ΗΥ-150
Προγραμματισμός

Δυναμική Διαχείριση Μνήμης

Δυναμική Διαχείριση Μνήμης

- Μέχρι τώρα στατική ανάθεση και δέσμευση μνήμης
  - Ζητούσαμε στο πρόγραμμά μας τη μέγιστη μνήμη που μπορεί να χρειαζόμασταν
  - #define MAX_SIZE 10000
  - int array[MAX_SIZE];
  - Μειονέκτημα : Υπάρχουν εφαρμογές που το MAX_SIZE είναι άγνωστο οπότε αναγκαζόμαστε να δηλώνουμε μεγάλες τιμές και να χάνουμε και σε μνήμη και σε ταχύτητα

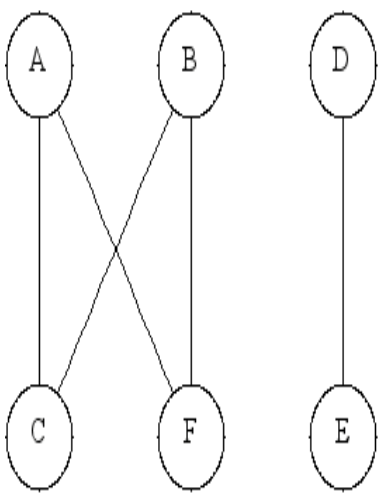
- Δυναμική κατανομή μνήμης (dynamic memory allocation)
  - Αίτηση από το λειτουργικό να μας παραχωρήσει μνήμη
  - Ειδοποίηση στο λειτουργικό ότι δεν χρειαζόμαστε πια ένα μέρος μνήμης
  - Ό,τι δεσμεύετε πρέπει να αποδεσμεύετε

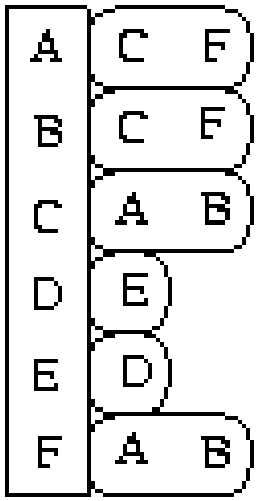
Γιατί χρειαζόμαστε δυναμική μνήμη;

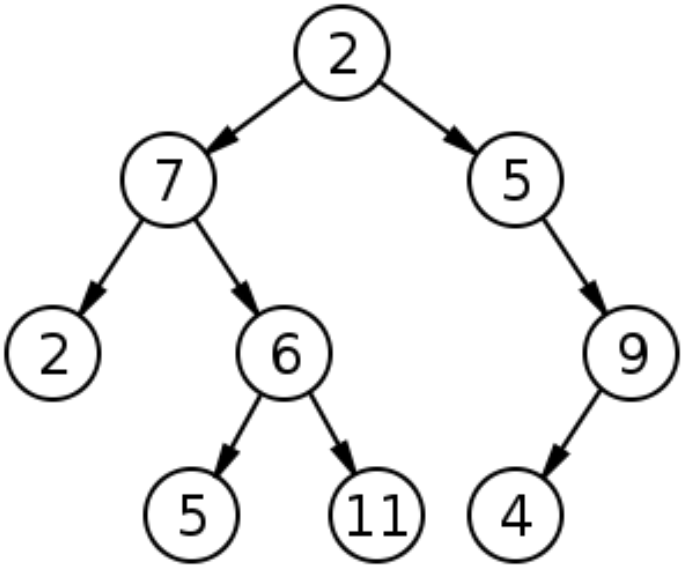
- Διότι ο αριθμός των δεδομένων μπορεί να μην είναι γνωστός τη στιγμή της μετάφρασης. Π.χ.
  - Να εξαρτάται από την είσοδο (π.χ. ο χρήστης επιλέγει να δώσει Ν αριθμούς)
  - Μπορεί να εξαρτάται από κάποιο ενδιάμεσο υπολογιστικό αποτέλεσμα
  - Τα δεδομένα μπορεί να αυξομειώνονται κατά την εκτέλεση του προγράμματος

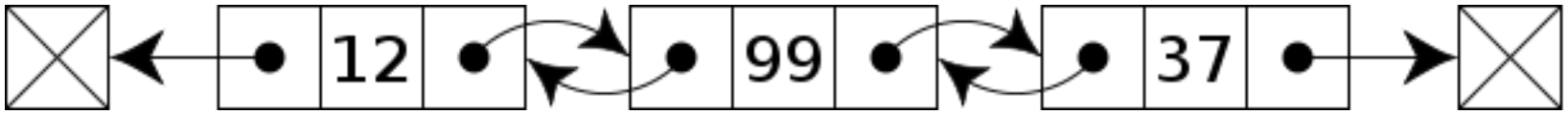
Γιατί χρειαζόμαστε δυναμική μνήμη;

- Βάση για πιο πολύπλοκες δομές δεδομένων





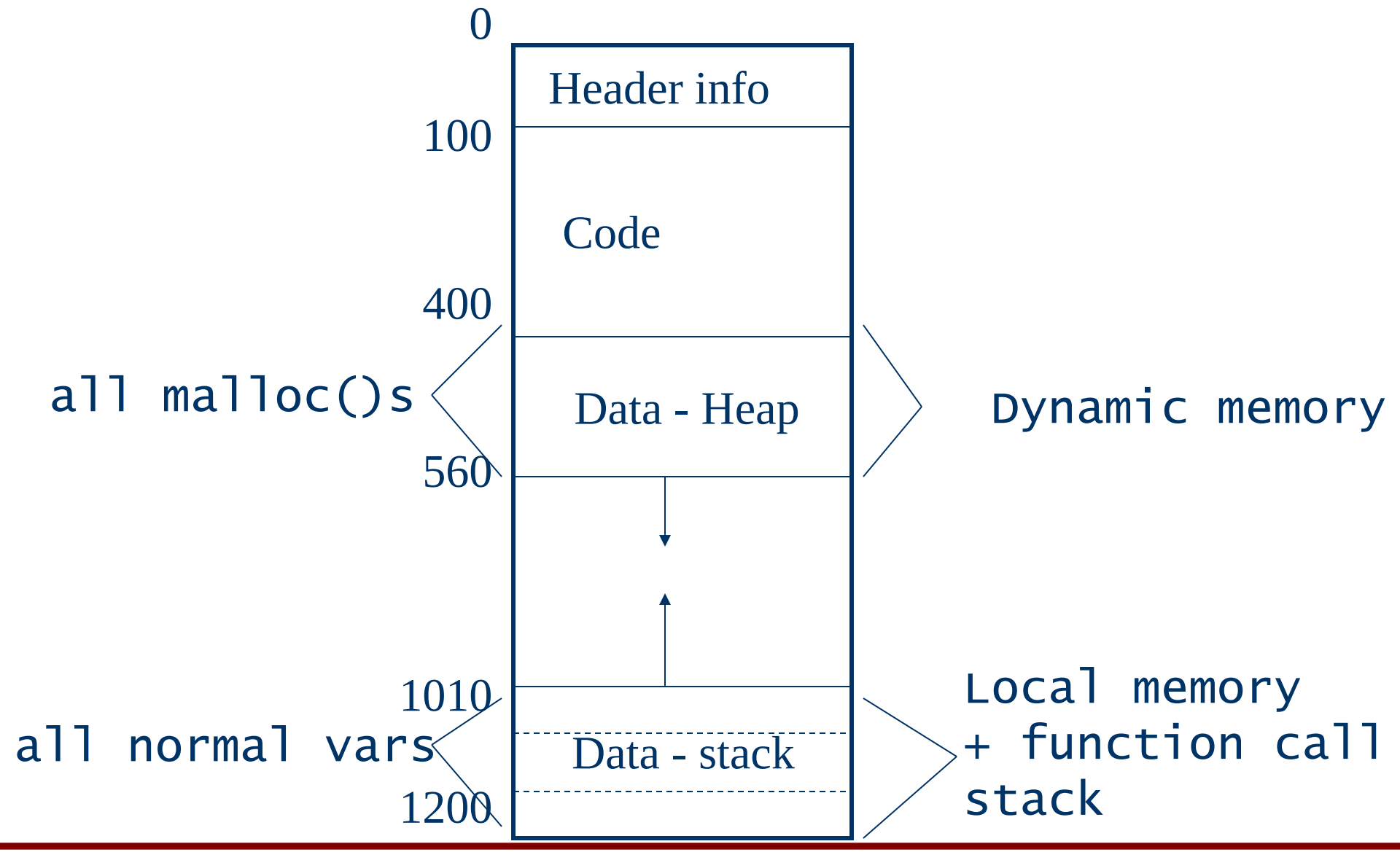




Κατηγορίες Μνήμης του Προγράμματος

- Μνήμη αποθήκευσης του κώδικα του προγράμματος
- Μνήμη αποθήκευσης των μεταβλητών ενός προγράμματος που δεσμεύονται με στατική ή αυτόματη διαχείριση μνήμης
  - Αυτή η μνήμη ονομάζεται στοίβα (stack)
- Μνήμη αποθήκευσης των μεταβλητών ενός προγράμματος που δεσμεύονται με δυναμική διαχείριση μνήμης
  - Αυτή η μνήμη ονομάζεται σωρός (heap)

Memory layout of programs



Συναρτήσεις Δυναμικής Διαχείρισης Μνήμης

- **void *malloc(size_t numBytes);**
  - Δεσμεύει numBytes διαδοχικά bytes και επιστρέφει δείκτη στην αρχή της δεσμευμένης μνήμης
  - Επιστρέφει NULL αν δε μπορεί να δεσμεύσει τη μνήμη

```
int *ptr;
ptr = (int *) malloc (100*sizeof(int));
```

Συναρτήσεις Δυναμικής Διαχείρισης Μνήμης

- **void free(void *pointer);**
  - Αποδεσμεύει τη μνήμη που έχει δεσμευτεί με μία από τις προηγούμενες κλήσεις στην malloc, calloc ή realloc
  - Δεν πρέπει να αποδεσμεύουμε πάνω από μια φορά την ίδια μνήμη

```
ptr = (int *)malloc (100*sizeof(int));
free(ptr); // Αυτό είναι οκ μέχρι εδώ
free(ptr); // Αποδεσμεύουμε την ίδια μνήμη
                    // δυο φορές! ΛΑΘΟΣ
```

Δυναμική Διαχείριση Μνήμης

Παράδειγμα

```
#include <stdlib.h>

int main(void)
{
        int *ptr;
                /* allocate space to hold an int */
        ptr = (int  *) malloc(sizeof(int));

                /* do stuff with the space */
        *ptr=4;

        free(ptr);
                /* free up the allocated space */
}
```

Δυναμικός πίνακας

```
int *ptr, i,size;
cout <<"Enter the size of the array\n";
cin >> size;

ptr = (int *) malloc( size * sizeof(int) );
for(i=0; i<size; i++){
   ptr[i] = i;
}
```

Συναρτήσεις Δυναμικής Διαχείρισης Μνήμης

- **void *calloc(size_t nelements, size_t bytes);**
  - Δεσμεύει nelements διαδοχικά αντικείμενα το καθένα μεγέθους bytes και επιστρέφει δείκτη στην αρχή της δεσμευμένης μνήμης
  - Επιστρέφει NULL αν δε μπορεί να δεσμεύσει τη μνήμη
  - Αρχικοποιεί τα περιεχόμενα της δεσμευμένης μνήμης στο 0
  - Παρόμοια με την malloc
    - Διαφορά 1: λίγο διαφορετική κλήση
    - Διαφορά 2: αρχικοποίηση των περιεχομένων με 0

```
int *ptr = (int *) calloc (100, sizeof(int));
```

Συναρτήσεις Δυναμικής Διαχείρισης Μνήμης

- **void *realloc(void *ptr, size_t size);**
  - Επέκταση μνήμης
  - Ο δείκτης *ptr πρέπει να δείχνει σε μνήμη ήδη δεσμευμένη με την malloc ή calloc
    - ptr = (int *)malloc (100*sizeof(int));
  - Επαναδέσμευση μνήμης με άλλο μέγεθος
    - ptr = (int *)realloc(ptr, 200*sizeof(int));
  - Τα προηγούμενα περιεχόμενα της δεσμευμένης μνήμης διατηρούνται (τουλάχιστον όσα χωρούν στην καινούργια)
  - Η καινούργια μνήμη μπορεί να είναι σε άλλη διεύθυνση αν δεν υπήρχε αρκετά μεγάλο συνεχόμενο block από ελεύθερη μνήμη για δέσμευση. Σε αυτήν την περίπτωση γίνεται μεταφορά των δεδομένων και **αποδεσμεύεται η μνήμη ptr,** σαν να γίνεται δηλαδή μια free(ptr)

Ο τύπος void*

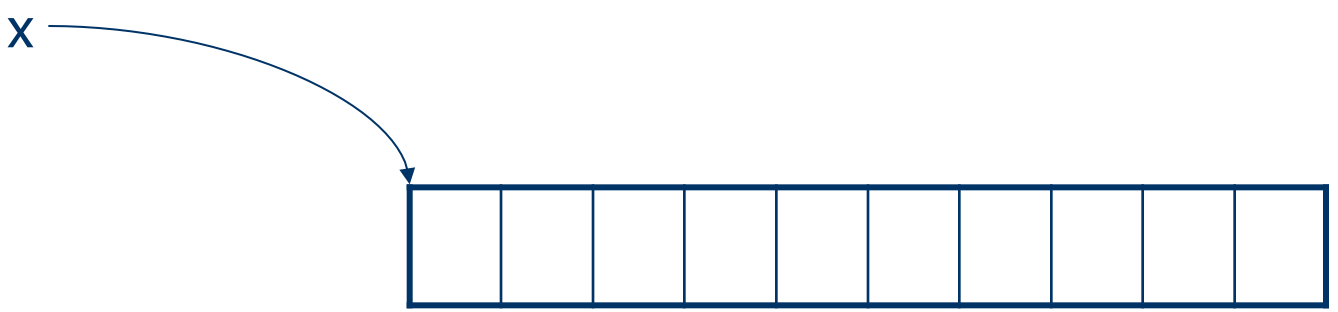
- Γενικός δείκτης
- Κατά την εκτέλεση του προγράμματος διευκρινίζεται ο τύπος του, π.χ. εάν θα είναι char *, int *, float *, double *, struct my_struct *, κ.ο.κ.

- Ο γενικός δείκτης μας δίνει τη δυνατότητα να ορίζουμε συναρτήσεις που μας επιστρέφουμε χώρο στη μνήμη χωρίς να μας ενδιαφέρει ο τύπος! Π.χ. malloc()

- ΠΡΟΣΟΧΗ: σε συναρτήσεις είναι τελείως διαφορετικό
  1. void f() // η συνάρτηση ΔΕΝ επιστρέφει τίποτα
  2. void *f() //επιστρέφει ένα δείκτη που ο προγραμματιστής πρέπει να //αποσαφηνίσει τον τύπο του μέσα στην συνάρτηση

Δυναμικοί Πίνακες

- Πίνακες που το μέγεθος τους καθορίζεται κατά την εκτέλεση του προγράμματος (σε run-time).
- Δυνατότητα προσαρμογής στις εκάστοτε απαιτήσεις χωρίς ανάγκη περιορισμού.
- int *x = (int *)malloc(10 * sizeof(int));



Πρόβλημα 1: Αποτυγχάνει η Δέσμευση

- Πρέπει να γίνεται **πάντα** έλεγχος επιστροφής της απαραίτητης μνήμης !!
- Είναι ευθύνη του προγραμματιστή και στοιχείο σωστού προγραμματισμού!
- Οι συναρτήσεις δυναμικής δέσμευσης επιστρέφουν NULL όταν αποτυγχάνουν, το οποίο μπορούμε να εξετάσουμε και να διακόψουμε την εκτέλεση του προγράμματος αν χρειάζεται

```
int *p = (int *)malloc(10000*sizeof(int));

if (p == (int *) NULL)
{
        printf( “No Memory Available\n”);
        exit(0);
}
```

Πρόβλημα 2: Διαρροή Μνήμης

- Απελευθέρωση
  - Πρέπει να ελευθερώνουμε την μνήμη όταν δεν την χρειαζόμαστε (με την free)
- **Δεν ελευθερώνουμε ποτέ μνήμη που δεν έχει αποκτηθεί δυναμικά (λ.χ. με malloc)**
- Αν δεν ελευθερώσουμε όλη τη μνήμη που δεσμεύουμε αυτό ονομάζεται διαρροή μνήμης (memory leak)

- Η διαρροή μνήμης μπορεί να οδηγήσει σε καταστάσεις που δεν υπάρχει άλλη ελεύθερη μνήμη
  - Προχωρημένο: όταν τελειώνει η ελεύθερη μνήμη του υπολογιστή, τότε το λειτουργικό στέλνει κάποια μνήμη στον δίσκο, ελευθερώνει έτσι μνήμη και την ξανα-χρησιμοποιεί. Ξαναφέρνει από τον δίσκο στη μνήμη όταν γίνεται πρόσβαση σε αυτήν. Αυτό οδηγεί στο φαινόμενο trashing όπου γράφεται και διαβάζεται από το δίσκο συνέχεια μνήμη. Με διαρροή μνήμης ένα πρόγραμμα μπορεί να αρχίσει να κάνει thrashing μετά από ώρες, μέρες, ή χρόνια συνεχής λειτουργίας του.

  - **Λύση :** Αποδεσμεύουμε αμέσως τη μνήμη που δεν χρησιμοποιούμε (χρήση της free)

Πρόβλημα 3:

- Χρήση Μετά την Αποδέσμευση
  - Κάνουμε free και μετά ξαναχρησιμοποιούμε την ίδια μνήμη
  - Παρόμοια αποτελέσματα όταν χρησιμοποιούμε μη δεσμευμένη μνήμη
    - Αλλάζουμε μεταβλητές που χρησιμοποιούμε χωρίς να το ξέρουμε – δύσκολα ανιχνεύεται
    - Πάμε να γράψουμε σε μνήμη που δεν μας ανήκει (segmentation fault) και το πρόγραμμα διακόπτεται αυτόματα

Πρόβλημα 4:

- Αποδέσμευση μνήμης που δεν έχουμε δεσμεύσει
  - Μπορεί να μπερδέψει το λειτουργικό στην διαχείριση ελεύθερης μνήμης

Πρόβλημα 5: “Dangling” pointers

```
int *i, *x;
i = (int*)malloc( 5 x sizeof(int));
x = i;          //both point to the same address.

free(x);

x = NULL;     // One way to prevent incorrect access
i = NULL;
```

Καλές πρακτικές

- Αρχικοποίηση
- Προσμέτρηση του '\0' στο μέγεθος strings.
- Έλεγχος τιμής επιστρογής της malloc()
- Για κάθε malloc() υπάρχει και η αντίστοιχη free()
- Το &variable είναι ένας pointer
- Μετά την αποδεύσμευση ενός pointer εκχώρηση της τιμής NULL σε αυτόν
- Π.χ.

```
int * pint=(int *)malloc(sizeof(int));
   /*
   Do something with pint
   */
   free(pint) ;
   pint = NULL;
```

Για «ασφάλεια» και αποφυγή κατά λάθος επαναχρησιμοποίησης της.

Pointers ως παράμετροι

```
#include <iostream>
#include <cstdlib>
using namespace std;
int sumAndInc(int *pa, int *pb,int* pc);
int main(){
   int a=4, b=5, c=6;
   int *ptr = &b;
   /* call the function */
   int total = sumAndInc(&a,ptr,&c);
   cout << "The sum of " << a << " and " << b << " is "
<< total << " and c is " << c << endl;
}
/* pointers as arguments */
int sumAndInc(int *pa, int *pb,int *pc ){
   *pc = *pc+1; // return a pointee value; NOT *(pc+1)
   return (*pa+*pb);      /* return by value */
}
```

```
int main(int argc, char *argv[]) {
   int *p;
   DoSomething( &p );
   cout <<*p <<endl;      /* will this work ? */
   return 0;
}
/* passed and returned using pointers */
void DoSomething(int **ptr){
   int temp= 8;
   *ptr = (int*)malloc(sizeof(int));
   **ptr = temp;
}
```

Σύγκριση pointers

- Είναι δυνατή ανεξαρτήτων που δείχνουν. Έχει νόημα όταν δείχνουν σε στοιχεία πίνακα.
- Γίνεται με τους τελεστές <, >, =, <= and >=
- Σύγκριση άσχετων μεταξύ τους pointers είναι απρόβλεπτη.

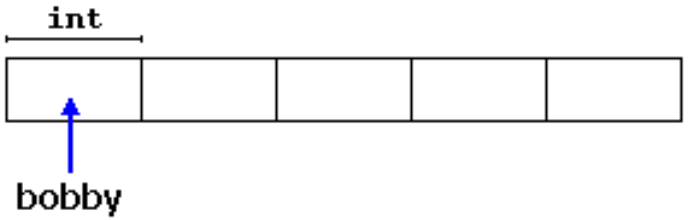
```
int data[100], *p1, *p2;
for (int i = 0; i <100;i++)
            data[i] = i;
p1 = &data [1];
p2 = &data [2];
if (p1 > p2)
            cout<<"\n\n p1 is greater than p2";
else
            cout<<"\n\n p2 is greater than p1";
```

Operators new and new[]

- pointer = new type
- pointer = new type [number_of_elements]

The first expression is used to allocate memory to contain one single element of type type. The second one is used to assign a block (an array) of elements of type type, where number_of_elements is an integer value representing the amount of these. For example:

- int * bobby;
- bobby = new int [5];



In this case, the system dynamically assigns space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to bobby. Therefore, now, bobby points to a valid block of memory with space for five elements of type int.

Ανεπάρκεια μνήμης

- int * bobby;
- bobby = new (nothrow) int [5];
- if (bobby == 0)
- {
  - // error assigning memory. Take measures.
- };

Operators delete and delete[]

- Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator delete, whose format is:

- delete pointer;
- delete [] pointer;

  The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements.

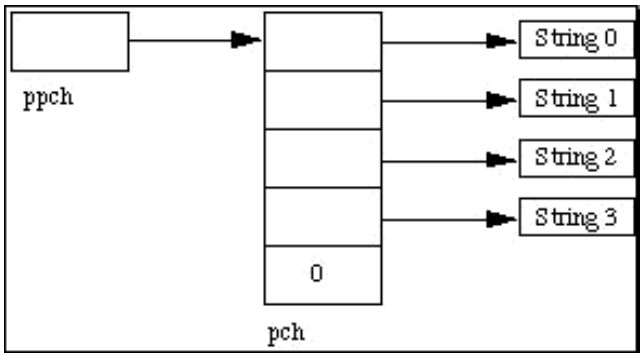
  The value passed as argument to delete must be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

```
#include <iostream>
#include <new>

using namespace std;
int main () {
      int i,n;
      int *p;
      cout << "How many numbers would you like? ";
      cin >> i;
      p= new (nothrow) int[i];
      if (p == NULL)
            cout << "Error: memory could not be allocated";
      else {
                  for (n=0; n<i; n++) {
                        cout << "Enter number: ";
                        cin >> p[n];
                  }
                  cout << "You have entered: ";
                  for (n=0; n<i; n++)
                        cout << p[n] << ", ";
                  delete[] p;
            }
      return 0;
}
```

```
How many numbers would you like?
5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32

You have entered: 75, 436, 1067, 8, 32,
```

Δείκτες σε δείκτες

- char ch; char *pch; char **ppch; char ***ppch;



- Δείκτης σε πίνακα από δείκτες (άγνωστο στην αρχή και το μέγεθος του πίνακα και το μέγεθος των περιοχών που δείχνουν οι δείκτες)

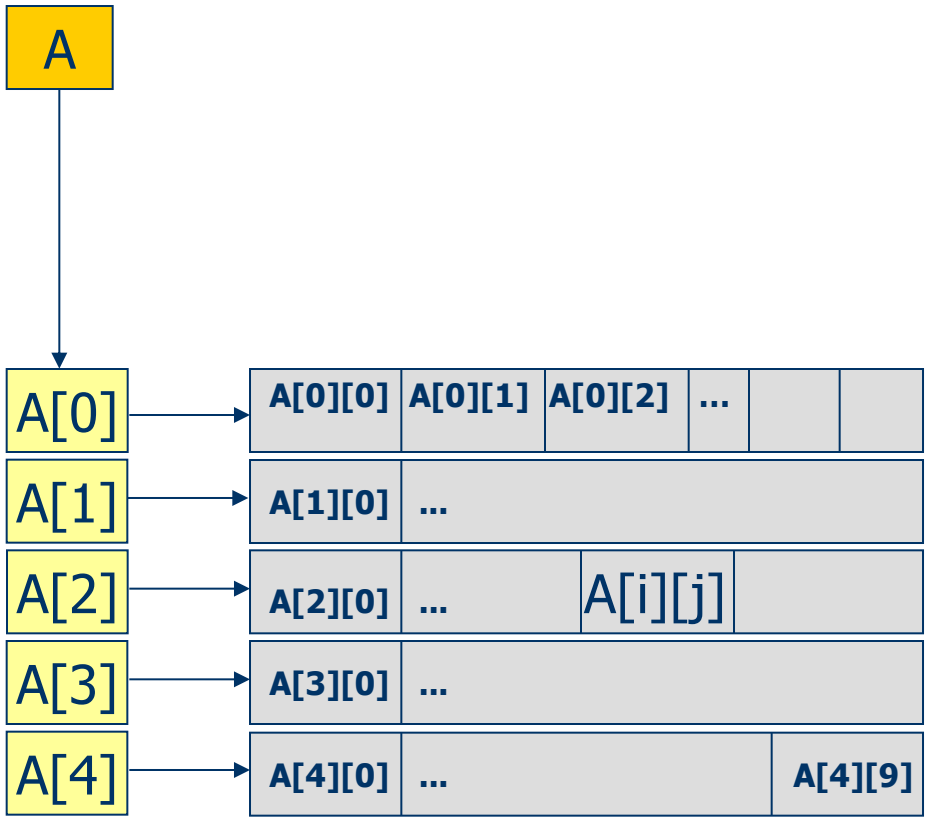
Παράδειγμα 2Δ πίνακα - C++

```
int **dynamicArray = 0;

//memory allocated for elements of rows.
dynamicArray = new int *[ROWS] ;
//memory allocated for elements of each column.

for( int i = 0 ; i < ROWS ; i++ )
        dynamicArray[i] = new int[COLUMNS];

//free the allocated memory
for( int i = 0 ; i < ROWS ; i++ )
        delete [] dynamicArray[i] ;
delete [] dynamicArray ;
```

Δείκτες σε δείκτες – C++

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(void)
{
        int lines = 5; /* Both lines and cols could be evaluated */
        int cols = 10; /* or read in at run time */
        int i, **A;
        A = new int *[lines] ;
        if (A == NULL)
        {
              cout<<"Failure to allocate room for row pointers.\n";
              exit(0);
        }
        for (i = 0; i < lines; i++)
        {
              A[i] = = new int[cols];
              if (A[i] == NULL)
              {
                  cout<<"\nFailure to allocate for row <<i<<endl;
                  exit(0);
              }
        }
        //free the memory
        for (i = 0; i < lines; i++)
              delete [] A[i];
        delete [] A;
        return 0;
}
```

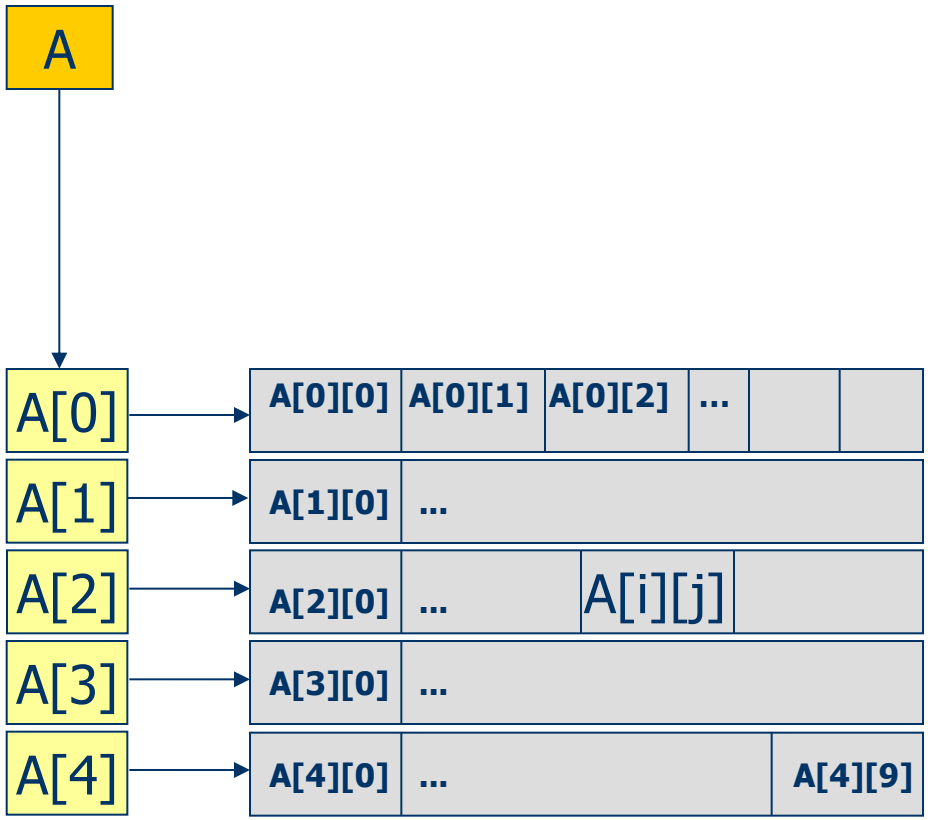
Εφαρμογή : πίνακας - εικόνα



Δείκτες σε δείκτες - C

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(void)
{
        int lines = 5; /* Both lines and cols could be evaluated */
        int cols = 10; /* or read in at run time */
        int i, **A;
        A = (int **)malloc(lines * sizeof(int *));
        if (A == NULL)
        {
              cout<<"Failure to allocate room for row pointers.\n";
              exit(0);
        }
        for (i = 0; i < lines; i++)
        {
              A[i] = (int *) malloc(cols * sizeof(int));
              if (A[i] == NULL)
              {
                  cout<<"\nFailure to allocate for row <<i<<endl;
                  exit(0);
              }
        }
        //free the memory
        for (i = 0; i < lines; i++)
              free (A[i]);
        free(A);
        return 0;
}
```

Εφαρμογή : πίνακας - εικόνα



Συνοπτικά …

- Έχουμε:

```
int x;
int *px;
px = &x
```

- Και ένα ακόμα βήμα:

```
int **ppx;
ppx = &px;
```

- Οπότε μπορούμε να κάνουμε:

```
**ppx = 42; /* Puts 42 into x */
```

- Αλλά όχι:

```
*ppx = 42; /* ERROR!! */
```

Παραδείγματα

ADDR	CONT	SYMB
1000	42	x
2000	1000	px
3000	2000	ppx

```
int x;
int *px;
int **ppx;
px = &x
ppx = &px;
**ppx = 42;
```

ΠΡΟΣΟΧΗ!

- Δείκτες σε Δείκτες έναντι Δυσδιάστατους Πίνακες
- int a[10][20];
  - a[r][c] είναι συντακτικά αποδεκτό
  - Δέσμευση μνήμης: 10×20 ακέραιοι
  - Υπολογισμός διεύθυνσης του a[r][c]: r*20+c+αρχή του πίνακα
- int **b;
  - b[r][c] είναι συντακτικά αποδεκτό
  - Δέσμευση μνήμης για b:10 δείκτες b = (int **)malloc (10*sizeof(int*))
  - Δέσμευση μνήμης για b[i]: 20 ακέραιοι σε κάθε γραμμή

```
for (i = 0; i < 10; i++)
    b[i] = (int *) malloc(20 * sizeof(int));
```

  - Υπολογισμός διεύθυνσης του b[r][c]: διεύθυνση μνήμης του δείκτη b[r] + c
  - ΠΡΙΝ ΤΟΝ ΥΠΟΛΟΓΙΣΜΟ ΤΟΥ b[r][c] πρέπει να έχει γίνει σωστή ανάθεση τιμών στα b[r] και **δέσμευση μνήμης**.

Strings

- Χρήση δεικτών για την διαχείριση strings
- Όπως έχουμε πει:

  int a[ ] = {0, 1, 3, 5}
- Και η διάσταση θα καθοριστεί σε a[4]
- Επίσης:

  char b[ ] = {‘t’, ‘e’, ‘s’, ‘t’, ‘\0’}

  ⇔

  char c[ ] = “test”;

Όμως …

char aLine[ ] = “Hi, there!”;

char *pLine = “Hi, there!”;

- Παρόμοια αλλά όχι τα ίδια!!!
- Το πρώτο καθορίζει έναν πίνακα χαρακτήρων (11) που μπορεί να αλλαχθεί:

  aLine[4] = ‘T’; /* Σωστό! */
- Το δεύτερο καθορίζει έναν δείκτη προς μία σταθερά τύπου string που δεν μπορεί να αλλαχθεί:

  pLine[4] = ‘T’; /* ΛΑΘΟΣ!! */
- Ενώ αν:

  pLine = aLine;

  pLine[4] = ‘T’; /* Σωστό! */

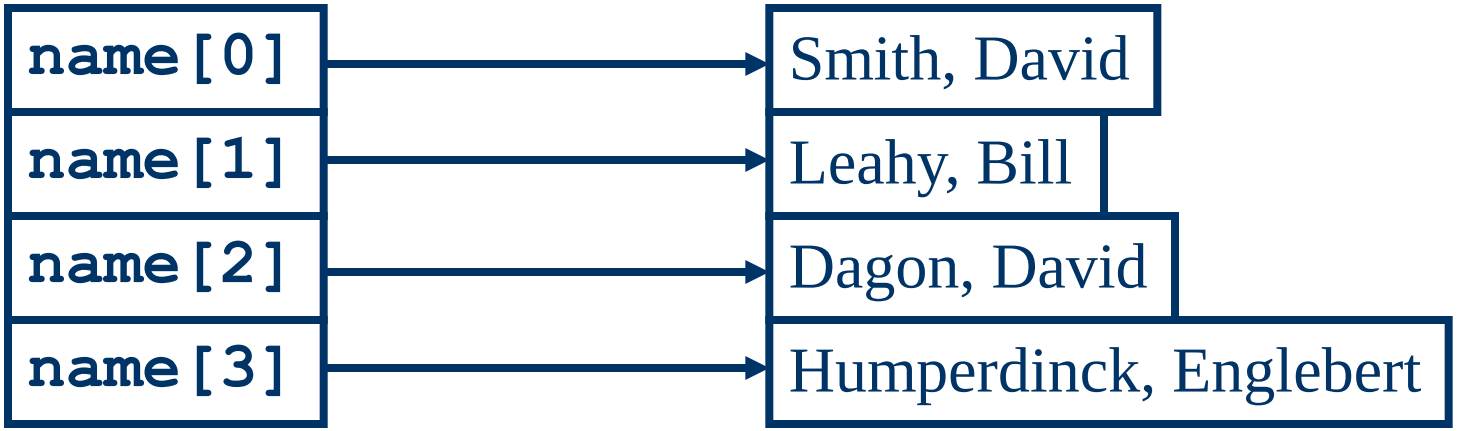
Κι άλλα …

- Όχι μόνο δείκτες σε δείκτες αλλά και πίνακες από δείκτες!
- char *name[ ] = {“Illegal”, “Jan”, ”Feb”, ”Mar”, ”Apr”, ”May”, “Jun”, “Jul”, “Aug”, “Sep”, “Oct”, “Nov”, “Dec”};
- Π.χ. name[2] είναι ένας δείκτης στο character string ‘F’, ‘e’, ‘b’, ‘\0’
- Γιατί θα θέλαμε έναν πίνακα από δείκτες;

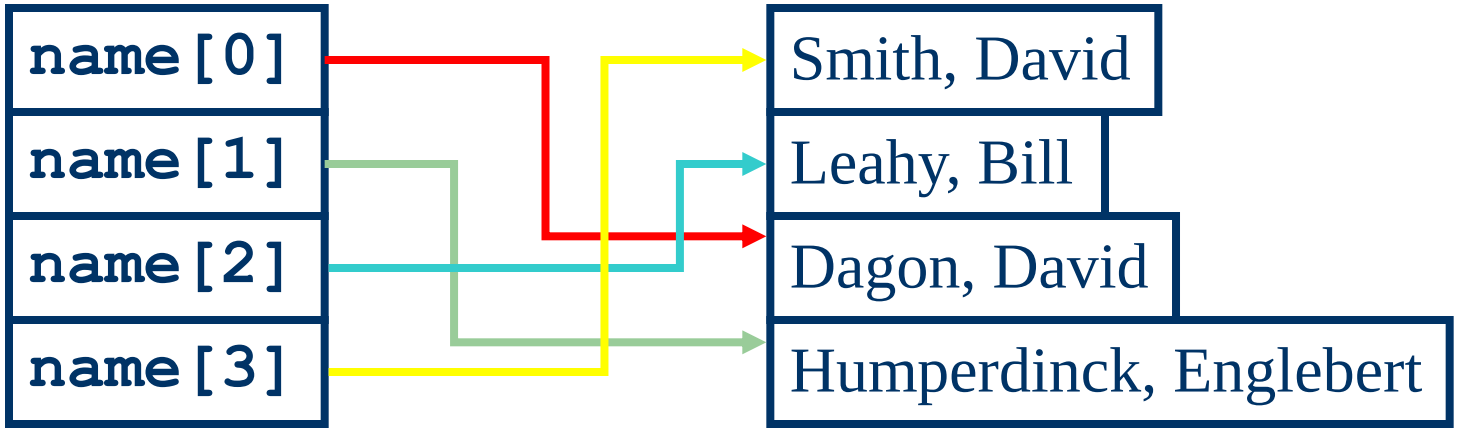
Φανταστείτε :

- Έχουμε ονόματα όπως:
  - Smith, David
  - Leahy, Bill
  - Dagon, David
  - Humperdinck, Englebert
- **Υποθέστε πως θέλουμε να τα ταξινομήσουμε αλφαβητικά αλλά παρατηρούμε πως είναι διαφορετικού μεγέθους.**
- **Αντί να χρησιμοποιήσουμε έναν αλγόριθμο ταξινόμησης που αλλάζει character strings around γιατί να μην χρησιμοποιήσουμε έναν πίνακα από δείκτες και να αλλάζουμε δείκτες;**
  - **πολύ πιο γρήγορο**

Έχουμε έναν πίνακα από pointers



Ταξινόμηση με μετακίνηση δεικτών



- struct emploee *A;
- int N;
- cin >> N;
- A = (struct emploee *) malloc(N*sizeof(struct emploee));

- A[100]…

- free(A);

Dynamic Memory Allocation

Static Allocation, in C

```
int g_x;

int main(void)
   {
   int x = 1;

   int a[20];

   …
   }
```

g_x may be put on the stack or allocated statically at compile time

x is pushed on the stack when main() is invoked

a (all 20 elements) is also pushed on the stack when main() is invoked

The size of a is also fixed at compile time

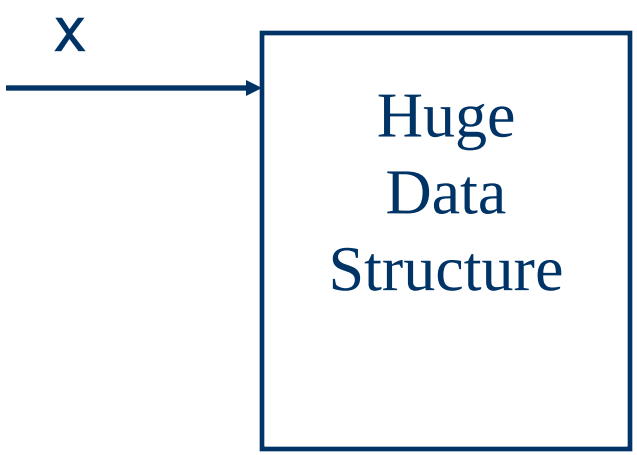
- Stack space may be limited
- May not know how much data must be stored at compile time

Dynamic Memory Allocation

Dynamic Allocation

- Reserves memory from a much larger *free store*, or *heap*
- Gives the programmer a pointer to refer to this memory

```
x = malloc ( sizeof (HDS));
```

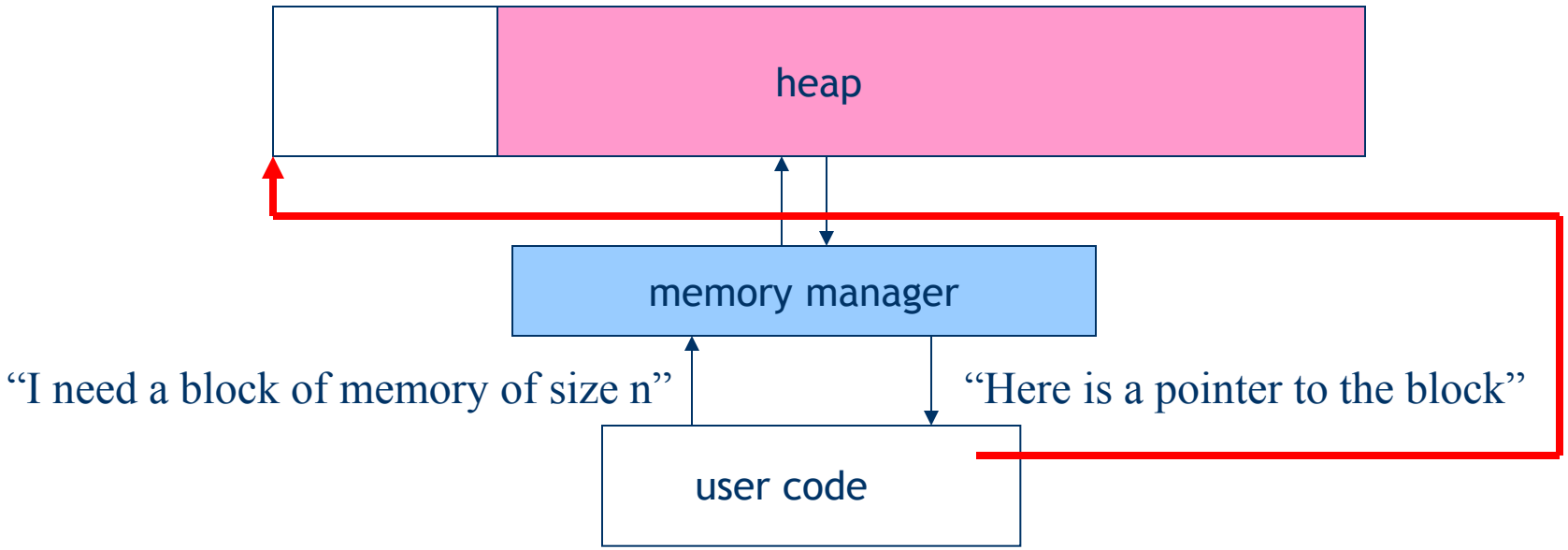



```
function(x);
```

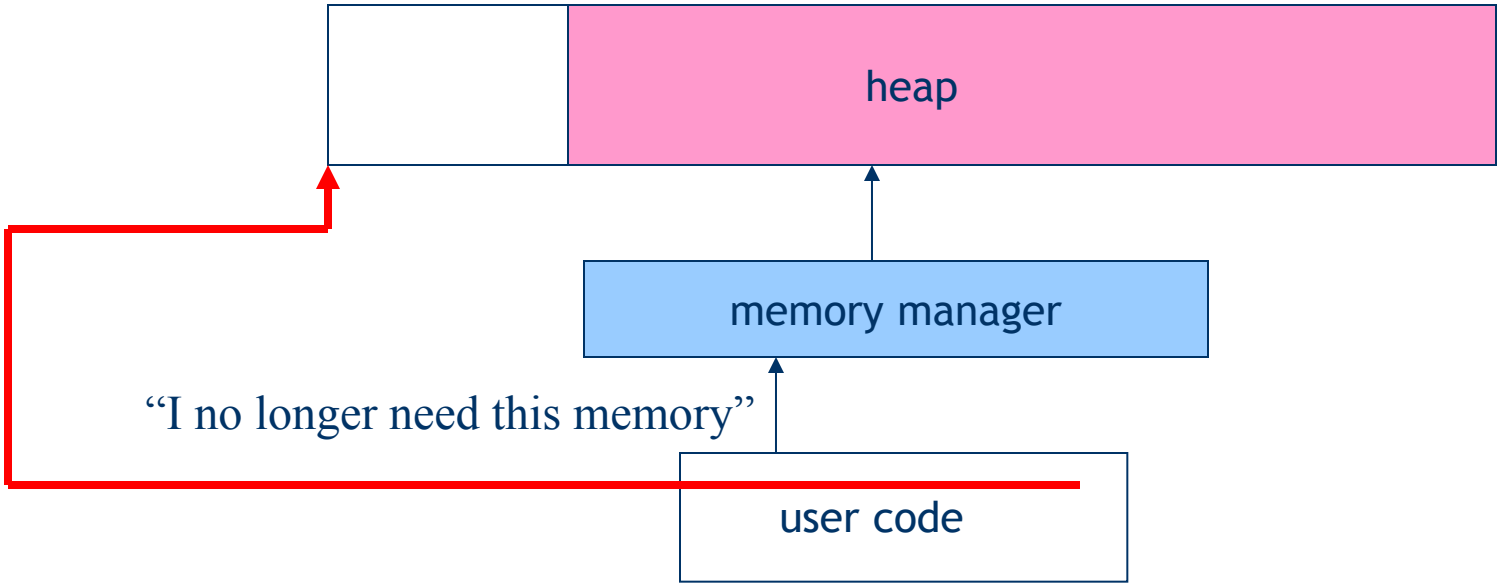
Only the pointer x
is pushed onto the stack

Call-by-reference

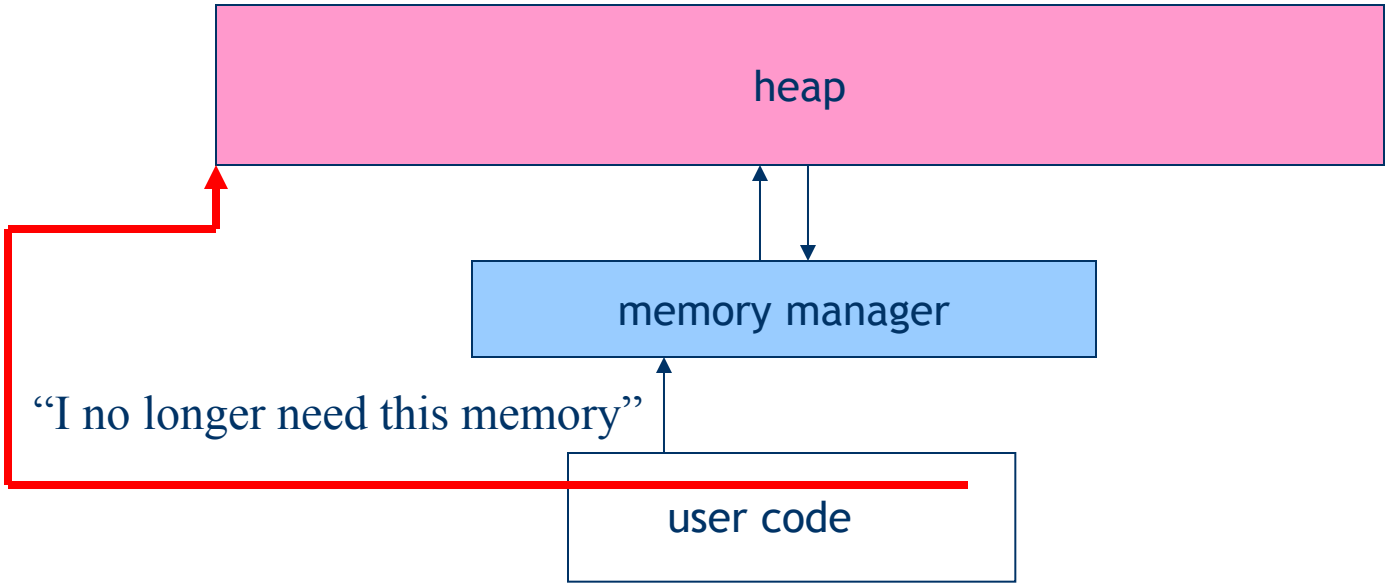
The Heap: Allocation



The Heap: Deallocation

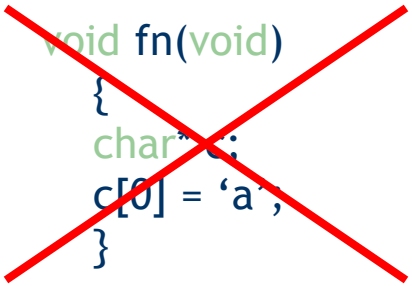


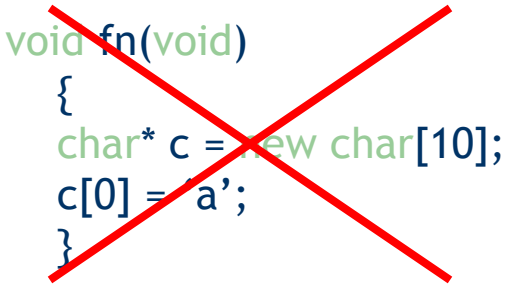
The Heap: Deallocation



Manual Allocation

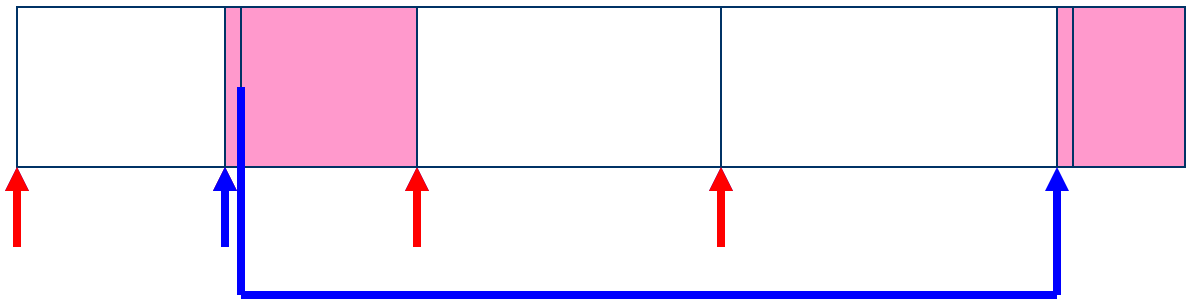
Memory deallocation driven through explicit requests to allocate and deallocate memory is called *manual allocation*





Some languages include facilities for the *automatic deallocation* of memory; these facilities are called *garbage collectors*

Allocation Method: First Fit



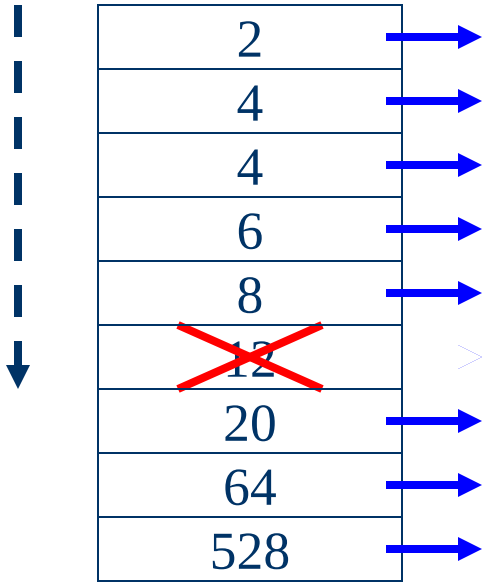
- The first block in the heap that is big enough is allocated first
- Each free block keeps a pointer to the next free block
- These pointers may be adjusted as memory is allocated

We can keep track of free space
in a separate list called the *free list*

Allocation Method: First Fit

- Sorting the free list may result in performance benefits
  - Increasing size:
    Scanning the free list to allocate new memory will always pick the smallest free block

```
x = malloc(10);
```



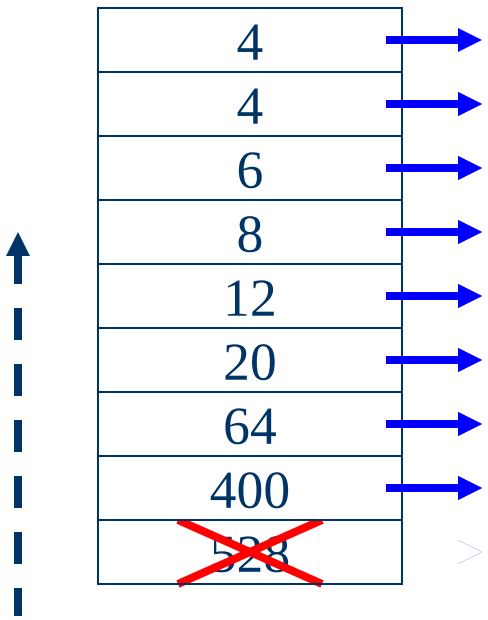
Allocation Method: First Fit

- Sorting the free list may result in performance benefits
  - Decreasing size:
    Scanning the free list to allocate new memory will always pick the first block since it is the largest

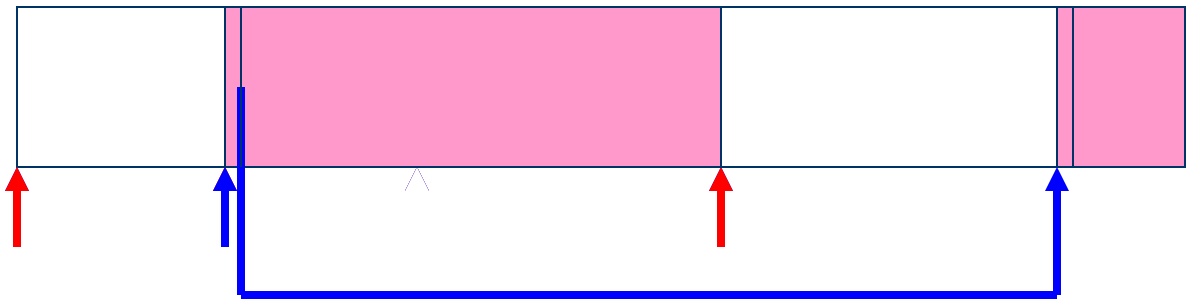
x = malloc(128);

x = malloc(512);

No contiguous block exists



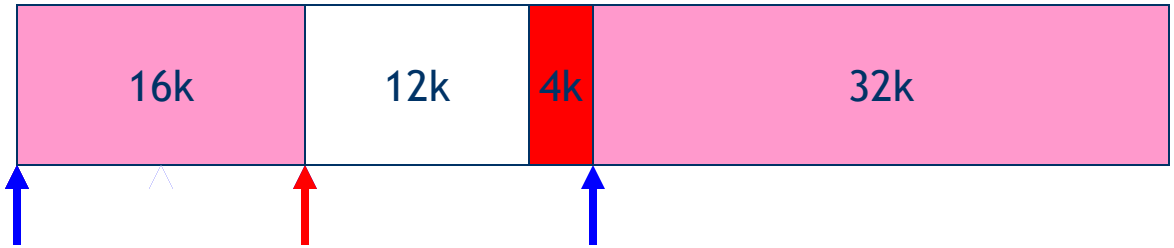
Coalescence



- When two free blocks are adjacent to each other, they can be merged into a single block

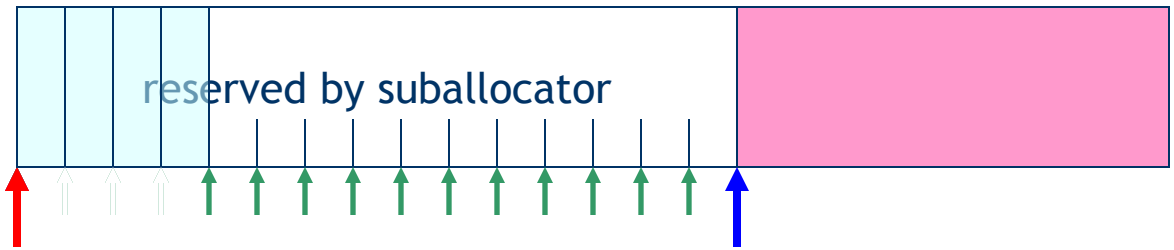
- This process is called *coalescence*

Allocation Method: Buddy System



- When a block is allocated, memory is divided into blocks, giving each block a same-size “buddy”
- If a block cannot be subdivided, memory can be wasted
- Coalescence of memory can be done quickly at the expense of this wasted memory

Allocation Method: Suballocators



- Suballocators are user code which reserve the system-provided heap and then manage that memory itself
- System memory management must operate in a general fashion
- Suballocators can take advantage of known characteristics of the application's memory needs to speed the process
  - e.g., equal-sized blocks

Memory Management Issues

- Dangling Pointers (widows)

```
char* c = new char[10];
```

Memory is allocated,
c points to that block

```
…
```

```
delete[] c;        c = 0;
```

c's memory is deallocated

```
…
```

```
c[0] = 'a';
```

c is dereferenced,
accessing memory that was
already deallocated

This problem can occur in any language which uses explicit deallocation

Memory Management Issues

Memory Leaks (orphans)

```
void fn(void)
  {
  char* c = new char[10];

  …
    delete[] c;
  return;
  }
```

Memory is allocated, and c points to that block

c falls out of scope without the memory it points to being deallocated

If no other pointer points to this memory, it can no longer be referenced, and thus cannot be deallocated.

With repeated application, eventually the entire heap will fill up with unreachable allocated memory!

Memory Management Issues

Heap Fragmentation

Internal heap fragmentation results from the allocation of wasted space by the memory manager

Some heap schemes waste space by placing allocation data in the heap (size of allocated blocks, etc.)

Where heap data is required to be aligned to particular byte boundaries, additional *padding* may be allocated to match those boundaries

Memory Management Issues

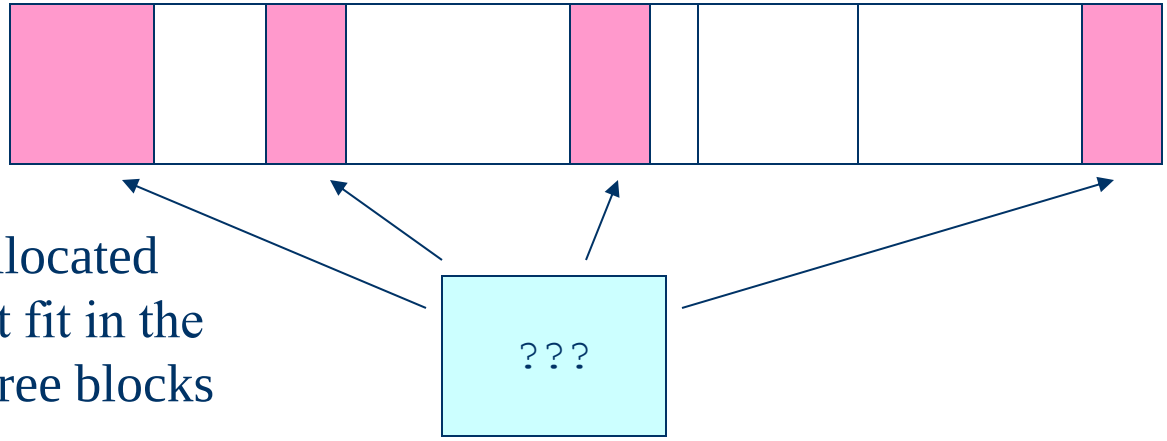
Heap Fragmentation

External heap fragmentation results from repeated allocation and deallocation from the heap

Can be corrected by periodic compaction of the heap and coalescence of the resulting adjacent free blocks

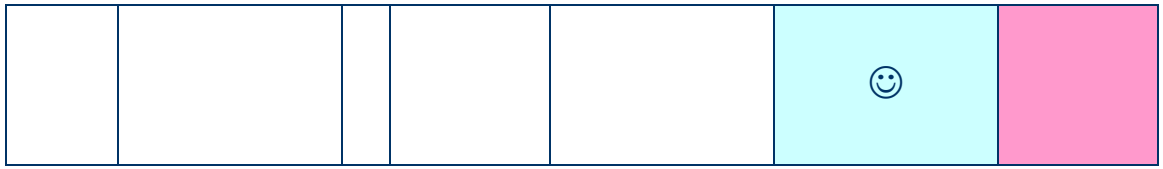
Memory Management Issues

Heap Fragmentation (external)



A newly allocated block can't fit in the available free blocks

But by *compacting* the heap...



...a larger contiguous block is created!

Memory Management Issues

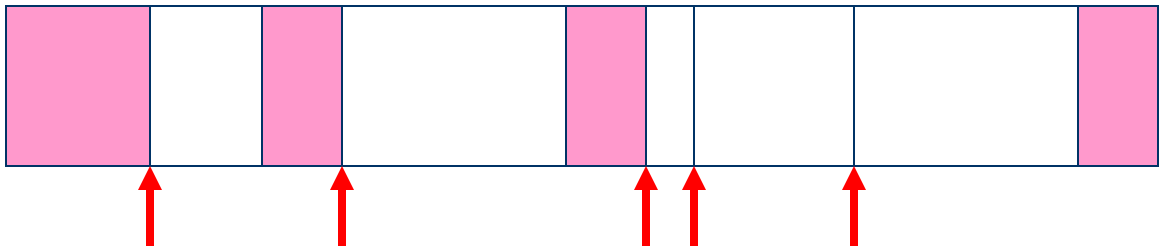
External Heap Fragmentation

But what about all the pointers referring to the moved blocks of memory?

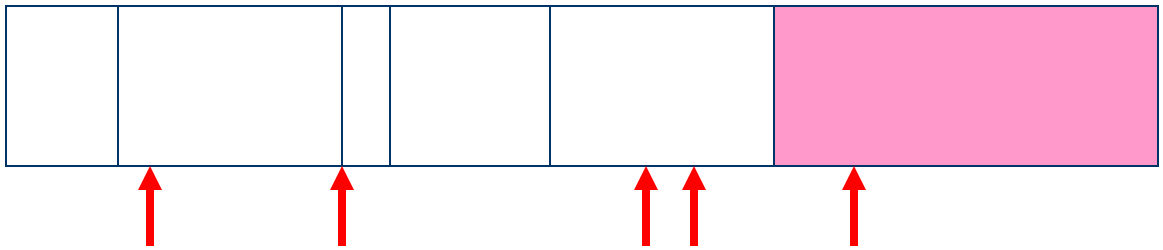
If the system knows the locations of every pointer to every allocated block, then it can adjust those pointers accordingly.

Some systems are not privy to such information; one solution is to use *handles* to allocated blocks rather than returning pointers directly to those blocks.

Memory Management issues

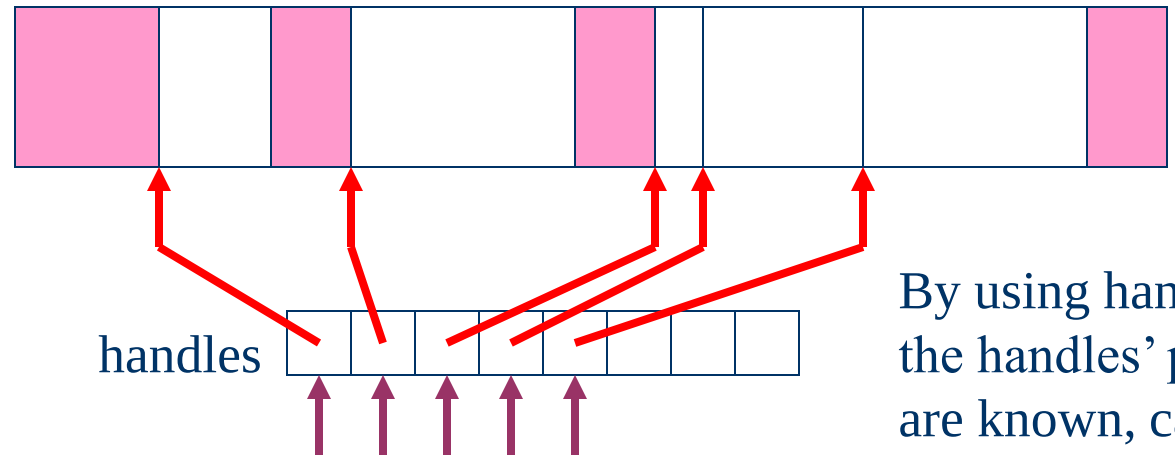


If the heap is compacted but pointers to blocks in the heap are not changed...

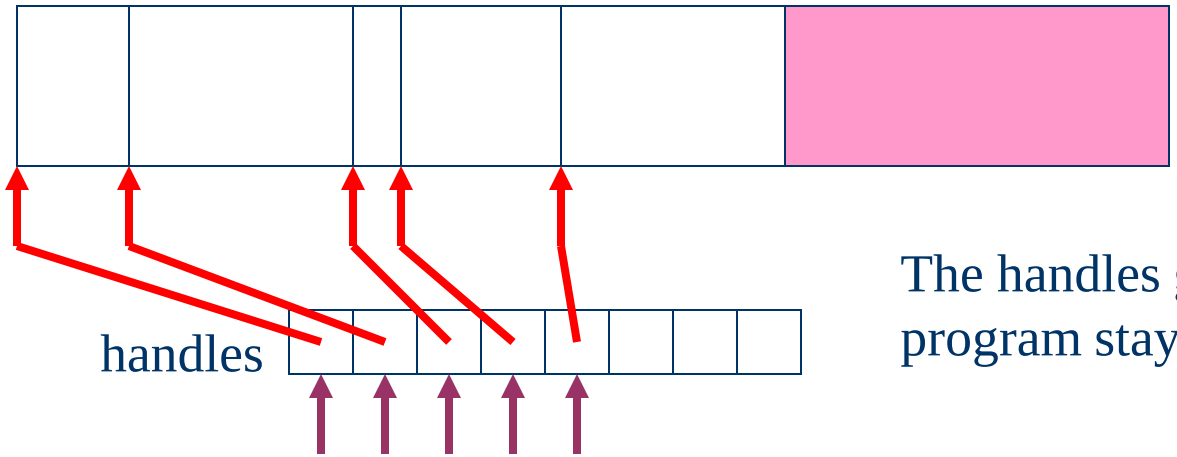


...the pointers will point to other blocks or to no blocks at all once the blocks are compacted.

Memory Management issues



By using handles to blocks, the handles' pointers, which are known, can be changed.



The handles given to the program stay the same.