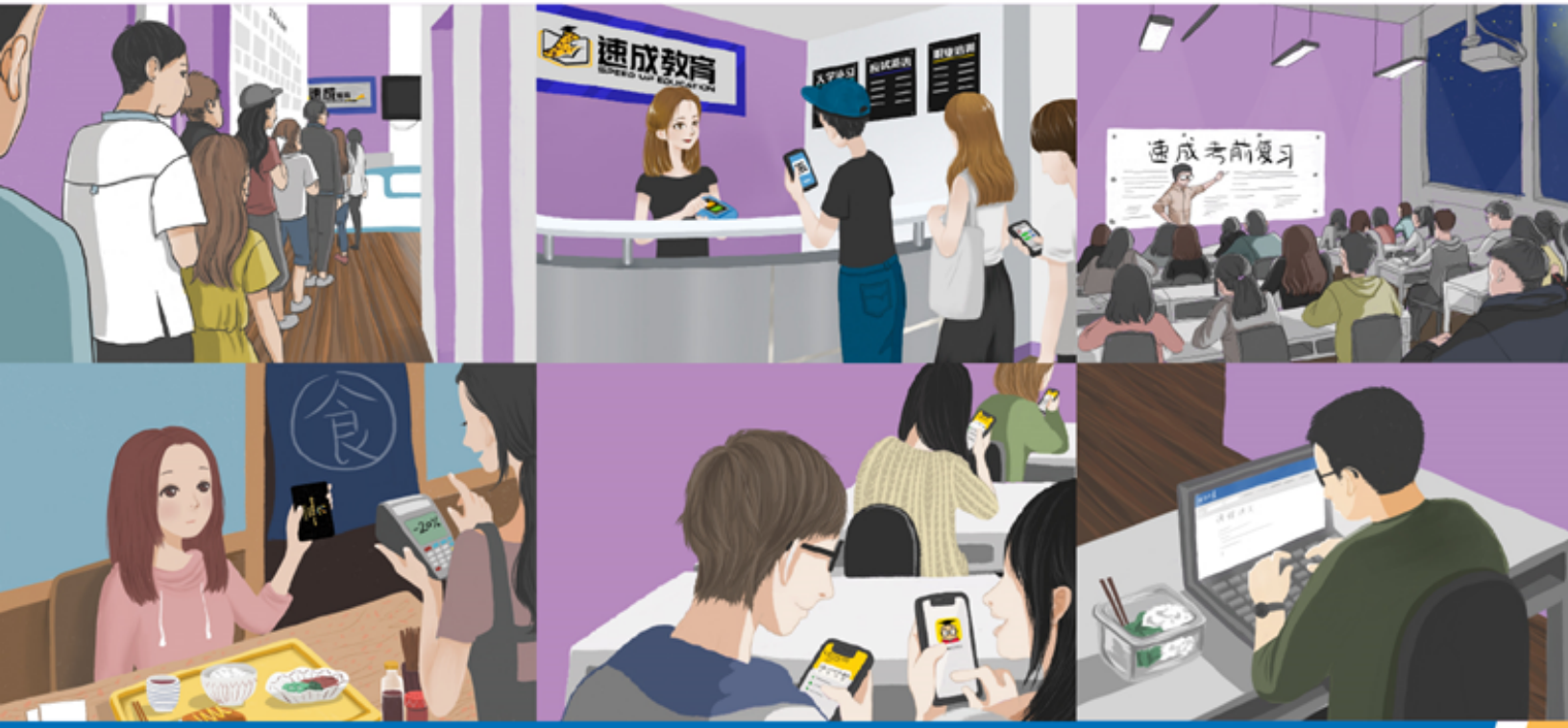
速成教育
SPEED UP EDUCATION



CSCB09

Introduction to Computer Science II

导师： **Vincent Tse**

UTSC Week 4 Class | 2024/2/6

Disclaimer

This study handout is provided by Speed Up Education and its affiliated tutors. The study handout seeks to support your study process and should be used as a complement, not substitute to the university course material, lecture notes, problem, sets, past tests and other available resources.

The study handout is distributed for free to the students participating in Speed Up Education classes and is not for sale or other commercial uses whatsoever. We kindly ask you to refrain from copying or selling in part or in whole any information provided in the handout.

Thanks for choosing Speed Up Education. We wish you the best of luck in your exam.

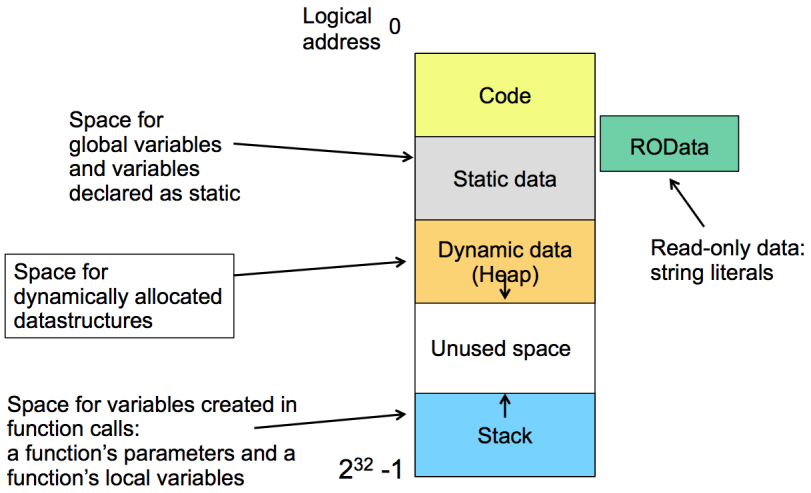
Types and Memory Size

Type	Size	Value Range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615
pointer	8 bytes	…

```
int a = 1234;
unsigned char *p = &a;
short *q = &a;

*(p + 1) = 66;
*(q + 1) = 666;

printf("%d\n", a);
printf("%c\n", a);
printf("%u\n", a);
```

Memory Allocation

```
int main() {

    char s1[] = "CSC209";

    char *s2 = "CSC209";

    char *s3 = malloc(7);
    strcpy(s3, "CSC209");

    s1[3] = 'B';
    s1[3] = 'B';    /* read only */
    s1[3] = 'B';
}
```



Typedef

```
typedef unsigned long uLong;
typedef int* IntPtr;

int main() {
    uLong x = 6666666;
    IntPtr p = (int *) &x;
}
```

Compound data Type

```
typedef struct human {
    char name[30];
    int age;
} Human;

int main() {
    Human vc;
    vc.age = 18;

    Human *paco = (Human *) malloc(sizeof(Human));
    paco->age = 69;  // => (*paco).age = 69;
}
```

Example

Complete the code below according to the comments.

```
typedef struct month {
    char name[10];
    int num_days;
    int num_holidays;
    char **holidays;
} Month;

/* Return a value greater than 0 if m1 is longer than m2, less then 0 if m2
 * is longer than m1 and 0 if they have the same number of days
 */
int longer(Month *m1, Month *m2) {
    return m1->num_days - m2->num_days;
}

int main() {

    struct month dec;

    // Set the number of days in dec to 31



    // Set the name of dec to December









    // Allocate space for an array of 2 december holidays
    // and set the holidays to mutable strings "christmas" and "thanksgiving"
```

```
    Month *june;

    // Create a second month pointed to by the variable june declared above.



    // Set the number of days to june to 30.



    // Call longer correctly on june and dec so this code works.



    if (                                                              ) {
        printf("December is longer than June.\n");
    }

    // free June




    return 0;
}
```

Command line arguments

```
int main(int argc, char *argv[]) {
    for (int i = 0; i < argc; i++)
        printf("%s", argv[i]);
}
```

```
> gcc main.c -o myprog
> ./myprog apple 123
myprog
Apple
123
>
```

Default Input / Output Streams

```
int n;
scanf("num=%d", &n);
printf("num: %d\n",n);
```

```
> ./myprog
> num=123
num: 123
>
```

`printf(format, args)` => `fprintf(stdout, format, args)`
`scanf(format, args)` => `fscanf(stdin, format, args)`

`stdin` and `stdout` are *pre-opened* streams for standard-in and standard-out.
There is one more: `stderr`, so you can output error messages and leave 'stdout' for normal data/output.

Stream	Standard Name	Default Location
Standard Input	stdin	Keyboard (Terminal)
Standard Ouput	stdout	Screen (Terminal)
Standard Error	stderr	Screen (Terminal)

* Usually just use 'perror' to print error message to stderr:

```
fprintf(stderr, "Somethings wrong.");
perror("Somethings wrong.");
```

Creating your own Stream (local File)

Open

```
FILE *fopen(const char *filename, const char *mode)
```

Mode	meaning	if file exists	if not
"r"	read	cool	error
"w"	write	truncate	create
"a"	append	cool	create
"r+"	read+write	cool	error
"w+"	write+read	truncate	create
"a+"	append+read	cool	create

"read" and "write" start at the beginning of the file.
"append" start at the end of the file.
"truncate" = erase original content.

Returns NULL if error (This one you should always check) !!!!!!!!!!!!!!!!!!!!

Close

```
int fclose(FILE *stream)
```

Returns **0** if success, `EOF` if error.

Formatted I/O

Read and Write ***Characters*** Return the number of item read/write

```
int fprintf(FILE *stream, const char *format, ...)
int fscanf(FILE *stream, const char *format, ...)
```

```
int i = 5;
fprintf(f, "%d", i);        > 0 0 1 1 0 1 0 1
```

Character I/O

Read and Write ***ONE*** single character:

```
int putchar(int c) /* stdout */
int putc(int c, FILE *stream)
int getchar(void) /* stdin */
int getc(FILE *stream)
```

Returns the character written/read if success, `EOF` if error or end of stream.

Important:

`EOF` fits in `int` but not `char`. Correct usage of '`getchar`' and '`getc`' involves:

1. Store return value in '`int`' variable, not '`char`'.
2. Test for `EOF`.
3. If not `EOF`, safe to down-convert to '`char`'.

```
int res;
if ((res = getchar()) == EOF) {
    fprintf(stderr, "failed to read char ...\n");
    exit(1);
}
char c = (char) res;
/* do something with c */
```

String I/O

`int fputs(const char *string, FILE *stream)`
Returns `EOF` if error.

`char *fgets(char *dest, int n, FILE *stream)`
Reads at most n − 1 characters or **until (and including) newline.**
Returns `NULL` if error or end of stream, `dest` if success.

```
char name[20];
if (fgets(name, 20, stdin) == NULL) {
    fprintf(stderr, "failed to read string...\n");
    exit(1);
}
printf("name: %s\n", name);

> ./myprog
> Vincent
name: Vincent

>
```

Note the the \n here

Arbitrary Data I/O

Read and Write *Raw Bytes*

```
size_t fread(void *dest, size_t s, size_t n, FILE *stream)
size_t fwrite(const void *data, size_t s, size_t n, FILE *stream)
```

Read/Write **n items**, each item **s bytes**.
Returns the **number of items** read/written.

```
short i = 5;
fwrite(&i, sizeof(short), 1, f);   > 0 0 0 0 0 0 0 0   0 0 0 0 0 1 0 1
```

Cross-platform watchout:

The same raw bytes can mean different values on different platforms. Check compatibility!

```c
typedef struct record {
    char name[20];
    short yob;    // year of birth
} Record

int main() {
    // array of at most 10 records
    Record records[10];
    int num_records;

    FILE *f;
    if ((f = fopen("records", "r")) == NULL) {
        fprintf(stderr, "failed to open ...\n");
        return 1;
    }
    if ((num_records = fread(records, sizeof(Record), 10, f)) == 0) {
        fprintf(stderr, "failed to read any record.. file empty?\n");
        return 2;
    }
    for (int i = 0; i < num_records; i++)
        printf("%s: %d\n", records[i].name, records[i].yob);
    return 0;
}
```

Seeking

`int fseek(FILE *stream, int offset, int origin)`

Set the file pointer to the location `origin + offset`

origin	**go to:**
SEEK_SET	beginning of the file stream
SEEK_END	end of the file stream `EOF`
SEEK_CUR	current position

Returns **non-zero** on error.

Assume the file “records” already have 2 struct data, { "Vincent", 1995} { "Paco", 1920 }

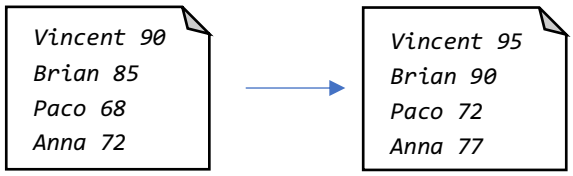
```
int get_yob(FILE *f, const char *name, short *pyob) {
    // reset the file pointer to the first byte
    fseek(f, 0, SEEK_SET);

    Record rcd;
    while (fread(&rcd, sizeof(Record), 1, f) == 1) {
        if (strcmp(rcd.name, name) == 0) {
            *pyob = rcd.yob;
            return 1;
        }
    }
    return 0;
}


void set_yob(FILE *f, const char *name, short yob) {
    // reset the file pointer
    fseek(f, 0, SEEK_SET);

    Record rcd;
    while (fread(&rcd, sizeof(Record), 1, f) == 1) {
        if (strcmp(rcd.name, name) == 0) {
            // change the year in the file.
            fseek(f, -2, SEEK_CUR);
            fwrite(&yob, sizeof(short), 1, f);
            return;
        }
    }
    // didn’t find the name, append a new record
    strcpy(rcd.name, name);
    rcd.yob = yob;
    fwrite(&rcd, sizeof(Record), 1, f);
}
```

```
int main() {

    FILE *f;
    if ((f = fopen("records", "w+")) == NULL) {
        fprintf(stderr, "failed to open ...\n");
        return 1;
    }

    set_yob(f, "Vincent", 2000);    // this modify Vincent's yob in file
    set_yob(f, "Brian", 1890);      // this create a new record in file
    set_yob(f, "Paco", 1930);       // this modify Paco's yob in file

    short yob;


    if (get_yob(f, "Brian", &yob))
        printf("Brian yob: %d\n", yob);
    else
        printf("Brian is not found..\n");

    if (get_yob(f, "Paco", &yob))
        printf("Paco yob: %d\n", yob);
    else
        printf("Paco is not found..\n");

    return 0;
}
```

Redirecting Input / Output Stream

```
int main () {
    int n;                          > ./myprog < 5 > result.txt
    scanf("%d", &n);
    printf("%d", n);
}
```

Exercise




```
#include <stdio.h>

int main() {
    FILE *input_file, *output_file;

    if ( ((input_file = fopen("input.txt", "r")) == NULL ) {
        fprintf(stderr, "Error opening file\n");
        return 1;
    }

    if ( ((output_file = fopen("output.txt", "w")) == NULL ) {
        fprintf(stderr, "Error opening output file\n");
        return 1;
    }

    // copy every line in input_file to output_file but with each mark
    // curved up by 5.
```

```
    if (fclose(input_file) != 0) {
        fprintf(stderr, "Error closing file\n");
        return 1;
    }

    if (fclose(output_file) != 0) {
        fprintf(stderr, "Error closing file\n");
        return 1;
    }

    return 0;
}
```

Macro

`#define TABLE_SIZE 20`
`#define GREETING "Hello"`

A macro is a text *substitution definition.*
Will be substitute into the code during the *pre-processing* stage (before gcc).

`#undef GREETING`
Remove macro definition.

Example: use macro as code

```
#include <stdio.h>
#define BEGIN {
#define END }
#define NUMBER int

NUMBER main() BEGINE
    NUMBER i = 1;
    while (i <= 10) BEGIN
        printf("%d \n", i);
    END
END
```

→

```
int main() {
    int i = 1;
    while (i <= 10) {
        printf("%d \n", i);
    }
}
```

Example: use macro as variable

```
#include <stdio.h>
#ifndef MESSAGE
    #define MESSAGE "I hate CSCB09!"
#endif
int main() {
    printf("%s\n", MESSAGE);
}
```

→

```
int main() {
    printf("%s\n", "I hate CSCB09!");
}
```

Example: use *parameterized* macro to simulate functions

```
#include <stdio.h>
#define square(x) ((x) * (x))
int main() {
    printf("square of 3: %d\n", square(3));
}
```

→

```
int main() {
    printf("square of 3: %d\n", ((3) * (3)));
}
```

What is good about this? Quick (not in run-time)

However, Its not oftenly used…

```
#include <stdio.h>
#define square(x) ((x) * (x))
int main() {
    int x = 3;
    printf("%d\n", square(x++));
}
```

→

```
int main() {
    printf("%d\n", ((x++) * (x++)));
}
```

Gaurding

```
#include <stdio.h>
int main() {
    #ifdef DEBUG
    printf("I hate cscb09")
    #endif
}
```

Including Library

`#include <foo.h>`

Look for file in system-wide places, e.g., '/usr/include'.

`#include "foo.h"`

Look for file among user source code too.

Header Files

Header files serve as interface

`linkedList.h` contains information for a linkedList; users do `#include "linkedList.h"`.

~~`linkedlist.c`~~ contains implementation detail; ideally users do not need to know.

Header files usually contains:

1. Macro definitions
2. struct
3. typedef
4. Public functions
5. global variables

Illegal to define a struct or typedef the second time.

Each header file `#define` a macro to flag *"I have been seen"*.

`linkedList.h` can go:

```
#ifndef _LINKEDLIST_H
#define _LINKEDLIST_H

#include <stdio.h>
#define MAX_STR_LEN 1024

typedef struct node {
    char name[MAX_STR_LEN];
    struct node *next;
} Node;

Node *insert(Node *head, const char *name);
void print_list(Node *head);

#endif
```

Idea:

First time, `_LINKEDLIST_H` not yet defined, **don't skip**, define `_LINKEDLIST_H` and Node.
Second time onwards, since `_LINKEDLIST_H` now defined, **skip**. No problem ☺

GCC Complie

1. Compile to object files:
   `gcc -c file1.c` (but only if necessary)
   `gcc -c file2.c` (but only if necessary)
   `gcc -c file3.c` (but only if necessary)
2. Link to executable (but only if any of the above happened):
   `gcc file1.o file2.o file3.o -o myprog`

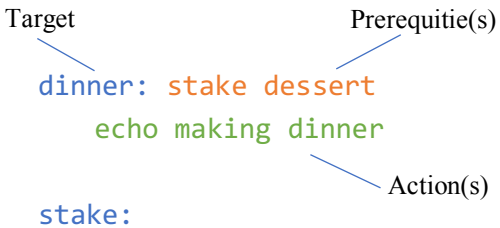
Wouldn't you like to automate "but only if necessary"?
That's what the 'make' program and "Makefile"s are for.

Make

`Make` command will excute the file called **`Makefile`** or **`makefile`**

If Target does **not** exist: execute Acion to create it.
If Target exist: ONLY execute Action if any of the Prerequitie(s) is more recent than Target.

Example 1




```
dinner: stake dessert
    echo making dinner

stake:
    echo making stake

dessert:
    echo making dessert
```

```
> make stake
echo making stake
making stake
> make dessert
echo making dessert
making dessert
> make
echo making stake
making stake
echo making dessert
making dessert
echo making dinner
making dinner
> make
...same as above ...
```

Example 2

```
dinner: stake dessert
    echo making dinner

stake:
    echo making stake > stake

dessert:
    echo making dessert > dessert
```

```
> ls
Makefile
> make
echo making stake > stake
echo making dessert > dessert
echo making dinner
making dinner
> ls
Makefile stake dessert
> make
echo making dinner
making dinner
> echo "new dessert" > dessert
> make
echo making dinner
making dinner
```

How to make dinner with the new

Example 3

```
dinner: stake dessert
    echo making dinner

stake:
    echo making stake > stake

dessert: chef_vc
    echo making dessert >
dessert
```

```
> ls
Makefile stake dessert
> make
No rule to make target 'chef_vc',
needed by 'dessert'. Stop.
> echo "new dessert" > chef_vc
> make
echo making dessert > dessert
echo making dinner
making dinner
> make
echo making dinner
making dinner
> echo "new dessert 2" > chef_vc
> make
echo making dessert > dessert
echo making dinner
making dinner
```

Example 4 (variable)

```
PRINT=echo

dinner: stake dessert
    $(PRINT) making dinner

stake:
    $(PRINT) making stake > stake

dessert: chef_vc
    $(PRINT) making dessert > dessert
```

Most basic Makefile clause (rule) goes like:

```
CFLAGS = -Wall -g -std=c99 -Werror

myprog: file3.o file2.o file1.o
    gcc $(CFLAGS) -o myprog file3.o file2.o file1.o

file3.o: file3.c file3.h file2.h file1.h
    gcc $(CFLAGS) -c file3.c

file2.o: file2.c file2.h file1.h
    gcc $(CFLAGS) -c file2.c

file1.o: file1.c file1.h
    gcc $(CFLAGS) -c file1.c

clean:
    rm myprog *.o
.PHONY clean
```

If `file2.o` does not exist, or if at least one of `file2.c file2.h file1.h` is newear than `file2.o`, then run `gcc -c file2.c`

Do not treat this as an actual file.

Automatic Variables And Pattern Rules

```
CFLAGS = -Wall -g -std=c99 -Werror

myprog: file3.o file2.o file1.o
    gcc $(CFLAGS) -o $@ $^

%.o : %.c
    gcc $(CFLAGS) -c $<

file3.o: file3.h file2.h file1.h
file2.o: file2.h file1.h
file1.o: file1.h

clean:
    rm myprog *.o
.PHONY clean
```

Petterns

`$@`	target
`$^`	All prerequistes
`$<`	First prerequistes

Example:

Olivia wishes to modularize the following C file into multiple C files for separate compilation.

```
typedef struct circle {
    double radius;
    double cx , cy;
} circle;

double circle_area(const circle *c) {
    return 3.14159 * c->radius * c->radius;
}

double cyl_vol(double radius , double height) {
    circle c;
    c.radius = radius;
    c.cx = 0;
    c.cy = 0;
    return area (&c) * height;
}

int main(void) {
    printf (" %f\n" , cyl_vol (4, 10));
    return 0;
}
```

Olivia wishes to put circle_area in ‘circle.c’, cyl_vol in ‘cyl.c’, and main in ‘myprog.c’; the circle struct also needs to be moved to a suitable header file.

1. Write the corresponding header files ‘circle.h’ and ‘cyl.h’.

2. Write a Makefile for building an executable from the C source files after the modularization. The executable is to be called ‘myprog’.

Function Pointers

```
int add(int a, int b) {
    return a + b;
}

int main() {
    int res;
    int (*p) (int, int);

    p = &add;              p = add;
    res = (*p) (2, 3);     res = p(2, 3);
    printf("%d", res);
}
```

Practice Write the function header;

```
int main() {
    double **matrix;
    int *column;
    double sum;
    // Assume three variables are appropriately initliazed
    mystery1(matrix[0], column, &sum);
```

```
    char *cats[] = { "Chelsea", "Buster Brown" };
    int weigths = { 10, 15 };
    char vet[100];
    strncpy(vet, mystery2(cats[1], weigths[1]), sizeof (vet));
```

```
    short *p;
    void **rest;
    char *s;
    // Assume three variables are appropriately initliazed
    p = mystery3(rest + *p, &rest, *rest)(s, 10);
```

```
}
```

Fork

Parent Process

pid = 677

```
int main() {
    int i = 5;

    printf("%d\n", i);

    pid_t pid = fork();

    if (pid > 0)
        i = i + 2;
    else if (pid == 0)
        i = i - 2;
    else
        perror("failed to fork");

    printf("%d: %d\n", (int) getpid(), i);
    return 0;
}
```

Output

```
5
677: 7
678: 3
```

OR

Output

```
5
678: 3
677: 7
```

Child Process

pid = 678

```
int main() {
    int i = 5;

    printf("%d\n", i);

    pid_t pid = fork();

    if (pid > 0)
        i = i + 2;
    else if (pid == 0)
        i = i - 2;
    else
        perror("failed to fork");

    printf("%d: %d\n", (int) getpid(), i);
    return 0;
}
```

Example

```
int main() {
    int i = 5;

    pid_t pid = fork();

    if (pid < 0) {
        perror("failed to fork");
        exit(1);
    }
    // child
    else if (pid == 0) {
        sleep(5); // do something
        printf("Child done..\n");
        exit(1);
    }
    // parent
    else if (pid > 0) {
        printf("Parent done..\n");
    }
    return 0;
}
```

Output

```
Parent done..
Child done..
```

Orphan Process

Child process is still running but Parent process has finished already.

Child process's ppid becomes 1.

Example

```
...
    // child
    else if (pid == 0) {
        sleep(5); // do something
        printf("Child done..\n");
        exit(1);
    }
    // parent
    else if (pid > 0) {
        printf("Parent waiting for child :)\n");
        wait(NULL);
        printf("Parent done..\n");
    }
...
```

Output

```
Parent waiting for child :)
Child done..
Parent done..
```

```
...
    // child
    else if (pid == 0) {
        sleep(5); // do something
        printf("Child done..\n");
        exit(1);
    }
    // parent
    else if (pid > 0) {
        printf("Parent waiting for child :)\n");
        wait(NULL);
        sleep(30);
        printf("Parent done..\n");
    }
...
```

Output

```
Parent waiting for child :)
Child done..
Parent done..
```

Zombie Process

Child process is done way before parent process is finished..
But the memory of the child process is locked.

Now we can wait for child, but how to get the return value from child (42) ?

```
...
    else if (pid == 0) {
        sleep(5); // do something
        printf("Child done..\n");
        exit(42);
    }
    // parent
    else if (pid > 0) {
        printf("Parent waiting for child :)\n");
        int status;
        pid_t childPid = wait(&status);
        int childReturnValue = WEXITSTATUS(status);
        printf("Parent knows child %d finished with status %d\n",
                (int) childPid, childReturnValue);
        printf("Parent done..\n");
    }
...
```

Output

```
Parent waiting for child :)
Child done..
Parent knows child 678 finished with status 42
Parent done..
```

CORRECT WAY to use wait..

```
...
// parent
else if (pid > 0) {
    printf("Parent waiting for child :)\n");
    int status;
    pid_t childPid;
    if ((childPid = wait(&status)) == -1) {
        perror("failed to wait");
    else {
        if (WIFEXITED(status))
            printf("Child %d terminated with %d\n",childPid, WEXITSTATUS(status));
        else if
            printf("Child %d terminated with signal %d\n",childPid, WTERMSIG(status));
    }
    printf("Parent done..\n");
...
```

Exec

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);

int execv(const char *path, const char *arg[]);
int execvp(const char *file, const char *arg[]);
```

p – search from PATH (env) variable

`file` is the path name of an executable file to be executed.
`arg` is the string we want to appear as `argv[0]` in the executable. By convention, `argv[0]` is this the file
name of the executanle, normally it's set to the same as `file`.
`...` are the *additional* arguments to give to the executable.

Exec requires a NULL as the last argument.

Example:

```
int main() {
    printf("About to call execl. My PID is %d\n", getpid());

    if (execl("./hello", "hello", (Char *)NULL) == -1) {
        perror(":(");
        return 1;
    }

    /* Code below won't be run because snapped away from exec. */
    printf("I'm still here\n");
    return 0;
}
```

```
int main() {
    printf("HELLO. My PID is %d\n", getpid());
    return 0;
}
```

Output

```
About to call execl. My PID is 677
HELLO. My PID is 677
```

Example 2:

```
int main() {
    int i = 5;

    pid_t pid = fork();

    if (pid < 0) {
        perror("failed to fork");
        exit(1);
    }
    // child
    else if (pid == 0) {
        /* Basically running "ls -l" in terminal */
        if (execl("ls", "ls", "-l", (Char *)NULL) == -1) {
            perror(":(");
            return 1;
        }
    }
    // parent
    else if (pid > 0) {
        /* This parent is nice and waits for child to finish */
        wait(NULL);
        return 0;
    }
}
```