סדנת תכנות בשפת סי (67315)

מסכם:

יחיאל מרצבך

סינטקס בסיסי בשפת C

הקדמה

C הינה שפה בסיסית יותר, במובן שקל יותר להעביר אותה לשפת מכונה, בשונה מפייתון למשל. היא מהירה ויעילה הרבה יותר מעצם היותה שפת LowLevel. השפה מיועדת לשימוש במערכות הפעלה כגון לינוקס, במעבדים קטנים ובמקומות בהם נדרש שימוש מצומצם בזיכרון ובמקום.

ב-C אין שגיאות זמן ריצה (ולכן קשה לדבג). צריך לדעת לנהל את הזיכרון. אמנם, אפשר להתנהל ב-C בלי לגעת בזיכרון וכך נעשה בשבועיים הראשונים.

++C היא הרחבה של C ובה אין צורך לנהל את הזיכרון. חלק גדול מהקורס מתעסק בבאגים.

Coding-Style הינו חשוב כי הוא קובע את **הסטנדרט** של הקוד. אפשר להשתמש ב-CamelCase או ב-sanke_case. הוא גם משמעותי גם לשמות של קבצים, הערות, ביטויים ועוד.

בדומה לImport בפייתון, בC קיימת המילה השמורה Inculde. כעת נראה דוגמת הדפסה ב-C:

```
// This line is a comment,
/* and those lines also.
Next line includes standard I/O header file */
#include <stdio.h> //part of the C Standard Library

// main function - program entry point.
// Execution starts here
int main()
{ // {…} define a block
  printf("Hello class!\n");
  return 0;
}
```

Printf היא פקודת ההדפסה. ב-C **חייבת** להיות פונקציית main, שבמקרה שלנו מחזירה 0.

קומפליציה – צעדים ראשוניים

בשונה מבפייתון, ב-C ישנם שני שלבים. השלב הראשון מעביר את השפה ל'שפת מכונה' – executable באמצעות **הקומפליציה**, כשבשלב השני מתבצעת **ההרצה**. הקומפליציה מתבצעת באמצעות קריאה לקומפיילר, במקרה שלנו הוא gcc:

```
> gcc -g -Wall hello.c -o hello
```

המלה -wall מאפשרת לקבל את רוב שגיאות הקומפיילר והמילה -o מגדירה את שם הקובץ.על מנת שנוכל לדבג, חשוב להכניס -g.

ניתן לעשות זאת גם באופן דיפולטיבי, כך:

```
> gcc hello.c
```

על מנת להריץ אותו יש לכתוב `a.out`.

כמו כן, נוכל להתבונן בדוגמה הבאה:

```
> gcc -std=c99 -Wall hello.c -o hello
> hello
```

במקרה זה, מתבצע שימוש ב-`c99`, שהינו הכלי שמחבר לממשק של מערכת הקומפליציה הסטנדרטית של C. כמו כן, המילה `wall` מאפשרת לקבל את רוב האזהרות של הקומפיילר. המילה o- מגדירה את שם הקובץ שנרצה להריץ.

טיפוסים ומשתנים

עד כה, בפייתון פשוט כתבנו את המשתנה, ללא התייחסות לטיפוס. למשל:

```
// Python Code
x = 4
x = "I am a string
```

ב-C עלינו להכריז על המשתנים, כיוון שהקומפיילר חייב לדעת מהו **סוג המשתנה**, על מנת שיוכל להגדיר לו את **כמות הזיכרון הנדרשת**.

```
int x;
int x,y;
```

אפשרות אופציונלית היא לאתחל את המשתנה, אך אין חובה שכזאת. מאידך, הדבר עלול לגרום לשגיאות קומפליציה (במידה ולא מאתחלים משתנה וניגשים אליו). – במצב כזה נוצרת שמירת מקום בזיכרון והדבר גורם **ל"ערך זבל"**.

ישנם למעשה 3 סוגים של טיפוסים פשוטים:

- סוגים אריתמטיים.
- מצביעים.
- סוגי ספירה (Enumeration types)

וישנם שני סוגים של משתנים מורכבים:

- מערכים.
- Struct.

משתנים אריתמטיים

ב-C ישנה אפשרות למשתנים בעלי יכולת אריתמטית – כלומר, משתנים שניתן לבצע עליהם למשל את הפעולות הבאות: `+, -, *, <,%`.

הטיפוסים הבסיסיים הינם: `long, short, int, char` ולמספרים ממשיים: `float, double`. דוגמאות לאתחול:

```
char c = 'A';
short s = 0;
int x = 1;
long y = 9;

float x = 0.0;
double y = 1.0; // ~ double precision
```

או בגרסה החיובית:

```
unsigned char c = 'A';
unsigned short s = 0;
unsigned int x = 1;
unsigned long y = 9;
```

הסיבה למשתנים הרבים, קשורה לכמות הזיכרון שמוקצעת להם.

זיכרון

כאמור, ב-C יש חשיבות לזיכרון. הזיכרון הקטן ביותר הוא ביט, שיכול לקבל 0 או 1. בייט הוא קבוצה של 8 ביטים. ייצוג מספרים מתבצע בצורה הבאה. 1 הוא סימון 'לכניסה', כך שלבסוף יש לחבר את כל ה'כניסות' על מנת לקבל את המספר הדרוש:

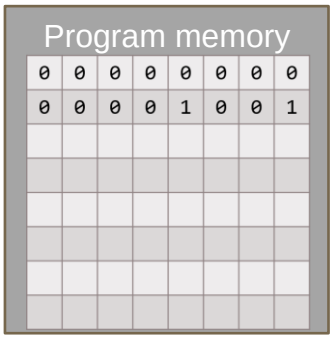
0	1	1	1	0	1	0	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

במקרה זה, המספר הינו unsigned.

נקשר זאת כאשר לזיכרון בתוכנה. אם כתבנו למשל את הקוד הבא:

```
int x = 0;
int y = 9;
```

בזיכרון, זה נראה כך:



בהקשר של המשתנים שראינו קודם לכן, אזי **הגודל** של משתנה הוא מספר הבייטים שהוא תופס בזיכרון, כשכל משתנה יש את **הגודל** שלו ושהוא תלוי מכונה. למשל הגודל של `char` הוא 1. תמיד נזכור כי הגודל של `char` קטן יותר מהגודל של `int` שקטן יותר מהגודל של `long`.

באמצעות **המילה השמורה** `sizeof` נוכל לדעת את הגודל של המשתנה:

```
sizeof(char); // always == 1
sizeof(int); // 4? 8? Machine dependent!
```

נעיר כי גם נוכל להשתמש ב`format`, על מנת להדפיס (במקרה שלנו, נשתמש ב% שמתאר את ה`format` ו-`lu` שמשתמש עבור `long unsigned`):

```
int main(int argc, char* argv[]) {
  printf("sizeof(char): %lu\n", sizeof(char));
  printf("sizeof(short): %lu\n", sizeof(short));
  printf("sizeof(int): %lu\n", sizeof(int));
  printf("sizeof(unsigned int): %lu\n", sizeof(unsigned int));
  printf("sizeof(long): %lu\n",sizeof(long));
  printf("sizeof(unsigned long): %lu\n",sizeof(unsigned long));
  printf("sizeof(double): %lu\n",sizeof(double));
  printf("sizeof(float): %lu\n",sizeof(float));

  return 0;
}
```

התוצאות הינן:

```
sizeof(char): 1
sizeof(short): 2
sizeof(int): 4
sizeof(unsigned int): 4
sizeof(long): 8
sizeof(unsigned long): 8
sizeof(double): 8
sizeof(float): 4
```

המשתנה `char` מעט מבלבל והוא מתבצע באמצעות קוד `ascii` ולכן למעשה ניתן לכתוב את `char` כך:

```
char c = 'A'; //'A' = 01000001
char c = 65; // 65 = 01000001 (same)
```

למעשה, `char` הוא `int` קטן:

```
char c; // uninitialized, garbage value!
  if (c <= 'Z' && c >= 'A' ){
    printf("Capital letter!");
  }
// same as above!
  if (c <= 90 && c >= 65){
    printf("Capital letter!");
  }
```

signed-ו unsigned

אנחנו יכולים להשתמש בטיפוסים שהם רק חיוביים או גם חיוביים וגם שליליים (signed). על מנת להשתמש בסימן, אפשר לקחת ביט אחד מהייצוג ולהשתמש בו עבור סימן. למשל, אם נניח כי sizeof(short) = 2 bytes = 16 bits אז נצטרך 'להוריד' ביט אחד עבור הסימן, וכיוון ש-0 לא משנה בסימן, נוריד 1 ונקבל $2^{15} - 1$.

התחום של ייצוג של unsigned בinterger הוא $[0,2^x - 1]$ ולעומת זאת, עבור signed הוא $[-2^{x-1}, 2^{x-1} - 1]$ כאשר x הוא מספר הביטים של אותו הסוג.

Overflow

Overflow הוא תופעה שמתרחשת במקרה בו 'נגמר לנו המקום'. אם למשל נתבונן ב-char, נקבל כי התחום של char הוא$[-2^7, 2^7 - 1]$ ואילו התחום של unsigned char הוא $[0, 2^x - 1]$. כעת, נתבונן בקוד הבא:

```
unsigned char x = 0;
x = x - 1;
printf("The value of x is %u", x);//255
```

תופעה זו נקראת overflow, כיוון שכביכול אנחנו נמצאים בתחום סגור, והחזרה אחורה, מביאה אותנו לערך הגדול ביותר – שהינו 255 – בדומה למודולו.

דוגמה נוספת לoverflow:

```
char x = 10000; // no error from compiler, undefined run-time behavior
```

ההתנהגות לא מוגדרת בתוכנה – לא ברור כיצד היא אמורה להתנהג וזה תלוי מכונה וקומפיילר.

משתנים וסקופים

ב-C ישנה הפרדה בין חלקי קוד, המתבצעת באמצעות סוגריים מסולסלים {}. ככל שהסקופ יותר פנימי, הוא רואה את כל מי שהוכרז לפניו.

נבחין כי ניתן להגדיר משתנים **גלובליים** (דבר לא מומלץ בדרך כלל) וגם משתנים שמוגדרים בתוך הסקופ:

```
int x = 0; // global
 int main() {
   int x = 1; //local hides global

   {int x = 2;//local hides outer scope
    // x is 2
    }
    // x is 1 again
    }
 }
```

במקרה כאן, מתבצע שימוש בסקופים, כשלמעשה ניתן בכל פעם להגדיר משתנה בשם דומה שדורס את המשתנה הקודם. ניתן לבצע גם שימוש מקונן בסקופים.

נבחין כי באזור הגלובלי כל משתנה מקבל ערך דיפולטיבי, למשל ב-`int`, מתקבל ערך `0`, זאת בשונה מסקופ פנימי יותר, שבו מתקבל ערך זבל. בעקבות כך, כדאי תמיד לבצע השמה.

אופרטורים

כפי שראינו, ישנם מספר אופרטורים בשפה.

ישנם מספר סוגי אופרטורים:

- אופרטורים אריתמטיים:

דוגמאות	משמעות	אופרטור
int x = y + 3;	חיבור	+
int x = y - 3;	חיסור	-
float z = x * y;	כפל	*

float x = 3 / 2; // = 1
float y = 3.0 / 2; // = 1.5
int z = 3.0 / 2; // = 1
	חלוקה	/

int x = 3 % 2;	שארית	%

כדאי להבחין בצורות השונות שמתבצעות בחלוקה:

```
int x = 75;
int y = 100;
float z = x / y; // = 0.0
float z = (float) x / y; // = 0.75 - cast x to float
int z = (float) x / y; // = 0 - truncation to zero
```

פעולות החסרה והוספה:

	x = x + 1;	x++;
prefix	y = x; x = x + 1;	y = x++;
postfix	x = x + 1; y = x;	y = ++x;

נבחין כי Prefix יעיל יותר וניגע בזה בהמשך.

ביטויים

ביטוי של `if`:

```
if (expression) {
  // ... (single statement or block)
}
  else if (expression) {}
    else {
}
  // ..
  }
```

ביטוי של `for`:

```
//for( initial;test:condition;update step)
  int i, j; // in ANSI C you can't declare inside the for loop!
  for (i=0, j=0; (i<10 && j<5); i++, j+=2)
 { // ...
}
```

ביטוי של `while`:

```
while (condition) { // ...
}
```

Const

בדרך כלל, נעדיף שגיאות קומפליציה על פני שגיאות זמן ריצה.

המילה השמורה Const אומרת כי למעשה המשתנה קבוע, ואם נרצה לשנות אותו, נקבל שגיאת קומפליציה:

```
const double E = 2.71828;

E = 10.5; //Compilation error!
```

בנוסף, לא ניתן להגדיר קבוע ללא השמה, אך אכן אפשרי שהמשתנה יוגדר בזמן הריצה:

```
const double E;

E = 10.5; //Compilation error!
```

טיפוסים בוליאנים

אין ביטוי בוליאני ב-C. במקום זאת, נשתמש ב-char או int.

כלומר 0 הוא false וכל מה ששונה מ-0 הוא true.

אמנם, נוכל להשתמש ב-#define. כאשר נגדיר #define TRUE 1 בכל מקום שנקבל TRUE, נקבל 1.

דוגמה לשימוש בטיפוסים בוליאניים:

```
int main()
{
  int a = 5;
  while(1)
  {
    if(!(a-3))
    {
      printf("3");
      break;
    }
    printf("%d", a--);
  }
  return 0;
}
```

פונקציות

דוגמה לפונקציה:

```
int power( int a, int b )
{
  // ...
  return 7;
}
```

בפונקציה יש ערך החזרה ופרמטרים, כשגם במקרה שלהם צריך להחזיר על הסוג.

ישנן גם פונקציות שלא מחזירות כלום:

```
void power( int a, int b )
{
  // ...

  return;
}
```

נדגים גם קריאה לפונקציה מתוך פונקציה:

```
#include <stdio.h>


  int power(int base, int n)
  {
 int p = 1;
 for(int i=0; i < n; ++i)
```

```
  {
  p *= base;
 }
    return p;
}
int main()
{
 int i;
 for(i = 0; i < 10; i++)
  {
  printf("%d %d %d\n",
    i,

  power(2, i),

  power(-3, i));

  }

  return 0;

}
```

נבחין כי פונקציה 'מכירה' רק פונקציות שהוכרזו קודם לכן. למשל:

```
void funcA()
{

  ...
}
    void funcB()
{


    funcC();\\Error
}
} void funcC()
{


  funcB();
}
```

על מנת לפתור את הבעיה הזאת, ניתן להכריז על הפונקציה, ולהגדיר אותה רק לאחר מכאן.

```
  void funcC(int param); //Declaration
   void funcA()
  {
} void funcB()
  {
  funcC(7);

} void funcC(int param) //Definition
{
}
}
const char c = getchar(); //OK
```

Struct

באמצעות המילה השמורה `Struct` נוכל ליצור מבני נתונים מורכבים יותר (דומה במידה רבה למחלקות ואובייקטים). למשל, דוגמה לבניית משתנה של סטודנט, ושימוש בו:

```
// declaration
struct Student
{
  char name[100];
    int id;
  int grade;
};

// use it
struct Student s; // uninitialized
s.id = 14; // access using '.' operator.
s.name[0] = 'a';
```

בחלק הראשון, אנו מגדירים את המשתנים שמכיל המשתנה. בחלק השני, אנחנו מאתחלים אותו, ומכניסים לו את סוגי המידע הרצויים. מדובר בסוג משתנה לכל דבר ועניין, דהיינו אפשר ליצור מצביע למשתנה זו, להעביר אותו לפונקציה, להכניס אותו למערך או להחזיר מתוך פונקציה. כמו כן, אפשר ליצור גם משתנים מקוננים.

בשביל להימנע מכתיבה מיותרת של Sturct בכל פעם שנרצה ליצור משתנה כזה, אפשר לבצע בשתי דרכים

```
// option 1
struct Complex
{

  double _real, _imag;
};
typedef struct Complex Complex;

// option 2
typedef struct Complex
{

  double _real, _imag;

} Complex;

// no need to write "struct Complex" anymore:

Complex addComplex(Complex, Complex);
```

אפשר להעתיק משתנים אלו באמצעות האופרטור '='. דבר זה מעתיק למעשה את כל המשתנים שהוא מכיל. אמנם, איננו יכולים להשוות בין משתנים אלו, ועלינו לכתוב פונקציה

שמשווה ביניהם.

Enum

פקודה שמאפשרת לנו ליצור סט של קבועים. למשל:

```
enum { SUNDAY=1, MONDAY, TUESDAY, …};
enum Color {BLACK,RED,GREEN,YELLOW,BLUE,WHITE=7,GRAY};
enum Seasons
```

```
{
E_WINTER, // = 0 by default
E_SPRING, // = E_WINTER + 1
E_SUMMER, // = E_WINTER + 2   E_AUTUMN  // = E_WINTER + 3
};
}
```

יש יתרון לכך מצד קריאות הקוד ומניעת שגיאות.

זיכרון ומצביעים

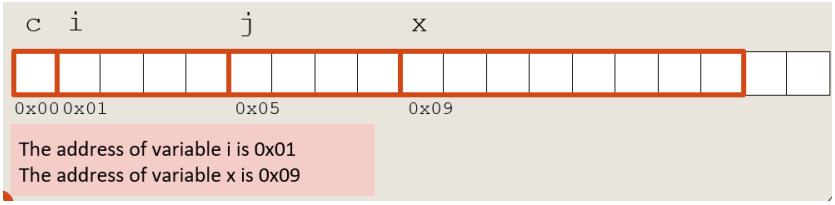
למחשב יש זיכרון, כשכל byte יש כתובת. למעשה, אם נרצה לגשת למשתנה מסוים, נצטרך לגשת לכתובת שלו בזיכרון. נהוג לתאר כתובות במחשב בבסיס 16 (כי מדובר בחזקה של 2). מספרים אלו נקראים בתור מספרים הקסדמיצליים, ונהוג לסמן אותם עם 0x.

בעיקר נתעסק ב**זיכרון מקומי**, שמתנהג כמו **מחסנית**.

נתבונן למשל בפונקציה הבאה:

```
int main()

{

char c;

int i,j; //suppose that sizeof(int) = 4

  double x; //suppose that sizeof(double) = 8

}
```

האחסון בזיכרון מתבצע בצורה הבאה:



כעת נוכל גם לדבר על הנושא הבא – מערכים.

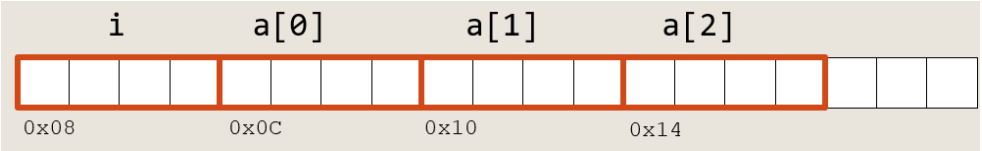
מערכים

נגדיר תוכנית שמייצרת מערכים:

```
int main()
{

int i;
```

```
int a[3]; //sizeof(a) = 3 * sizeof(int)
}
```

בזיכרון, זה נראה כך:



בעקבות כך, ברור מדוע חייבים לדעת את גודל המערך בזיכרון מראש.

נראה דוגמאות לכך:

```
int arr[3] = {3, 4, 5}; // OK
int arr[] = {3, 4, 5}; // OK: equivalent
int arr[3];  // undefined values
arr = {2,5,7}; // Bad: array assignment only in initialization
int arr[3] = {0};     // Init all items to 0, takes O(n)
int arr[4] = {1}; // Bad style - {1, 0, 0, 0}
int arr[2] = {3, 4, 5}; // Bad
int arr[2][3] = {{2,5,7},{4,6,7}}; // OK
int arr[2][3] = {2,5,7,4,6,7};    // OK: equivalent
int arr[3][2] = {{2,5,7},{4,6,7}}; // Bad
```

כיצד הכתובת בזיכרון מחושבת?

```
arr[0]  32+0*sizeof(int) = 32
arr[3]  32+3*sizeof(int) = 44
arr[i]  32+i*sizeof(int) = 32 + 4*i
arr[-1] 32+(-1)*sizeof(int) = 28 // can be the code
      // segment or other variables
```

הביטוי האחרון מהווה שגיאת זמן ריצה, שעלולה להוביל לשגיאה בלתי ידועה – זאת אחריותו של המשתמש לבדוק האם מדובר בבעיית אינדיקציה.

בעקבות הדרך בה בנויים המערכים בשפה, לא ניתן לבצע את הפעולות הבאות:

```
int a[4]; int b[4];

a = b; // illegal

// and how about:
if( a == b ) // legal, address comparison
```

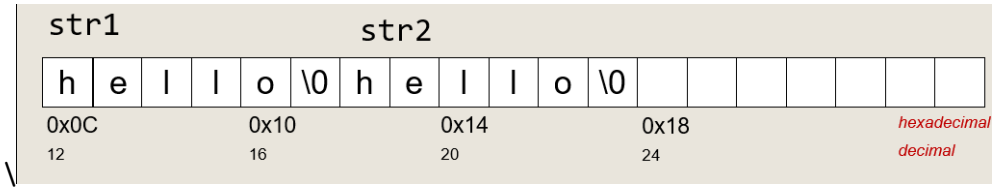
כלומר, בביטוי האחרון יש השוואה בין הכתובות בזיכרון.

נבחין כי `string` מאוחסן בזיכרון כמערך של `char`:

```
int main()
{
```

```
  char str1[] = "hello";

  char str2[6] = "hello";
}
```

שמאוכסן בזיכרון כך:



משמעות התו `\0` היא למעשה להפריד בין ה-`String` השונים. בקוד הבא עלולה להיווצר שגיאה כחוסר בכתיבת ה-`0`:

```
int main()
{
char str[] = {'c', 'o', 'd', 'e'};
char ch = 'a';
printf("%s", str);

  return 0;
}
```

כדאי להבחין כי תו זה אכן תופס מקום בזיכרון.

מצביעים

מדוע שנצטרך מצביעים? מוטיבציה לכך נוכל לקבל בתוכנית הבאה:

```
void swap(int a, int b)
{

int temp = a;

  a = b;

b = temp;
}
int main()
{

int x, y;

  x = 3; y = 7;

  swap(x, y);

  // now x==?, y==?
}
```

הפונקציה הזאת לא תעשה דבר. כאשר אנחנו קוראים לפונקציה, אנחנו לא מעבירים את הארגומנטים עצמם אלא **עותקים** שלהם. על מנת שנרצה להחליף ביניהם נוכל להשתמש ב**פוינטרים-מצביעים**. מדובר בטיפוס מידע לכל דבר ועניין, אלא שהם מסוגלים להחזיק **כתובות** בזיכרון.

```
int x = 3;
//variable that stores the value (3)

int *p = &x
//variable that stores an address  a pointer
```

אם נדפיס את p, נקבל את הכתובת בזיכרון של המשתנה x. דבר זה מתבצע באמצעות התו &.

מקרה בוחן נוסף:

```
int *p = &y;
int z = *p;
*p = x;
```

בשורה הראשונה אנו **מכריזים** על p בתור פוינטר לערך int.

בשורה השנייה, נגדיר את z להיות הערך p. כלומר כמו לומר z=y. אפשר לשים לב כי * משתמש בתפקידים שונים – האחד, הגדרת הטיפוס, השני ביצוע פירוק – dereference.

הגודל של המצביע נשאר **קבוע לכל סוגי המצביעים**. כאשר נשלח מערך לפונקציה, הוא יתקבל כ**מצביע** ולכן נצטרך להעביר גם את **הגודל** של המערך. נוכל גם באופן כללי להתייחס למערך כמו אל מצביע לתא הרלוונטי במערך.

חשוב להבחין כי בשונה ממשתנים אחרים, בהכרזה של שני המשתנים, ה* מתייחסת למשתנה ספציפי. כלומר:

```
int *p, q; // p is a pointer to an int

      // q is an int
```

כעת, באמצעות מצביעים נוכל לבצע את מה שרצינו בפונקציה swap.

בהקשר של Const, נבחין כי הקבוע 'שומר' על מי שנמצא משמאלו, אם אין אף אחד כזה, הוא שומר על זה שנמצא מימינו.

כעת, נוכל

המצביע NULL

המצביע NULL הוא מצביע מיוחד, בעל ערך 0. פעמים רבות ייתכן שכאשר נבצע השמה של מצביע, הוא יהיה NULL ולכן כדאי לבדוק זאת בתחילת כל שימוש במצביעים.

למצביע זה בלתי אפשרי לעשות dereference וזהו עלול לגרום לשגיאות זמן ריצה.

פעולות על מצביעים

על מנת לבצע פעולות על מצביעים, נוכל ללכת בשני דרכים. בדרך הראשונה, להוסיף **מספר** למצביע שלמעשה 'יזיז' את **הכתובת** בזיכרון. בדרך השנייה, אם נבצע ptr++ אזי נקדם את

המצביע מספר מקומות ביחס לסוג המשתנה עליו הוא עובד. למשל, עבור `int`, נקדם ב-4 מקומות.

כאשר נבצע **החסרה** בין מצביעים, נקבל למעשה את ההחסרה בין המקומות שלהם בזיכרון, כלומר, למשל בדוגמה הבאה:

```
int* p = 100;
int* q = 92;
printf("%d\n", p-q);
```

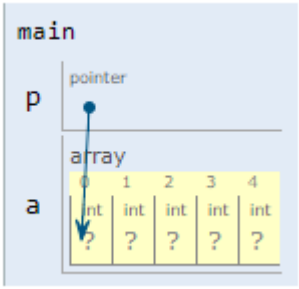
יודפס 2 ולא 8 כמו שהיינו חושבים.

כאשר נבצע ++ משמאל למשתנה, ואז dereference, קודם כל יתבצע **הקידום** ורק לאחר מכן הdereference, אם מתבצע בצורה הפוכה, אז קודם כל יש dereference ורק לאחר מכן הקידום.

מצביעים ומערכים

מה הקשר בין מצביעים ובין מערכים? באמצעות שם המערך להשים מצביע לערך הראשון בו. כך למשל:

```
int *p = NULL;
int a[5];
p = a;     // same as p = &a[0];
```



מצד שני, נוכל גם לבצע את הפעולה הבאה:

```
p[1] = 102;  // same as *(p+1)=102;

*(a+1) = 102; // same as prev. line
```

כאן החלק המבלבל. `p` הוא מצביע למערך, ולכן כאשר נבצע את השורה הראשונה, ניגע למעשה למקום הראשון במערך, ונבצע לו dereference.

העברה של מצביעים ומערכים לפונקציות

כל הביטויים הבאים למעשה מציינים ביטוי אחד – העברה של מצביע ל-`int`:

```
int foo( int *p );
int foo( int a[] );
int foo( int a[NUM] );
```

ניתן גם לגשת למקום מסוים במערך של Struct באמצעות:

p_x->_a[2] = 3; // same (כאשר p_x הוא הפוינטר).

מצביעים לפונקציות

בחלק משפות התכנות, פונקציות הן ממש משתנים. במקרה שלנו, פונקציה היא איננה משתנה, אבל `c` מאפשרת לנו להגדיר מצביע לפונקציה.

דבר זה ייעודי למשל למצבים בהם נרצה לבחור מספר פונקציות ולבחור את המקסימילי מבין ההפעלות עליהן.

הסינטקס של מצביע לפונקציה ייראה כך:

```
void (*fun_ptr)(int) = &fun;
```

מדובר במצביע לפונקציה שמחזירה `void` ומקבלת `int`.

עוד קצת סינטקס:

```
int fun(int x) { ... }  int main()
{
// fun_ptr is a pointer to a function that
// returns an int and receives an int
   int (*fun_ptr)(int);
    fun_ptr = &fun; // fun_ptr points to fun

fun_ptr = fun; // same
    int x = (*fun_ptr)(7); // same as x = fun(7)

int y = fun_ptr(7);     // same as y = fun(7)
}
```

חשוב לציין שאם נשים מצביע למצביע או נבצע dereference לפונקצייה, לא יקרה דבר, וכך גם לגבי שם הפונקציה.

חוק right-left

1. חפש את שם הפונקציה.
2. תתבונן בסימן מימין לשם:
- אם יש `()` מדובר בפונקציה.
- אם יש `[]` מדובר במערך.
3. תתבונן בסימנים משמאל לשם.
4. תחזור על צעד 2 עד שההכרזה מסתיימת.

סוגריים משנות את הסדר הטבעי.

הקצאת זיכרון דינמי

כפי שאמרנו, בשפת c עלינו להגדיר בזמן הקומפילציה את כמות הזיכרון הנדרשת. אם כך, מה יהיה במקרה בו נרצה להגדיר בזמן ריצה גודל של מערך, למשל על ידי בקשה מהמשתמש? כיצד אפשר לבצע זאת בצורה דינמית?

פתרון אפשרי לכאורה הוא הגדרת גודל מקסימלי של מערך, כך שתמיד נקצה את כמות הזיכרון הזו בזמן קומפילציה, אך דבר זה עלול לגרום לבזבוז זיכרון מיותר. לכן נשתמש ברעיון של 'ערימה דינמית'. ברעיון זה, הזיכרון מוקצה בזמן הריצה, כשהמתכנת כמה להקצות ומתי. ההקצאה תלויה בפעולות זמן הריצה, ומתאפשרת לפי הזיכרון במחשב הספציפי.

על מנת להקצות זיכרון, נכיר את הפעולה הבאה:

```
void *malloc( size_t Size );
```

פונקציה זו, מחזירה מצביע מגודל מסוים (בבייטים), או NULL במידה ואין אפשרות להקצאת זיכרון כזו. כיוון שלעיתים רבות לא נוכל לקבל את הזיכרון הדרוש, חובה עלינו לבדוק בכל פעם האם המצביע הינו NULL.

אפשר להשתמש גם בפונקציה הבאה:

```
void *calloc( size_t numOfCells, size_t sizeOfEachCell);
```

פונקציה זו מקצה זיכרון בצורה שונה. היא דורשת את כמות הבלוקים וכמה 'המשקל' של כל בלוק. פונקציה זו ממלאת את המקום בזיכרון באפסים.

פעולה נוספת שקיימת היא realloc():

```
void* realloc(void *ptr, size_t new_size);
```

פעולה זו מגדילה את ההקצאה מהכמות ההתחלתית בהתאמה. אם נכניס מצביע NULL זו פעולה שקולה ל-malloc.

כך נבצע את הקצאת הזיכרון:

```
int main(int argc, char *argv[])
{
 int num_complex;

ReadN(&num_complex);

Complex *complex_ptr =
    (Complex *) malloc(num_complex * sizeof(Complex));
 complex_ptr[1].real = 10;
    // access the second element  //
```

חשוב להדגיש כי יש להוסיף `#include <stdlib.h>`.

אם נרצה 'לשחרר' את הזיכרון, נשתמש בפונקציה `free(void *p)`. בדומה לפתיחת וסגירת קובץ, הרגל טוב הוא אחרי הקצאת זיכרון לשחרר אותו. חייבים להכניס לפונקציה זו את הכתובת המדויקת שהוקצתה.

בכל הקצאת זיכרון, עלינו לוודא שלא התקבל מצביע `NULL`. כמו כן, אחרי שחרור זיכרון, מומלץ להפוך את המצביע ל-`NULL`. אין צורך לעשות זאת אם מדובר במשתנה מקומי, שכאן הוא נמחק מיד השחרור.

שגיאת סגמנטציה

מקבלים שגיאת סגמטציה כאשר מנסים לגשת לזיכרון שאין לנו גישה אליו.

מחסנית ועוד

מחסנית היא המנהלת את הזיכרון במהלך קריאת לפונקציות. למשתנים במחסנית זה זמן חיים מוגבל, והגודל שלהם מוגדר במהלך קומפילציה.

אין להחזיר את הכתובת של משתנה שהוא מקומי, אבל מותר להחזיר את המשתנה עצמו, כיוון שהכתובת 'נעלמת' לאחר ההפעלה של הפונקצייה.

חשוב לשים לב שגם למחסנית זו יש זיכרון מוגבל, ולאחר הפעלה של לולאה אינסופית רקורסיבית למשל, עלול להיווצר stack-overflow.

משתנים גלובליים נשמרים ב-data memory segment.

מחרוזות סטטיות ודינמיות

כאשר משתמשים במרכאות כפולות, אף בתוך פונקציה, דבר זה מייצר זיכרון סטטי (כמו `mutable` בפייתון).

למשל, כמו בדוגמה הבאה:

```
char* msg = "text"; char msg2[] = "text";
msg[0]= 'n';  // seg fault - trying to change what is written in the
read-only part of the memory
 msg2[0]= 'n';   // ok - changing what is written in the stack part
of the memory
```

על מנת להעתיק טקסט, כדאי להשתמש בפונקציית `strcpy`:

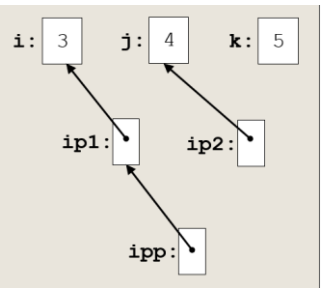
```
#include <string.h>
char *strcpy(char *restrict dest,
       const char *src);
// don't worry about 'restrict'
```

מצביעים למצביעים

נתבונן בקטע הקוד הבאה:

```
int i=3;
int j=4;
int k=5;

int *ip1 = &i;
int *ip2 = &j;
int **ipp = &ip1;
ipp = &ip2;
```

מבחינה ויזואלית, כך מתרחש:



מה המוטיבציה לכך? כמו שכשרצינו לשנות ערך של משתנה מסוים דרך פונקציה שלחנו לו את המצביע, כך אם נרצה לשנות מצביע מסוים (ולא להחזיר אותו), נצטרך לתת מצביע למצביע. כלומר, המצביע הוא משתנה לכל דבר בהקשר זה.

מערכים דו-מימדיים

ניתן להגדיר מערכים דו מימדיים במספר דרכים. אפשר לבצע זאת בצורה סטטית, באמצעות הגדרת גדול המערך בזמן **קומפילציה** (יעיל יותר), בצורה חצי **דינמית** – מספור השורות ידוע בזמן קומפליציה, או **בצורה דינמית לגמרי**.

הצורה הסטטית נראית כך:

```
int arr[5][7]; // 5 rows, 7 columns
```

הצורה החצי דינמית:

```
int *arr[5]; // array of 5 pointers to int
```

והצורה הדינמית:

```
int **arr; // pointer to pointer to int
```

הצורה הסטטית היא הבטוחה והיעילה ביותר, אבל לא תמיד ניתן להשתמש בה. במצב כזה, גודל הזיכרון הדרוש ייקבע בזמן קומפליציה.

כך תתבצע הצורה החצי דינמית (שימו לב שעלינו להקצות זיכרון, במקרה שלנו k=8):

```
int *pa[4]; // allocates memory for 4 pointers
for (int i=0; i<4; i++)
{
  pa[i] = malloc(k*sizeof(int)); // k=8
}
```

במקרה זה, מספר השורות צריך להיות ידוע בזמן קומפילציה (שימו לב שייתכן וה'עמודות' יהיו שונות בכל שורה).

על מנת לבצע מערך בצורה הדינמית המלאה, נצטרך קודם כל להקצות את מספר השורות בזיכרון, ורק לאחר מכן להקצות את העמודות:

```
int ** arr;
arr = (int**)malloc(k*sizeof(int*)); // k=4
for (i=0; i<4; i++)
{

  arr[i] = (int*)malloc(j*sizeof(int)); // j=8
}
```

צריך לזכור במקרה זה לשחרר את כל הזיכרון. למשל:

```
for (i = 0; i < nrows; i++)
{

  free(array[i]);
}

free(array);
array = NULL;
```

ניתן להשתמש בדרך נוספת לגישה למידע – דרך יעילה יותר.

במקום להקצות זיכרון למצביע למצביע, אפשר להגדיר פשוט זיכרון למצביע כגודל המקום שאנו רוצים. למשל:

```
int *arr =(int*)malloc(5*7*sizeof(int))
```

הגישה לתא בזיכרון תתבצע בצורה הבאה - **arr[i][j] -> arr[i*ncols + j]**.

אמנם הגישה מהירה והמימוש פשוט יותר, אבל מדובר בקוד פחות קריא.

עקרונות נוספים בתכנות ב-C

תהליך הקומפילציה

ב-C תהליך הקומפילציה יותר מורכב משפות אחרות. שלב הקומפילציה הופך את הטקסט לפקודות מכונה שהמחשב יודע לבצע. דבר זה מתרחש בשלושה שלבים: ראשית, מתקיים שלב קדם-המעבד, לאחר מכאן שלב הקומפיילר ואז שלב הLinker.

שלב הקדם-מעבד - Preprocessor **איננו** חלק מהקומפליציה.

תחילה, נקדים ונתייחס למספר תכונות שקיימת בקוד:

#include

הפקודה `#include` הינה למעשה 'העתקה של קובץ header לתוך הטקסט שלנו.

קובץ header הוא למעשה קובץ שמכיל הכרזות על פונקציות וקבועים שמשותפים עם המשתמשים האחרים.

#define

הפקודה #define מגדירה למעשה 'מקרואים'. ניתן להשתמש בהם גם על מנת להגדיר 'פסאודו-פונקציות'. למה לא מדובר באמת בפונקציות? כי זה רק עיבוד טקסט. השלכה לכך יכולה להיות כאן, למשל:

```
#define SQUARE(x) x*x
#define PLUS(x) x+x
b = SQUARE(a+1);
c = PLUS(a)*5;
We actually get the following:
b = a+1*a+1; // b = 2*a + 1
c = a+a*5;  // c = 6*
```

#if

הפקודה #if מאפשרת לנו לבצע דיבוג בתנאים מסוימים. כביכול אנחנו אומרים לקומפיילר לקמפל דבר מסוים בתנאי שדבר אחד מתקיים. למשל, נוכל להגדיר דיבוג בצורה הבאה:

```
#define DEBUG
#ifdef DEBUG
// compiles only when DEBUG exists (defined)

printf("X = %d\n", X);
#endif
```

כלומר, אנחנו מבצעים את הפעולה הבאה רק אם מוגדר `Debug`.

על מנת לבטל את הדיבוג, נגדיר `#DNDEBUG`בתחילת התוכנית.

שלב הקדם-מעבד - Preprocessor

בקדם- המעבד, התוכנית מבצעת את פעולות תרגום הטקסט הבאות:

1. עיבוד קבצי header והעתקתם לקובץ.
2. פתיחת **מאקרואים**.
3. ביצוע 'קומפילציה על תנאי' – אם התנאי לא מתקיים, מוחק את הקוד.
4. הורדת הערות.

התוצאה הינה קוד שמוכן לקמפול.. נזכיר כי ניתן גם להשתמש במקרואים, שהינם סוג של פונקציות, למשל #define SQUARE(x) x*x. האלטנרטיבות הם קבועים או פונקציות ממש.

דרך הקדם-מעבד אפשר לבצע #if – קודם לבצע קומפליציה רק בתנאים מסוימים. התיקייה assert משתמשת בדרך זו, כפי שניתן לראות:

```
#include <assert.h>
 //
// Sqrt(x): compute square root of x
// Assumption: x is non-negative
double sqrt(double x )
{
  assert(x >= 0);
 // aborts if x < 0   …
}
```

מה קורה בשני השלבים הבאים?

שלב הקומפילציה

בשלב הקומפליציה כל קבצי הc הופכים להיות קבצי o - קובץ מסוג object. בשלב הקומפליציה

ייתכן ונרצה לעבוד עם כמה קבצים, שכן דבר זה גורם לתוכנה להיות מודולרית וקריאה יותר. לאחר מכן צריך לעשות `#include` שיכיל את הקבצים. נבחין כי את ההכרזות נבצע בקובץ `h` ואת המימושים בקובץ `c`.

ברמה האופרטיבית מבצעים זאת כך:

```
$ gcc -c -Wall stack.c -o stack.o

$ gcc -c -Wall main.c -o main.o
```

Linker ושגיאותיו

הLinker מאחד בין כל קבצי הobject לקובץ executable, אותו ניתן להריץ. ניתן לראות תהליך זה כאן:

```
$ gcc -c -Wall stack.c -o stack.o

$ gcc -c -Wall main.c -o main.o
```

```
$ gcc stack.o main.o libc.a libm.a -o stack
```

השימוש ב-c מונע הפעלת תהליך הlinker מראש. השורה האחרונה מבצעת את תהליך הlinker.

ייתכן ויהיו בעיות לLinker – אם הוא לא מוצא את הפונקציות למשל:

```
" Main.o(.text+0x2c):Main.c: undefined reference to `foo "
```

או אם יש יותר מדי מימושים:

```
.foo.o(.text+0x0):foo.c: multiple definition of `foo'
```

על מנת למנוע מצב בו נבצע #include פעמיים לאותו קובץ ואז למעשה נשפוך את תוכן הקובץ פעמיים. נצטרך להשתמש בקדם-המעבד ולעטוף את קובץ ה-h כך:

```
#ifndef COMPLEX_H
#define COMPLEX_H
```

כלומר, הוא למעשה מוודא האם כבר הקובץ הוגדר, או לא.

ספריות

ישנם שני סוגים של ספריות:

1. ספריות סטטיות:
   a. מקושרת עם הקובץ executable בזמן הקומפליציה.
   b. בלי תלות במיקום ובנוכחות של הספריות.
   c. בלתי תלות בגרסאות של הספריות.
   d. לדוגמא: `#include <utils.h>`
2. ספריות משותפות. נטענות בזמן ריצה:
   a. קובץ executable קטן יותר.
   b. מספר תהליכים פועלים על אותו קוד.
   c. לא צריך לבצע קומפליציה מחדש כאשר הספריות משתנות.

על מנת לייצר ספרייה סטטית, מספיק לכתוב למשל:

```
ar rcs libutils.a data.o stack.o list.o
```

המילה ar היא למעשה סוג של קובץ archive.

על מנת להריץ ספריה דינמית:

```
gcc -Wall -fPIC -c utils.c

gcc -shared utils.o -o libutils.so
```

עיצוב תוכנה

ממשקים הם למעשה אוסף של פונקציות שמספקות מודל קוהרנטי, אך מסתיר את המימוש ואת התוכן. דבר זה מאפשר מודלוריות עם מודלים אחרים.

כך למשל אם נרצה ליצור רשימה מקושרת של `string` אז נוכל להכניס אותה לתוך `Struct` כללי, שאותו נכנס לתוך מבנה הנתונים הקיים. דבר זה מאפשר לנו להרחיב את הממשקים הקיימים, שאינם מתחשבים בתוכן הרלוונטי או בדרך המימוש.

תכנות גנרי

מבני נתונים גנריים אלו מבני נתונים שיכולים להחזיק כל סוג מידע. הסוג הספציפי נוצר בזמן הריצה עצמה. הדרך שלנו לממש מצביע למשתנה גנרי בשפת `c` הינו `void*`.

לפני שנעסוק במבני נתונים אלו, נראה קודם לכן את הפונקציה `memcpy`.

חתימת הפונקציה:

```
void *memcpy(void *destination,
             const void *source,
             size_t num);
```

פונקציה זו מעתיקה בלוק של זיכרון בגודל ספציפי למקום אחר, כשהיא **איננה** יודעת מה הסוג שלו. האתגרים המרכזיים הם כיצד אפשר לבצע איטרציה על `void*` וכיצד אפשר לבצע עליו `dereference`.

כאשר נרצה לממש את פונקציה זו, נצטרך קודם כל לבצע cast לסוג הרצוי.

באמצעות התכנות הגנרי נוכל לממש למשל רשימות מקשורות גנריות, עץ בינאריים גנריים ועוד מבני נתונים, שלא יגבילו אותנו לסוג הספציפי בו אנו מעוניינים כרגע. באמצעות memecpy נוכל להעתיק את המידע מבלוק אחד לאחר בצורה גנרית.

תכנות גנרי באמצעות מצביעים לפונקציות

נוכל דרך מצביעים לפונקציות לממש חלק מהפונקציות בצורה גנרית.

למשל, נוכל ליצור פונקציה שהינה `comparator` שמשווה בין אלמנטים בצורה הרצויה לנו, ולהכניס אותה לפונקצייה המובנת ב-`c` שנקראת:

```
void qsort(
        void* base, size_t n, size_t size,    int (*compare)(void
const*, void const*)
);
```

דיבוג וטיפול בשגיאות

נזכיר כי אחת הדרכים שראינו שמאפשרת לנו לדבג היא באמת `#ifdef` ו-`#Debug`.

ניתן לבצע דיבוג גם באמצעות `assert`.

קודם כל, עלינו לייבא את התיקייה `#include <assert.h>`. ולאחר מכן לכתוב את שורת הקוד הבאה `assert(x >= 0); // aborts if x < 0` למשל. דבר זה מאפשר לנו לתפוס את השגיאה בזמן הדיבוג.

על מנת לטפל בשגיאות, ניתן להשתמש בספרייה `stderr`. הפונקציות הרלוונטיות מבחינתנו הן:

```
fprintf(stderr, "format string", …)
perror
strerror (with errno)
```

הפונקציה הראשונה מדפיסה הודעת שגיאה ל-stderr. האחרות, מדפיסות הודעת שגיאה או מחפשות הודעת שגיאה מתאימה לפי הerrnum.

הקונבנציות הן להחזיר `0` במצב של הצלחה ו-`NULL` או מספר שונה מ-0 (בדרך כלל 1 או -1) במקרה של כשלון.

נגדים כעת שימוש ב`errno`:

```
#include <stdio.h>
#include <errno.h> // for the global variable  errno
#include <string.h> // for strerror
int main( int argc, char **argv )
{
  FILE *fp;
fp = fopen("file.txt", "r");
if( fp == NULL ) {
printf("Error: %s\n", strerror(errno));
}
return 0;
}
```

תחילה כללנו את התיקייה, ולאחר מכאן במצב של שגיאה הדפסנו את errno (מדובר בסוג של משתנה שעבור פונקציות שונות מאותחל כאשר הן מזהות שגיאה).

תנאי טרנרי ב-`c`

נוכל להגדיר תנאי משולש:

```
variable = Expression1 ? Expression2 : Expression3
```

ששקול למעשה לביטוי:

```
if(Expression1)
{
variable = Expression2;
```

```
} else {
variable = Expression3;
}
```

משתנים קבועים – static ו-extern

למילה `static` יש שתי משמעויות:

1. מחוץ לפונקציה – משתנים ופונקציות סטטיות נראות רק בקובץ הספציפי ולא גלובלית.
2. בתוך פונקציה- לוקאליות, מאותחלים פעם אחת בתוכנית, גם עם הפעלה חדשה של הפונקציה.

למעשה, אם נשתמש במספר קבצים, פונקציה סטטית תהיה זמינה רק במודל הנוכחי ולא במודל אחר, זאת בשונה מהמילה `extern` שתאפשר זאת, כמו גם הצגת המשתנה בקבצים אחרים.

משמעות המילה השמורה `extern` היא לומר "המשתנה הזה מוגדר איפושהו", כשבדרך כלל זה קורה בקובץ אחר. למשל, כמו בדוגמה הבאה:

```
file1.h
// declare only
extern float pi;

file1.c
#include <file1.h>
// define once only
float pi=3.141;

file2.c

#include <file1.h>
// use through file1.h
float R = 10.0;
float C = 2 * pi * R;
```

פעולות אופרטריות על הביטים

שפת `c` מאפשרת לנו לבצע פעולות אופרטריות על ההצגה הבינארית של משתנה:

פעולה	סימן
and	&
or	|
xor	^
One complement (not)	~
Left shift	<<
Right shift	>>

דבר זה נעשה באמצעות טבלת אמת. כך למשל אם נבצע פעולה "וגם" על שני מספרים חיוביים, נקבל את המשותף ביניהם בהצגה הבינארית. נניח:

```
14 & 11 =   ?  // decimal representation
0b1110 & 0b1011 = 0b1010  // binary representation

14 & 11 =   10  // decimal representation
int x = 14;
int y = 11;
int z = x & y; // z = 10
```

משמעות ביצוע shift-left למשל היא 'הכפלה ב2', ונוכל לבצע זאת גם בשפה:

```
0b001101 << 1 == 0b011010,
0b001101 << 2 == 0b110100
```

נוכל גם לקבל את הערך של ביט כלשהו בהצגה הבינארית (כביכול האם הנורה דלוקה או לא) וגם להזיז אותו. נדגים את קבלת הערך.:

```
int isZero (int variable, int position)
{
    return (variable & (1 << position)) == 0;
}
```

נוכל גם לעצב ולארגן שדות שיהיו בגודל ספציפי של ביטים.

למשל, אם נרצה `Struct` של משתנים בוליאנים:

```
struct {
    unsigned int widthValidated;
    unsigned int heightValidated;
} status;
```

דבר זה דורש לפחות 8 ביטים. אבל למעשה אנחנו הולכים להשתמש רק בביט אחד בכל פעם (0 או 1). ולכן נוכל לקבוע מראש בכמה ביטים נשתמש. למשל:

```
struct {
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status;
```

שימו לב כי אם נעבור את סף ה32 ביטים, כלומר מבחינת זיכרון של 4 בייטים, אזי נעבור באופן אוטומטי ל8 בייטים.

Volatile

לפעמים משתנים יכולים להשתנות חיצונית, כלומר לא מתוך הקוד הנוכחי, אך הקומפיילר לא יודע זאת, ומבחינתנו אם נשים למשל משתנה זה בתנאי (ששונה מערך אחר), זה יכול להיות שקול ללולאה אינסופית. ניתן, באמצעות המילה השמורה `Volatile`, לגרום לקומפיילר לא להתייחס לביטוי זה כביטוי אמת.

Restrict

ניתן להשתמש במילה השמורה Restrict בהכרזות על מצביעים. זה לא מוסיף פונקציונליות על המצביע, אלא למעשה אומר לקומפיילר "זאת הדרך היחידה לגשת לאובייקט הזה" (או באמצעות מצביע המתקשר למצביע הנוכחי. נגיד הוספת 1 למצביע הנוכחי). דבר זה מאפשר אופטימיזציה של הקמפול.

Switch

באמצעות המילה השמורה Switch ניתן לבצע מספר תנאים על תו כלשהוא:

```
switch (n)
{
case 1: // code to be executed if n = 1;
break;
case 2: // code to be executed if n = 2;
break;
default: // code to be executed if n doesn't match any cases
}
```

VLA

מדובר בראשי תיבות של הביטוי **Variable-Length Array**. הוא מאפשר לקבוע גודל מערך בזמן ריצה ולא באמצעות malloc. נעדיף שלא להשתמש בו כיוון שאין דרך להתמודד עם שגיאות, וכיוון שאיננו יודעים היכן מאוחסן הזיכרון, או לשחרר אותו או להשתמש ב-realloc. מעבר לכך, הוא לא תומך בספרייה הסטנדרטית של c++.

Union

באמצעות המילה השמורה Union ניתן לשמור סוגי מידע **שונים** באותו מקום בזיכרון. דבר זה מאפשר דרך יעילה של שימוש בזיכרון למטרות שונות. הגודל של ה-Union יכול להכיל את המשתנה הגדול ביותר. כל משתנה מקבל הקצאת זיכרון כאילו הוא המשתנה היחיד בStruct.

נגדיר זאת כך:

```
typedef union u_all
{

   int i_val;

   double d_val;

} u_all
u_all u; //definition of the variable u
u.i_val= 3;//assignment to the int member of u
printf("%d\n", u.i_val);
u.d_val= 3.22; //this corrupts the previous                assignment

printf("%f\n", u.d_val);
```

באחריותנו לוודא איזה משתנה הינו פעיל ברגע זה.

אפשר להשתמש ב-Union למשל עבור ניתוח מילולי של משפטים – פירוק רצף של מילים לאותיות.

נספחים

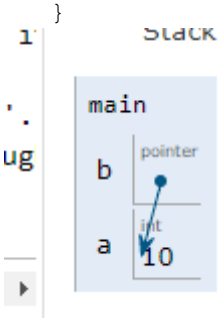
סיכומים נוספים

מצביעים

בואו וניקח סצינה מסוימת. נגיד ויש לנו לוקר A שהוא נעול, ולוקר B שהוא פתוח. המפתח ללוקר A נמצא בלוקר B ואפשר לקחת את המפתח מלוקר B ובדרך זו לשנות את האלמנטים בלוקר A. זה למעשה מה שמתרחש במצביעים.

נתבונן בדוגמת קוד שתסביר את העניין:

```
int main()
{
  int *b; //1 declaring pointer but not assigning it anything
  int a = 10; //2 declaring the int variable and assigning it a
value;
  printf("%d\n", a); //3 Scenario 1
  b = &a; //4 assigning the pointer 'b' the location of 'a'.
  printf("%d\n", *b); //5 viewing the contents of 'a' through 'b'.
  *b = 20; //6 changing the contents of 'a' through 'b'.
}
```



בשורה הראשונה, אנחנו מכריזים על פוינטר מסוג int. כלומר, מדובר בסוג חדש של משתנה, שמצביע על מיקום בזיכרון. בשורה השנייה, אנחנו מכריזים על משתנה רגיל. בשורה השלישית, באמצעות הסימן & אנחנו מבצעים השמה ל**מיקום** של a בזיכרון. כלומר, כרגע b מצביע על המיקום של a בזיכרון. בשורה האחרונה, אנחנו משנים את התוכן של a דרך המצביע b.

אפשר לראות שהסמן * משמש בשני תפקידים. בתחילה, אנחנו משתמשים בו על מנת להכריז על המצביע b. בהמשך (בשורה האחרונה) אנחנו משתמשים בו על מנת 'לפרק' את b, כלומר לקחת את התוכן שהכתובת בזיכרון מצביעה עליו, ולשנות אותה.