

## Problems with Pointers

### C's Dynamic Allocation Functions

C/C++ מצביעים מספקים תמיכה הכרחית למערכת ההקצאה הדינמית של הקצאה דינמית היא האמצעי שבאמצעותו תוכנית יכולה להשיג זיכרון בזמן שהיא פועלת.

כפי שאתה יודע, למשתנים גלובליים מוקצים אחסון בזמן הקומפילציה. משתנים מקומיים משתמשים במחסנית. עם זאת, לא ניתן להוסיף משתנים גלובליים או מקומיים במהלך הפעלת התוכנית. עם זאת, יהיו זמנים שבהם לא ניתן לדעת מראש את צורכי האחסון של תוכנית. לדוגמה, תוכנית עשויה להשתמש במבנה נתונים דינמי, כגון רשימה מקושרת או עץ בינארי. מבנים כאלה הם מטבעם דינמיים באופיים, גדלים או מתכווצים לפי הצורך. כדי ליישם מבנה נתונים כזה, נדרשת שתוכנית תהיה מסוגלת להקצות ולפנות זיכרון.

תומך למעשה בשתי מערכות הקצאה דינמיות מלאות: זו המוגדרת על C++ מכילה מספר C++-המערכת הספציפית ל C++-וזו הספציפית ל C ידי גישה זו נידונה בחלק השני. כאן, C, שיפורים ביחס לזו שבה משתמש C. מתוארות פונקציות ההקצאה הדינמית של מתקבל מהערימה C זיכרון שהוקצה על ידי פונקציות ההקצאה הדינמית של - אזור הזיכרון הפנוי שנמצא בין התוכנית שלך לבין אזור האחסון הקבוע שלה והמחסנית. למרות שגודל הערימה אינו ידוע, הוא בדרך כלל מכיל כמות די גדולה של זיכרון פנוי. ( ) free( ) ו malloc( ) מורכבת מהפונקציות C הליבה של מערכת ההקצאה של רוב המהדרים מספקים עוד כמה פונקציות הקצאה דינמית, אבל שתי אלה.) (הם החשובים ביותר

(A C++ program may also use the C++-style header <cstdlib>.)

The malloc( ) function has this prototype: יש אב טיפוס זה:

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

(The type size\_t is defined in stdlib.h as, more or less, an unsigned integer.)

The malloc( ) function returns a pointer of type void \*, which means that you can assign it to any type of pointer.

malloc כמספר שלם ללא סימן הפונקציה  
void מחזירה מצביע מסוג  
כלומר ניתן להקצות אותו לכל סוג של מצביע  
char int double and etc

After a successful call, malloc( ) returns a pointer to the first byte of the region of memory allocated from the heap. If there is not enough available memory to satisfy the malloc( ) request, an allocation failure occurs and malloc( ) returns a null.

אם אין מספיק זיכרון זמין

malloc( ) - תרחש כשל בהקצאה ו זיכרון ומחזיר ערך ריק null

p points to the start of 1,000 bytes of free memory.

מצביע על ההתחלה של 1,000 בתים של זיכרון פנוי p

```
char *p;
```

```
p = malloc(1000); /* get 1000 bytes */
```

important to understand that this automatic conversion does not occur in C++. In C++, an explicit type cast is needed when a void \* pointer is assigned to another type of pointer. Thus, in C++, the preceding assignment must be written like this:

יש צורך, ב-C++-ב. C++-חשוב להבין שהמרה אוטומטית זו אינה מתרחשת ב מוקצה לסוג אחר של מצביע. void \* בהטלת סוג מפורש כאשר מצביע: המטלה הקודמת חייבת להיכתב כך, ב-C++-לפיכך, ב

```
p = (char *) malloc(1000);
```

As a general rule, in C++ you must use a type cast when assigning (or otherwise converting) one type of pointer to another. This is one of the few fundamental differences between C and C++.

The next example allocates space for 50 integers. Notice the use of sizeof to ensure portability.

בעת הקצאה (או המרה אחרת) Type Cast-עליך להשתמש ב C++-ככלל, ב של סוג אחד של מצביע לאחר. זהו אחד מההבדלים הבסיסיים הבודדים בין C ו-C++.

-הדוגמה הבאה מקצה מקום ל-50 מספרים שלמים. שימו לב לשימוש ב- sizeof כדי להבטיח נייחות.

```
int *p;
```

```
p = (int *) malloc(50*sizeof(int));
```

Since the heap is not infinite, whenever you allocate memory, you must check the value returned by `malloc( )` to make sure that it is not null before using the pointer. Using a null pointer will almost certainly crash your program. The proper way to allocate memory and test for a valid pointer is illustrated in this code fragment:

מכיוון שהערימה אינה אינסופית, בכל פעם שאתה מקצה זיכרון, עליך לבדוק הערך המוחזר על ידי `malloc( )` כדי לוודא שהוא אינו null לפני השימוש במצביע. שימוש במצביע null תרסק כמעט בוודאות את התוכנית שלך. הדרך הנכונה להקצות זיכרון ולבדוק מצביע חוקי מומחשת בקטע קוד זה:

```
p = (int *) malloc(100);
if(!p) {
    printf("Out of memory.\n");
    exit(1); }
```

Of course, you can substitute some other sort of error handler in place of the call to `exit( )`. Just make sure that you do not use the pointer `p` if it is null.

The `free( )` function is the opposite of `malloc( )` in that it returns previously allocated memory to the system. Once the memory has been freed, it may be reused by a subsequent call to `malloc( )`.

**The function `free( )` has this prototype:**

```
void free(void *p);
```

Here, `p` is a pointer to memory that was previously allocated using `malloc( )`. It is critical that you never call `free( )` with an invalid argument; otherwise, you will destroy the free list.

כמובן, אתה יכול להחליף סוג אחר של מטפל שגיאות במקום הקריאה ליציאה ( ). רק ודא שאינך משתמש במצביע `p` אם הוא null. הפונקציה `free( )` היא ההפך מ-`malloc( )` בכך שהיא מחזירה זיכרון שהוקצה קודם לכן למערכת. לאחר שהזיכרון שוחרר, ניתן לעשות בו שימוש חוזר על ידי קריאה נוספת ל-`malloc( )`. לפונקציה `free( )` יש אב טיפוס זה:

```
;void free(void *p)
```

כאן,  $p$  הוא מצביע לזיכרון שהוקצה בעבר באמצעות `malloc` ( ). זה קריטי שלעולם לא תתקשר ל-`free` ( ) עם ארגומנט לא חוקי; אחרת, אתה תהרוס את הרשימה החינמית.

## Initializing Pointers

After a nonstatic local pointer is declared but before it has been assigned a value, it contains an unknown value. (Global and static local pointers are automatically initialized to null.) Should you try to use the pointer before giving it a valid value, you will probably crash your program—and possibly your computer's operating system as well—a very nasty type of error!

לאחר הכרזת מצביע מקומי לא סטטי אך לפני שהוקצה לו ערך, הוא מכיל ערך לא ידוע.

(.null-מצביעים מקומיים גלובליים וסטטיים מאותחלים אוטומטית ל) אם תנסה להשתמש במצביע לפני שתיתן לו ערך חוקי, סביר להניח שתתרוסק את התוכנית שלך - ואולי גם את מערכת ההפעלה של המחשב שלך - סוג מאוד מגעיל של שגיאה

```
char *p = "hello world";
```

As you can see, the pointer `p` is not an array. The reason this sort of initialization works is because of the way the compiler operates. All C/C++ compilers create

what is called a *string table*, which is used to store the string constants used by the program. Therefore, the preceding declaration statement places the address of **hello world**, as stored in the string table, into the pointer `p`. Throughout a program, `p` can be used like any other string (except that it should not be altered). For example, the following program is perfectly valid:

הסיבה שזה עובד כי המהדר מאחסן את המחרוזת בטבלה לתוך המצביע  $p$  יכול לשמש לכל מחרוזת פרט שאין לשנות אותו

```
#include <stdio.h>
#include <string.h>
char *p = "hello world";
```

```

int main(void)
{
    register int t;
    /* print the string forward and backwards */
    printf(p);
    for(t=strlen(p)-1; t>=0; t--) printf("%c", p[t]);
    return 0; }

```

In Standard C++, the type of a string literal is technically `const char *`. But C++ provides an automatic conversion to `char *`. Thus, the preceding program is still valid. However, this automatic conversion is a deprecated feature, which means that you should not rely upon it for new code. For new programs, you should assume that string literals are indeed constants and the declaration of `p` in the preceding program should be written like this.

`const char *` הסוג של מחרוזת מילולית הוא מבחינה טכנית, בתקן C++, לפיכך, התוכנית הקודמת `char *`-מספק המרה אוטומטית ל C++ אבל עדיין תקפה. עם זאת, המרה אוטומטית זו היא תכונה שהוצאה משימוש, מה שאומר שאין להסתמך עליה עבור קוד חדש. עבור תוכניות חדשות, אתה צריך להניח שמיתרמית של מחרוזת הם אכן קבועים וההצהרה של בתוכנית הקודמת צריכה להיכתב כך ק

`const char *p = "hello world";` //it can initialize מאותחל char יכול להיות  
but not type of int  
`p[0] = 'a' or "a"` //but if we try change we get error

#### error if we try initialize

`const int *p=123;` //you can't initialize a variable of type 'const int \*'  
with a value of type 'int'

program should be written like this:

```

int var=123;
const int *p= &var;

```

```

/**/

```

## examples const vs without keyword const:

1.

```
const int var=7;
var=1; // error, cannot assign to variable 'var' with const-qualified
type //'const int'
```

שגיאה, לא יכול להקצות למשתנה const int var עם סוג מוסמך ב const-qualified type=סוג מוסמך

2.

error/invalid, default initialization must be assigned

```
const char var1;
const int var;
```

3.

```
int x{ 10 };
char y{ 'M' };
```

initialize

```
const int* i = &x;
const char* j = &y;
```

```
x = 9;
y = 'A';
cout << *i << " " << *j << "- "<< i;
Value of x and y can be altered,
they are not constant variables
x ו-y ניתן לשנות את הערך של
הם אינם משתנים קבועים
```

invalid:

```
*i = 6;
*j = 'L' or 1;
```

Change of constant values because,  
i and j are pointing to const-int  
& const-char type value

שינוי של ערכים קבועים בגלל  
const-int מצביעים על j ו i  
const-char ערך סוג &

```
int h=7;
```

```
i= &h;
```

//valid, this way i can change because i const to value of h and not to address;

4.

// x and z non-const var

```
int x = 5;
```

```
int z = 6;
```

// y and p non-const var

```
char y = 'A';
```

```
char p = 'C';
```

const pointer(i) pointing  
to the var x's location

```
int* const i = &x;
```

const pointer(j) pointing  
to the var y's location

```
char* const j = &y;
```

// The values that is stored at the memory location can modified

// even if we modify it through the pointer itself

// No CTE error

```
*i = 10;
```

```
*j = 'D';
```

error CTE because pointer variable

is const type so the address

pointed by the pointer variables

can't be changed

```
*i = &z;
```

```
*j = &p;
```

```
cout << *i << " and " << *j
    << endl;
cout << i << " and " << j;
```

5.

if i try change value of address var by pointer it could be just by the rule:

(at the string char is allow const chare var[]="hello word")

1)

```
int var=123;
const int *p = &var;
*p=7; // invalid because const mean read-only variable is not
assignable
cout<< var << *p << "\n";
```

2)valid

```
int var=123;
int *p = &var; //without const mean read and write variable is
assignable
int k=1;
*p= k; // it is not pointer to address of k just get the value
p= &k; // now pointer to address of k
*p=7 //valid, because we statement without const (read and write
to variable), variable is assignable
cout<< var << *p << "\n";
```

3)

```
int var=123;
int *p = &var; // read and write variable is assignable
int k=1;
p=k;
incompatible integer to pointer conversion assigning to 'int *' from
'int';
p=k;
& קח את הכתובת עם 'int'-מ 'int *'-המרת מספר שלם למצביע לא תואם ל
```

```
/**/
```



6.

```
char str='H';  
char arr[]="Hello";
```

```
const char *p = "hello";// valid
```

הסיבה שזה עובד כי המהדר מאחסן את המחרוזת בטבלה לתוך המצביע p יכול לשמש לכל מחרוזת פרט שאין לשנות אותו

```
p="word";
```

//OVERRIDE, it is valid, because const of char string don't pointer to other address, mean you can read and write to p pointer but not allow to change the value like p[0]='H';

```
p[0]='H';// invalid, indirection requires pointer operand
```

```
p=&str; valid
```

```
p=str;invalid
```

```
p=&arr; initialize invalid, it because operator [] of arr
```

incompatible pointer types assigning to 'const char \*' from 'char (\*)[6]'

סוגי מצביע לא תואמים שמקצים ל

p=arr; valid, it's get first address of arr, could be pointer to next by p+1/p++,

7.

```
char str='a';
```

```
char array[7]='b'/"b";
```

```
or char *array[]=&str;
```

error: array initializer must be an initializer as list

אתחול מערך חייב להיות רשימת אתחול

```
current write
char array[7];
then you can Assignment to array
array[index]='a';
```

8.

```
const char *p1[5];
char st='a';
p1[1]=&st;//valid
*p1[2]='b';//invalid
```

```
char *p1[5];
char st='a';
p1[1]=&st;//valid
*p1[2]='b';//valid
```

## Problems with Pointers

Nothing will get you into more trouble than a wild pointer! Pointers are a mixed blessing. They give you tremendous power and are necessary for many programs. At the same time, when a pointer accidentally contains a wrong value, it can be the most difficult bug to find.

An erroneous pointer is difficult to find because the pointer itself is not the problem. The problem is that each time you perform an

operation using the bad pointer, you are reading or writing to some unknown piece of memory. If you read from it, the worst that can happen is that you get garbage. However, if you write to it, you might be writing over other pieces of your code or data. This may not show up until later in the execution of your program, and may lead you to look for the bug in the wrong place. There may be little or no evidence to suggest that the pointer is the original cause of the problem. This type of bug causes programmers to lose sleep time and time again.

Because pointer errors are such nightmares, you should do your best never to generate one. To help you avoid them, a few of the more common errors are discussed here. The classic

**example of a pointer error is the uninitialized pointer. Consider this program.**

**הדוגמה הקלאסית של שגיאת מצביע היא המצביע הלא מאותחל.**

This program assigns the value 10 to some unknown memory location. Here is why: Since the pointer *p* has never been given a value, it contains an unknown value when the assignment  $*p = x$  takes place. This causes the value of *x* to be written to some unknown memory location. This type of problem often goes unnoticed when your program is small because the odds are in favor of *p* containing a "safe" address—one that is not in your code, data area, or operating system. However, as your program grows, the probability increases of *p* pointing to something vital. Eventually, your program stops working. The solution is to always make sure that a pointer is pointing at something valid before it is used.

תוכנית זו מקצה את הערך 10 למיקום זיכרון לא ידוע כלשהו. הנה הסיבה: מעולם לא ניתן ערך, הוא מכיל ערך לא ידוע כאשר *p* מכיוון שלמצביע להיכתב למיקום זיכרון לא *x* זה גורם לערך של  $*p = x$  מתרחשת ההקצאה ידוע כלשהו. סוג זה של בעיה לרוב לא מורגש כאשר התוכנית שלך קטנה תכיל כתובת "בטוחה" - כזו שאינה בקוד, *p*-מכיוון שהסיכויים הם בעד ש באזור הנתונים או במערכת ההפעלה שלך. עם זאת, ככל שהתוכנית שלך יצביע על משהו חיוני. בסופו של דבר, התוכנית *p*-גדלה, ההסתברות עולה ש שלך מפסיקה לעבוד. הפתרון הוא תמיד לוודא שמצביע מצביע על משהו חוקי לפני השימוש בו.

```
int main(void)
```

```

{
int x, *p;
x = 10;
*p = x;
    printf("%d", *p);
return 0; }
/**/

```

**A second common error is caused by a simple misunderstanding of how to use a pointer.**

**שגיאה נפוצה שנייה נגרמת מאי הבנה פשוטה של אופן השימוש במצביע.**

The call to **printf( )** does not print the value of **x**, which is 10, on the screen. It prints some unknown value because the assignment is wrong. That statement assigns the value 10 to the pointer **p**. However, **p** is supposed to contain an address, not a value.

*/\* This program is wrong. \*/*

```

int main(void)
{
int x, *p;
x = 10;
p = x; <-
    printf("%d", *p);<-
return 0; }

```

**To correct the program, write:**

```

p = &x; <-
/**/

```

Another error that sometimes occurs is caused by incorrect assumptions about the placement of variables in memory. You can never know where your data will be placed in memory, or if it will be placed there the same way again, or whether each compiler will treat it in the same way. For these reasons, making any comparisons between pointers that do not point to a common object may yield unexpected results. For example,

**שגיאה נוספת שמתרחשת לפעמים נגרמת מהנחות שגויות לגבי מיקום**

משתנים בזיכרון. לעולם אינך יכול לדעת היכן הנתונים שלך ימוקמו בזיכרון, או אם הם יוצבו שם שוב באותו אופן, או אם כל מהדר יתייחס אליהם באותה צורה. מסיבות אלו, כל השוואה בין מצביעים שאינם מצביעים על אובייקט משותף עשויה להניב תוצאות בלתי צפויות. לדוגמה,

```
/* This program is wrong. */
char s[80], y[80];
char *p1, *p2;
p1 = s;
p2 = y;
if(p1 < p2) . . .<- error
/**/
```

**error: incompatible pointer types assigning to 'char \*' from 'char (\*)[80]'**

**/\*Another error\*/**

```
char s[80];
char *p1;
p1 = &s;
```

**/\*\*/**

is generally an invalid concept. (In very unusual situations, you might use something like this to determine the relative position of the variables. But this would be rare.)

A related error results when you assume that two adjacent arrays may be indexed as one by simply incrementing a pointer across the array boundaries.

This is not a good way to initialize the arrays first and second with the numbers 0 through 19. Even though it may work on some compilers under certain circumstances, it assumes that both arrays will be placed back to back in memory with first first. This may not always be the case.

הוא בדרך כלל מושג לא חוקי. (במצבים חריגים מאוד, ייתכן שתשתמש במשהו כזה כדי לקבוע את המיקום היחסי של המשתנים. אבל זה יהיה נדיר.)

שגיאה קשורה נוצרת כאשר אתה מניח ששני מערכים סמוכים עשויים להיות

מצורפים לאינדקס כאחד פשוט על ידי הגדלה של מצביע על פני גבולות המערך.

זו לא דרך טובה לאתחל את המערכים הראשונים והשניים עם המספרים 0 עד 19. למרות שזה עשוי לעבוד על כמה מהדריים בנסיבות מסוימות, היא מניחה ששני המערכים יוצבו גב אל גב בזיכרון עם הראשון הראשון. זה אולי לא תמיד המקרה.

For example,

```
/* This program is wrong. */
```

```
int first[10], second[10];
```

```
int *p, t;
```

```
p = first;
```

```
for(t=0; t<20; ++t){
```

```
    *p++ = t;}
```

```
/**/
```

This program uses **p1** to print the ASCII values associated with the characters contained in **s**. The problem is that **p1** is assigned the address of **s** only once. The first time through the loop, **p1** points to the first character in **s**. However, the second time through, it continues where it left off because it is not reset to the start of **s**. This next character may be part of the second string, another variable, or a piece of the program! The proper way to write this program is המשויכים לתווים ASCII-כדי להדפיס את ערכי ה **p1**-תוכנית זו משתמשת ב פעם אחת בלבד. **s** מוקצית הכתובת של **p1**קהביה היא של **s**-הכלולים ב עם זאת, **s**-מצביע על התו הראשון ב **p1**, בפעם הראשונה דרך הלולאה **s**. בפעם השנייה, הוא ממשיך מאיפה שהפסיק כי הוא לא מאופס לתחילת התו הבא הזה עשוי להיות חלק מהמחרוזת השנייה, משתנה אחר או חלק מהתוכנית! הדרך הנכונה לכתוב תוכנית זו היא

```
/* This program has a bug. */
```

```
#include <string.h>
```

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    char *p1;
```

```
    char s[80];
```

```
p1 = s; // <-bug
```

```

do {
    gets(s); /* read a string */
    /* print the decimal equivalent of each
       character */
    while(*p1) printf(" %d", *p1++);
} while(strcmp(s, "done"));
return 0; }

```

**// correct.**

Here, each time the loop iterates, **p1** is set to the start of the string. In general, you should remember to reinitialize a pointer if it is to be reused.

The fact that handling pointers incorrectly can cause tricky bugs is no reason to avoid using them. Just be careful, and make sure that you know where each pointer is pointing before you use it.

**/\* This program is now correct. \*/**

```

int main(void)
{
    char *p1;
    char s[80];
do {
    p1 = s; //<- correct.
    gets(s); /* read a string */
    /* print the decimal equivalent of each
       character */
    while(*p1) printf(" %d", *p1++);
} while(strcmp(s, "done"));
return 0; }
/**/

```

## C's Dynamic Allocation Functions

C/C++ מצביעים מספקים תמיכה הכרחית למערכת ההקצאה דינמית של הקצאה דינמית היא האמצעי שבאמצעותו תוכנית יכולה להשיג זיכרון בזמן שהיא פועלת.

כפי שאתה יודע, למשתנים גלובליים מוקצים אחסון בזמן הקומפילציה.

משתנים מקומיים משתמשים במחסנית. עם זאת, לא ניתן להוסיף משתנים גלובליים או מקומיים במהלך הפעלת התוכנית. עם זאת, יהיו זמנים שבהם לא ניתן לדעת מראש את צורכי האחסון של תוכנית. לדוגמה, תוכנית עשויה להשתמש במבנה נתונים דינמי, כגון רשימה מקושרת או עץ בינארי. מבנים כאלה הם מטבעם דינמיים באופיים, גדלים או מתכווצים לפי הצורך. כדי ליישם מבנה נתונים כזה, נדרשת שתוכנית תהיה מסוגלת להקצות ולפנות זיכרון.

תומך למעשה בשתי מערכות הקצאה דינמיות מלאות: זו המוגדרת על C++ מכילה מספר C++-המערכת הספציפית ל C++-וזו הספציפית ל C ידי גישה זו נידונה בחלק השני. כאן, C שיפורים ביחס לזו שבה משתמש C. מתוארות פונקציות ההקצאה הדינמית של מתקבל מהערימה C זיכרון שהוקצה על ידי פונקציות ההקצאה הדינמית של - אזור הזיכרון הפנוי שנמצא בין התוכנית שלך לבין אזור האחסון הקבוע שלה והמחסנית. למרות שגודל הערימה אינו ידוע, הוא בדרך כלל מכיל כמות די גדולה של זיכרון פנוי. ( free() ו malloc() מורכבת מהפונקציות C הליבה של מערכת ההקצאה של רוב המהדרים מספקים עוד כמה פונקציות הקצאה דינמית, אבל שתי אלה.) (הם החשובים ביותר)

(A C++ program may also use the C++-style header <cstdlib>.)

יש אב טיפוס זה: The malloc( ) function has this prototype:  
void \*malloc(size\_t number\_of\_bytes);

(The type size\_t is defined in stdlib.h as, more or less, an unsigned integer.)

The malloc( ) function returns a pointer of type void \*, which means that you can assign it to any type of pointer.

malloc כמספר שלם ללא סימן הפונקציה

מחזירה מצביע מסוג void

כלומר ניתן להקצות אותו לכל סוג של מצביע.

char int double and etc

After a successful call, malloc( ) returns a pointer to the first byte of the region of memory allocated from the heap. If there is not enough available memory to satisfy the malloc( ) request, an allocation failure occurs and malloc( ) returns a null.

אם אין מספיק זיכרון זמין

malloc( ) - תרחש כשל בהקצאה ו זיכרון ומחזיר ערך ריק null



מצביע על ההתחלה של 1,000 בתים של זיכרון פנוי p

```
char *p;  
p = malloc(1000); /* get 1000 bytes */
```

important to understand that this automatic conversion does not occur in C++. In C++, an explicit type cast is needed when a void \* pointer is assigned to another type of pointer. Thus, in C++, the preceding assignment must be written like this:

יש צורך, ב-C++, חשוב להבין שהמרה אוטומטית זו אינה מתרחשת ב מוקצה לסוג אחר של מצביע. void \* בהטלת סוג מפורש כאשר מצביע: המטלה הקודמת חייבת להיכתב כך, ב-C++, לפיכך, ב

```
p = (char *) malloc(1000);
```

As a general rule, in C++ you must use a type cast when assigning (or otherwise converting) one type of pointer to another. This is one of the few fundamental differences between C and C++.

The next example allocates space for 50 integers. Notice the use of sizeof to ensure portability.

בעת הקצאה (או המרה אחרת) Type Cast-עליך להשתמש ב C++-ככלל, ב של סוג אחד של מצביע לאחר. זהו אחד מההבדלים הבסיסיים הבודדים בין C ו-C++.

הדוגמה הבאה מקצה מקום ל-50 מספרים שלמים. שימו לב לשימוש ב- sizeof להבטיח נייודות.

```
int *p;  
p = (int *) malloc(50*sizeof(int));
```

Since the heap is not infinite, whenever you allocate memory, you must check

the value returned by malloc( ) to make sure that it is not null before using the pointer. Using a null pointer will almost certainly crash your program. The proper way to allocate memory and test for a valid pointer is illustrated in this code fragment:

מכיוון שהערימה אינה אינסופית, בכל פעם שאתה מקצה זיכרון, עליך לבדוק

הערך המוחזר על ידי malloc ( ) כדי לוודא שהוא אינו null לפני השימוש במצביע. שימוש במצביע null תרסק כמעט בוודאות את התוכנית שלך. הדרך הנכונה להקצות זיכרון ולבדוק מצביע חוקי מומחשת בקטע קוד זה:

```
p = (int *) malloc(100);
if(!p) {
    printf("Out of memory.\n");
    exit(1); }
```

Of course, you can substitute some other sort of error handler in place of the call to exit( ). Just make sure that you do not use the pointer p if it is null.

The free( ) function is the opposite of malloc( ) in that it returns previously allocated memory to the system. Once the memory has been freed, it may be reused by a subsequent call to malloc( ).

**The function free( ) has this prototype:**

**void free(void \*p);**

Here, p is a pointer to memory that was previously allocated using malloc( ). It is critical that you never call free( ) with an invalid argument; otherwise, you will destroy the free list.

כמובן, אתה יכול להחליף סוג אחר של מטפל שגיאות במקום הקריאה ליציאה ( ). רק ודא שאינך משתמש במצביע p אם הוא null. הפונקציה free ( ) היא ההפך מ- malloc ( ) בכך שהיא מחזירה זיכרון שהוקצה קודם לכן למערכת. לאחר שהזיכרון שוחרר, ניתן לעשות בו שימוש חוזר על ידי קריאה נוספת ל- malloc ( ). לפונקציה free ( ) יש אב טיפוס זה:

```
void free(void *p);
```

כאן, p הוא מצביע לזיכרון שהוקצה בעבר באמצעות malloc ( ). זה קריטי שלעולם לא תתקשר ל-free ( ) עם ארגומנט לא חוקי; אחרת, אתה תהרוס את הרשימה החינמית.

