Protótipos podem ser utilizados para separar especificação de interfaces (o que fazer) de implementações (como fazer)

```
1  /* interface */
2  int f (int x);
3
4  /* programa principal */
5  int main() {
6   printf ("%d\n", f(10));
7   return 0;
8  }
9
10 /* implementacao */
11 int f (int x) {
12  return x*x;
13 }
```

Arquivos de interface

- Objetivo: separar declaração e definição de funções (ou seja, o que a função faz de como ele faz)
- Declarações são descritas em arquivo .h
- Definições são descritas em arquivo .c
- Podemos trocar implementações de funções sem alterar código
- Para usar, declara-se `#include <arquivo.h>` ou `#include "arquivo.h"`

Arquivos de interface

```c
/* fatorial.h */
int fatorial (int n);
```

```c
/* fatorial.c */
int fatorial (int n) {
  int i, f = 1;
  if (n < 1) return 0;
  for (i = 1; i <= n; i++)
    f *= i;
  return f;
}
```

```c
/* programa.c */
#include "fatorial.h"

int main (void) {
  printf ("%d",fatorial(10));
}
```

Compilação (gcc): `gcc -o programa programa.c fatorial.c`

Bibliotecas

- Uma biblioteca é um conjunto de definições de funções armazenadas em um mesmo arquivo (em geral relativas a um mesmo tópico)
- C conta com várias bibliotecas padrão para tarefas comumente utilizadas
- Para utilizar uma função em uma biblioteca, deve-se incluir o arquivo de interface correspondente com a diretiva `#include`
- Por exemplo, a função `printf` faz parte da biblioteca `stdio` (standard input/output library) que é acessada incluindo o arquivo `stdio.h`

Biblioteca stdio

```
1  #define NULL 0
2  #define EOF (-1)
3  int fgetc (FILE *f);
4  int getc (FILE *f);
5  int fputc (int c, FILE *f);
6  int putc (int c, FILE *f);
7  int printf (char *s, ...);
8  int scanf (char *s, ...);
9  typedef struct {
10   int             _cnt;
11   unsigned char *_pnt;
12   unsigned char *_base;
13   unsigned char _flag;
14   unsigned char _file;
15 } FILE;
16 extern FILE *stdin, *stdout;
17 FILE *fopen (char *str, char *modo);
18 ...
```

Biblioteca stdlib

```
1  #define EXIT_FAILURE 1
2  #define EXIT_SUCCESS 0
3  void exit (int status);
4  #define RAND_MAX (32767)
5  int rand (void);
6  void srand (unsigned int u);
7  int atoi (char *s);
8  void *malloc (unsigned int N);
9  void free (void *pntr);
10 void qsort (void *base, unsigned int nmemb, unsigned int
       size, int (*compar)(void *, void *));
11 ...
```

Biblioteca math

```
1 double sin (double x);
2 double cos (double x);
3 double tan (double x);
4
5 double exp (double x);
6 double log (double x);
7
8 double sqrt (double x);
9 ...
```

Biblioteca string

```
1 unsigned int strlen (char *x);
2 int strcmp (char *x, char *y);
3 char *strcpy (char *y, char *x);
4 ...
```

Biblioteca limits

```
1 #define INT_MIN (-2147483648)
2 #define INT_MAX (2147483648)
3 #define UINT_MAX (4294967295)
4 #define DBL_MIN (2.2250738585072014E-308)
5 #define DBL_MAX (1.7976931348623157E+308)
6 ...
```

Biblioteca time

```
1 long time (long *t);
2 #define CLOCKS_PER_SEC (1000000)
3 long clock (void);
4
5 ...
```

Números (pseudo)aleatórios

`int rand (void);`

Gera números inteiros (pseudo)aleatórios entre 0 e uma constante `RAND_MAX-1`

`int srand (unsigned int s);`

Configura *semente* do algoritmo gerador de números pseudoaleatórios

```
1    int i, n = 5;
2
3    srand (time (NULL));
4    /* Exibe n numeros aleatorios entre 0 e 49 */
5    for ( i = 0 ; i < n ; i++ )
6        printf ("%d ", rand () % 50);
```

Qual o problema?

```c
/* Gera um numero entre 0 e 5 com igual probabilidade */
int dado (void) {
    int r;
    r = rand();
    return r % 6;
}
/* DICA: Suponha que RAND_MAX = 13 */
```

Qual o problema?

```c
/* Gera um numero entre 0 e 5 com igual probabilidade */
int dado (void) {
    int r;
    r = rand();
    return r % 6;
}
/* DICA: Suponha que RAND_MAX = 13 */
```

r	0	1	2	3	4	5	6	7	8	9	10	11	12
r % 6	0	1	2	3	4	5	0	1	2	3	4	5	0

Qual o problema?

```c
/* Gera um numero entre 0 e 5 com igual probabilidade */
int dado (void) {
    int r;
    while ((r = rand()) >= 6) ;
    return r;
}
```

r	0	1	2	3	4	5	6	7	8	9	10	11	12
r % 6	0	1	2	3	4	5	–	–	–	–	–	–	–

Solução

```c
/* Gera um numero entre 0 e 5 com igual probabilidade */
int dado (void) {
    int r, corte = RAND_MAX - (RAND_MAX % 6);
    while ((r = rand()) >= corte) ;
    return r % 6;
}
```

r	0	1	2	3	4	5	6	7	8	9	10	11	12
r % 6	0	1	2	3	4	5	0	1	2	3	4	5	–

Números (pseudo)aleatórios

Gerando números inteiros aleatórios no intervalo $[a, b)$:

```
1
2  /*
3   * 1. Gera um número inteiro entre [0,RANDMAX]
4   * 2. Normaliza o número para um real no intervalo [0, 1)
5   * 3. Redimensiona o valor para o intervalo [0,b-1)
6   * 4. Arredonda (para baixo) o valor para obter inteiro
7   * 4. Translada o inteiro para o interval [a,b)
8   */
9  int randint (int a, int b) {
10     double r, x;
11     int i;
12     r = rand();
13     x = r / (RAND_MAX+1);
14     i = x * (b - a + 1);
15     return a + i;
16 }
```

Pré-processamento

São operações realizadas pelo compilador antes da compilação (isto é, antes da transformação em código de máquina); em geral, são operações simples de processamento de texto e manipulação de arquivos

`#include <arquivo>`
`#include "arquivo"`
Substitui a linha com o conteúdo de `arquivo`; se o nome é delimitado por <> o arquivo é procurado numa lista de diretórios padrão (ex. `/include`), caso contrário o arquivo é procurado no diretório atual

```
1 #include <fatorial.h>
2
3 int main (void) {
4   fatorial(10);
5   return 0;
6 }
```

`#define nome valor`

Substitui toda ocorrência de `nome` no arquivo por `valor`

```
1 #define PI 3.1415
2 #define FAT10 fatorial(10)
3
4 int main (void) {
5   printf("O valor aproximado de pi e' %d\n", PI);
6   printf("O valor de 10! e' %d\n", FAT10);
7 }
```

```
#ifdef nome
#ifndef nome
#endif
```

Remove código se `nome` não estiver definida/indefinida (por `#define`)

```
1 #ifndef _WINDOWS_
2 /* code */
3 #else
4 /* code */
5 #endif
```

```
1 #ifndef _MINHA_BIBLIOTECA_
2 #define _MINHA_BIBLIOTECA_
3 /* Retorna n! */
4 int fatorial (int n);
5 #endif
```

Serve para evitar que função seja redefinida quando arquivo (de interface) é incluído múltiplas vezes

Para casa

- Exercícios 2A-2C