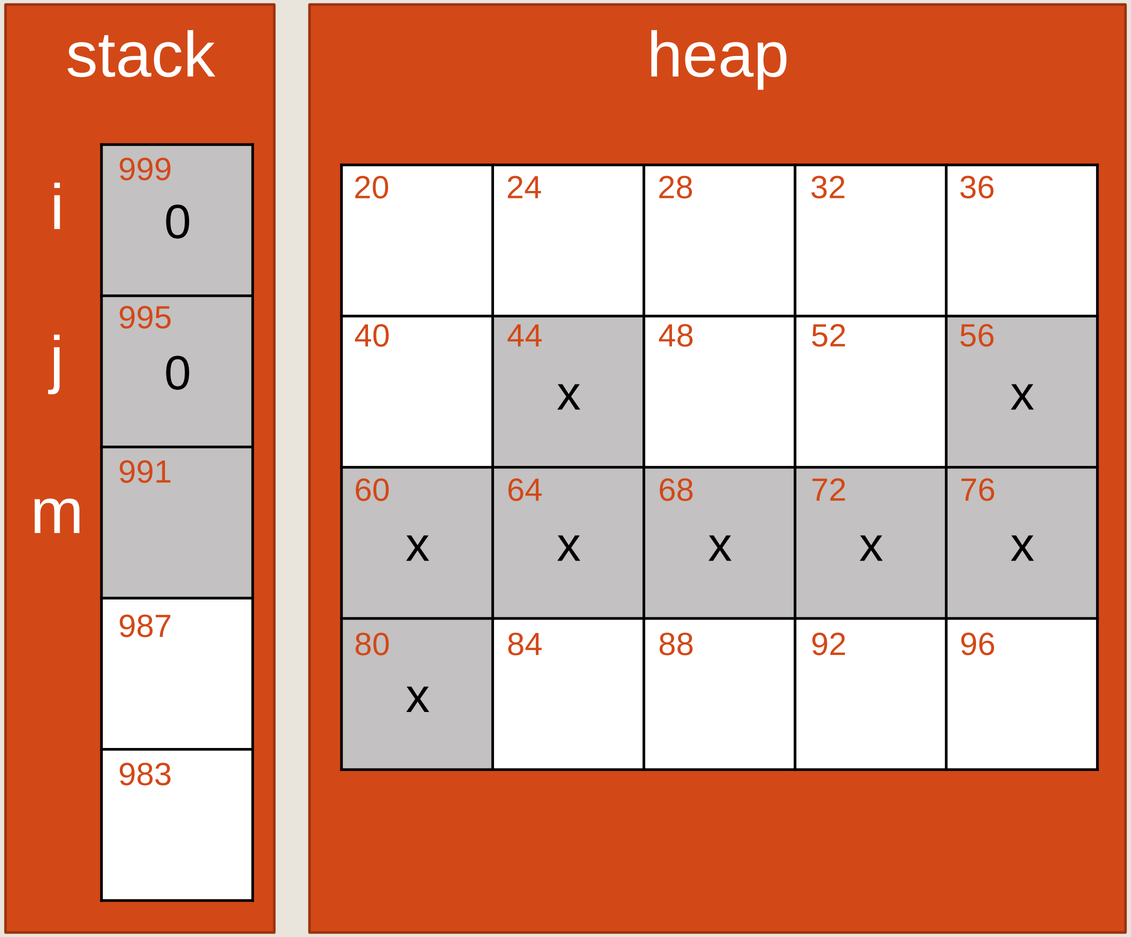
Memory Leaks – yet again…

Another memory leak example

```
void createMemLeak() {
	int i, j;
	int **m = (int**) malloc(sizeof(int)*3);
	for (i = 0; i < 3; i++) {
		m[i] = (int*) malloc(sizeof(int)*2);
		for (j = 0; j < 2; j++) {
			m[i][j] = i + j;
		}
	}
}
```

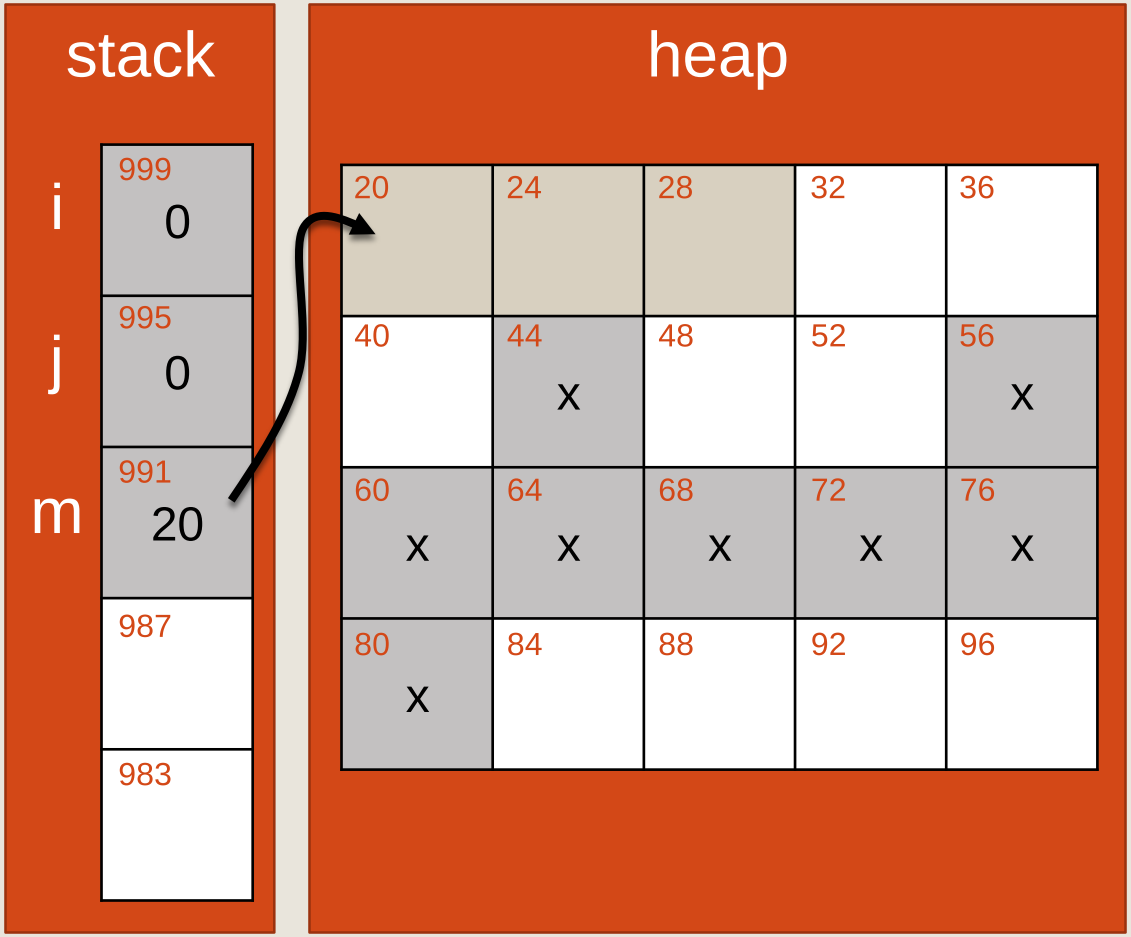
In the following example we assume:
`sizeof(int)==sizeof(void*)==4`

Another memory leak example



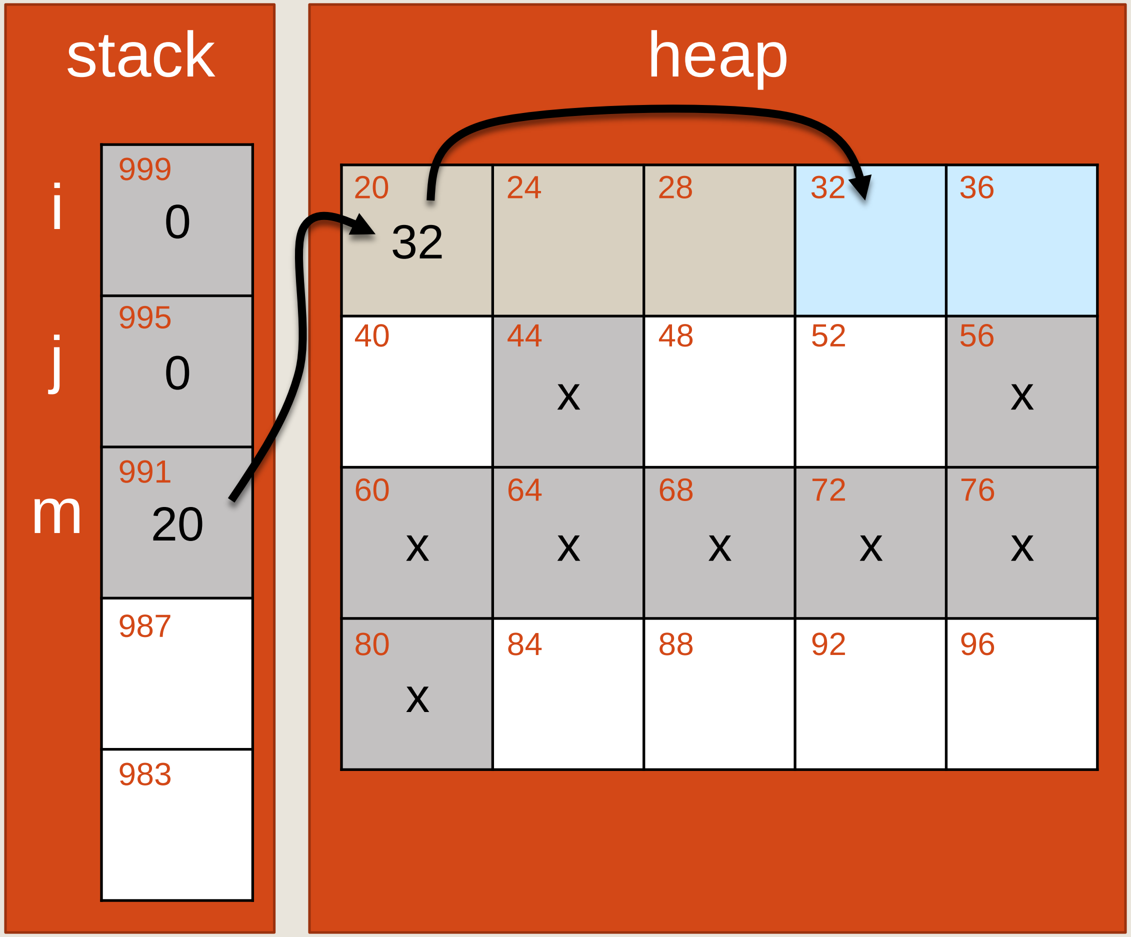

```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

Another memory leak example



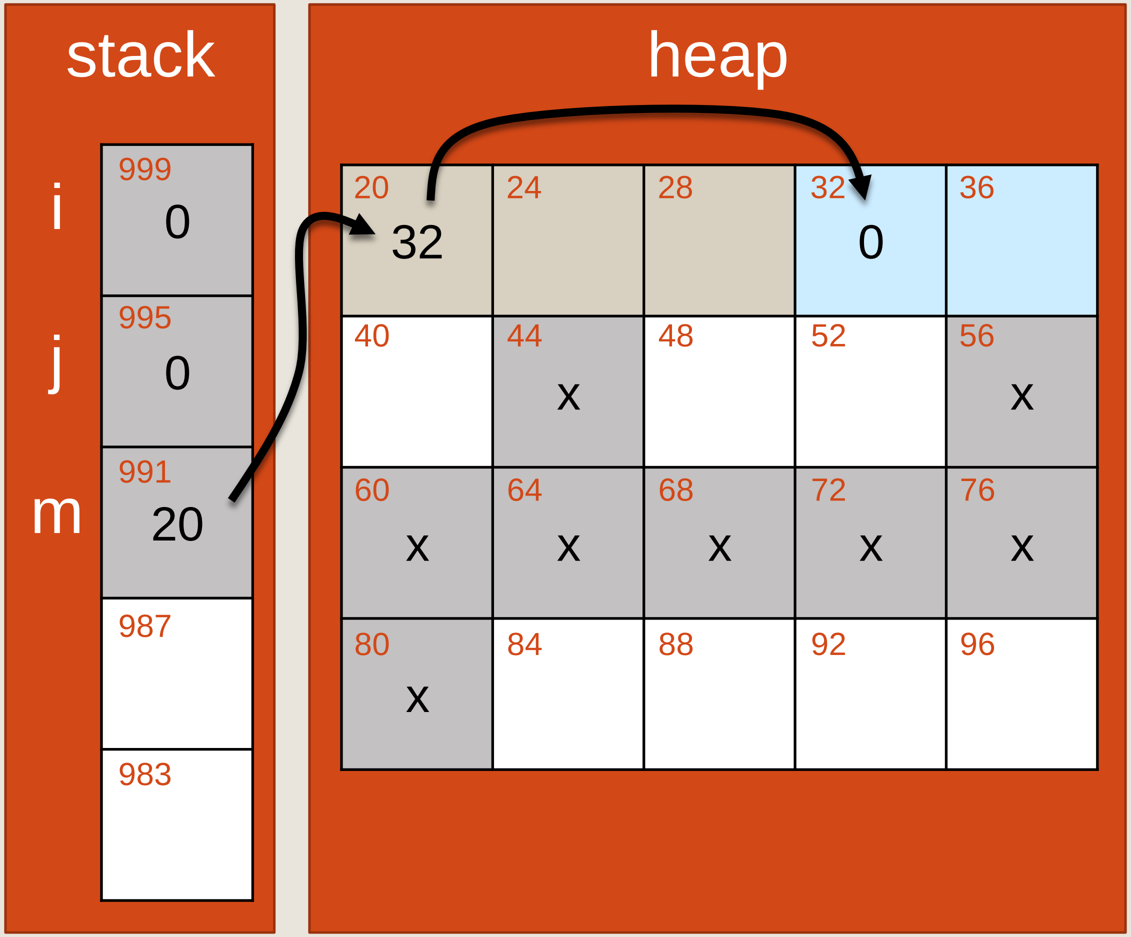

```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

Another memory leak example



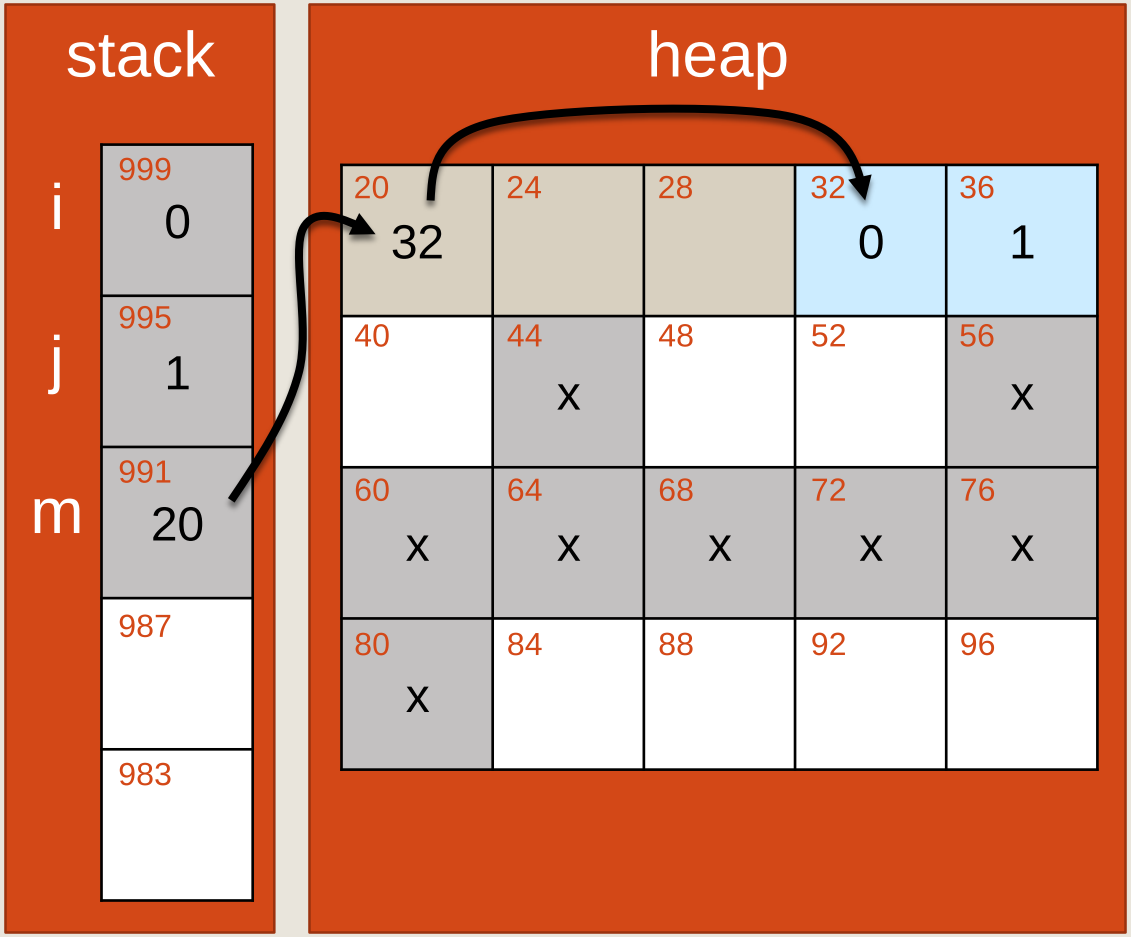

```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

Another memory leak example



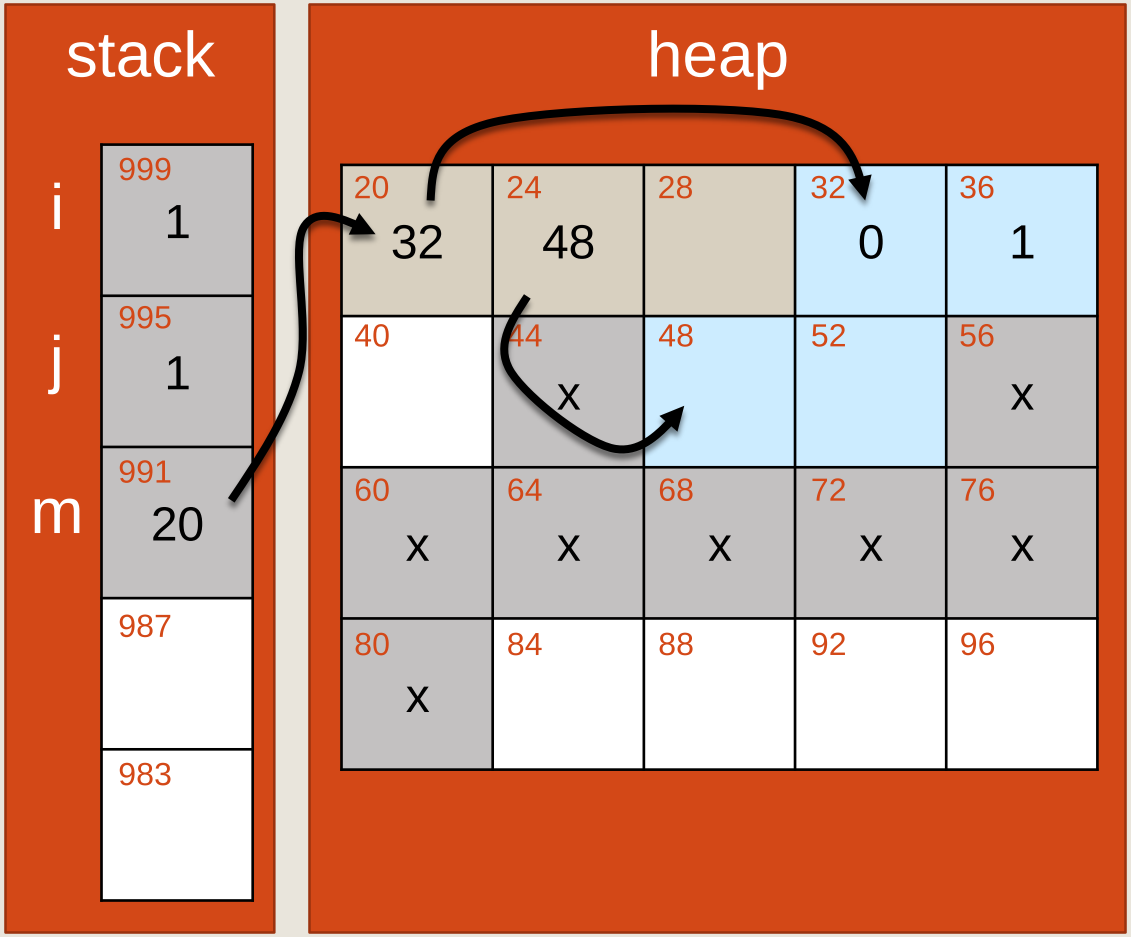

```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

Another memory leak example



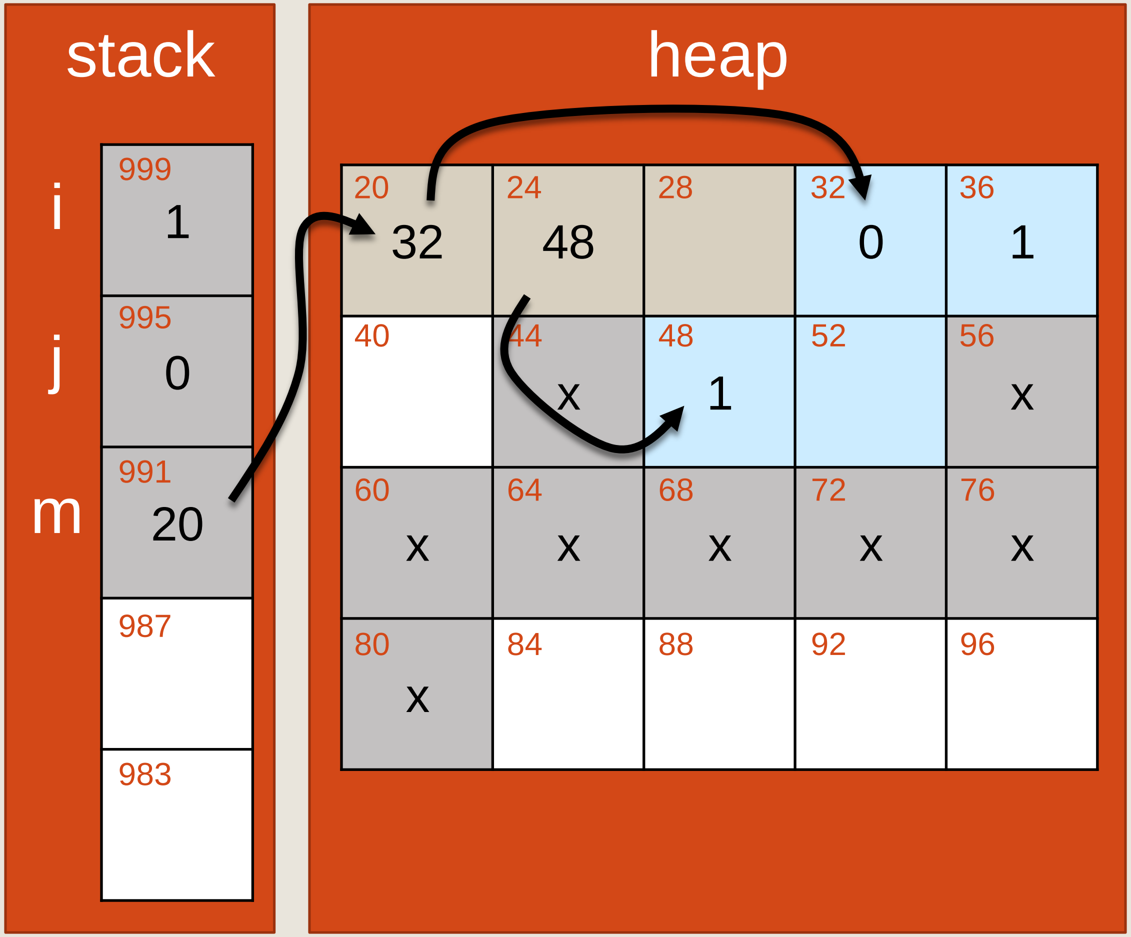

```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

Another memory leak example



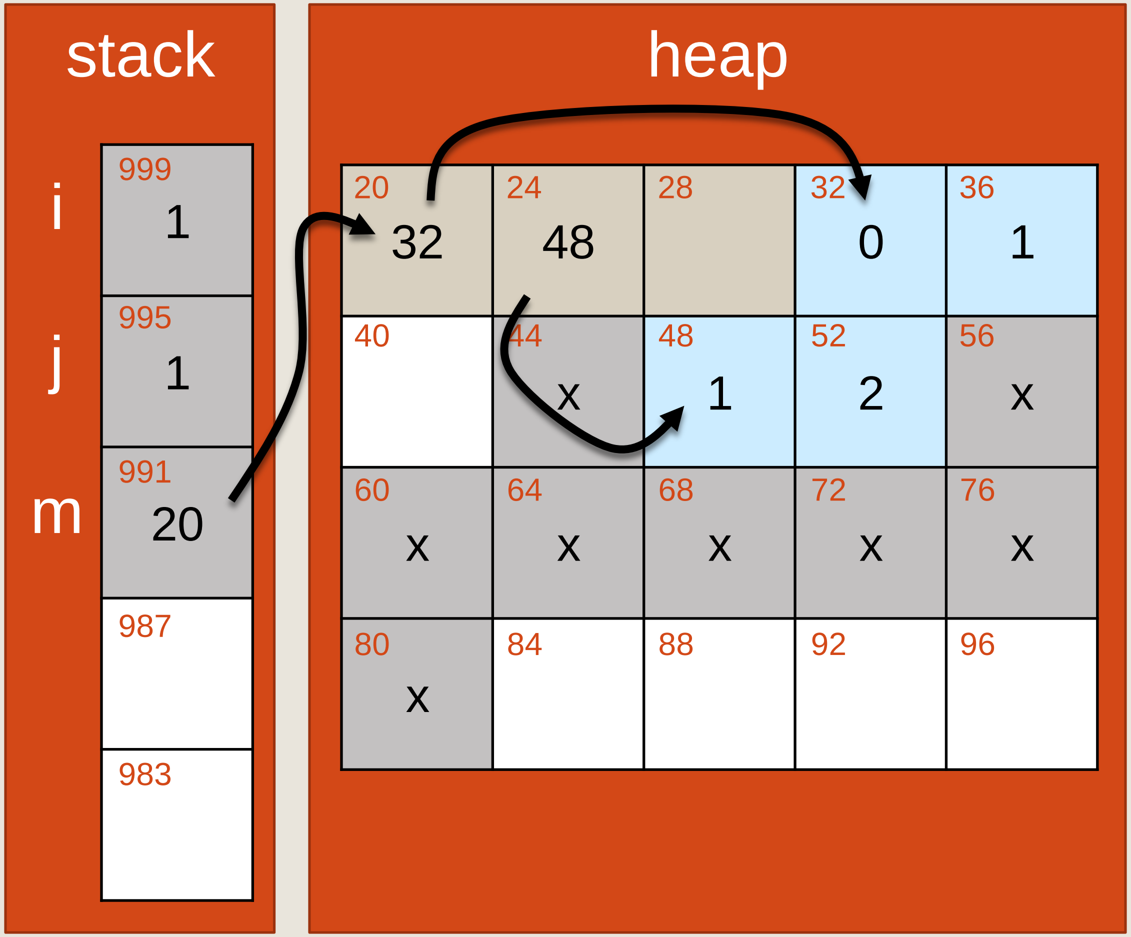

```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

Another memory leak example



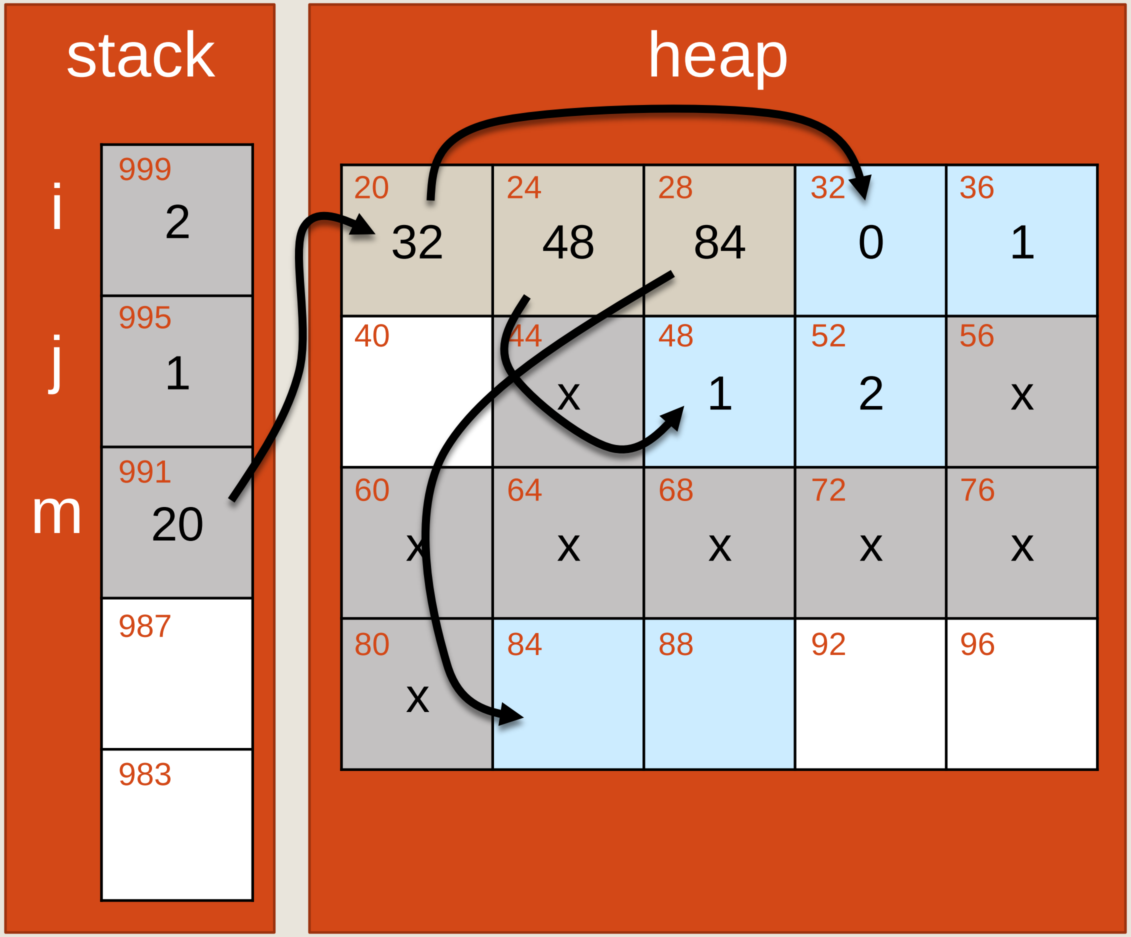

```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

Another memory leak example



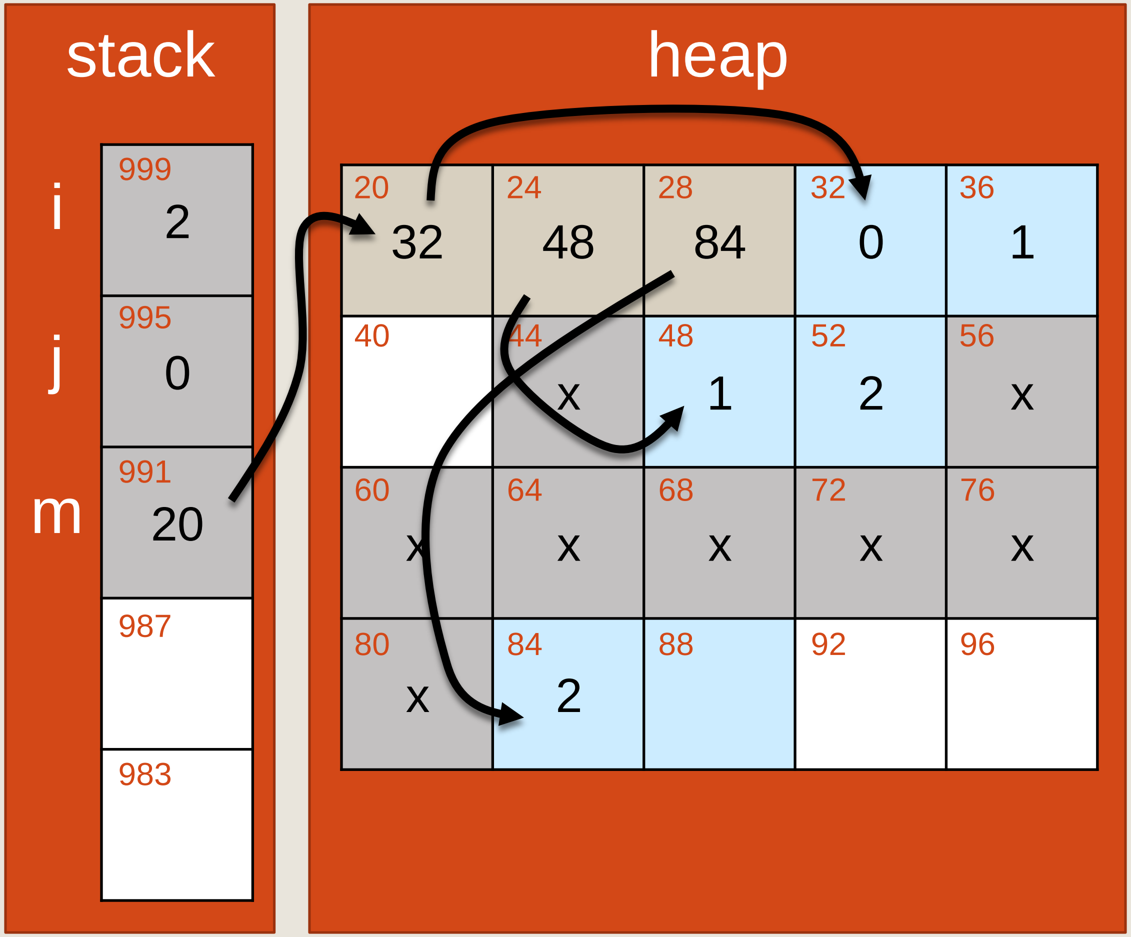

```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

Another memory leak example



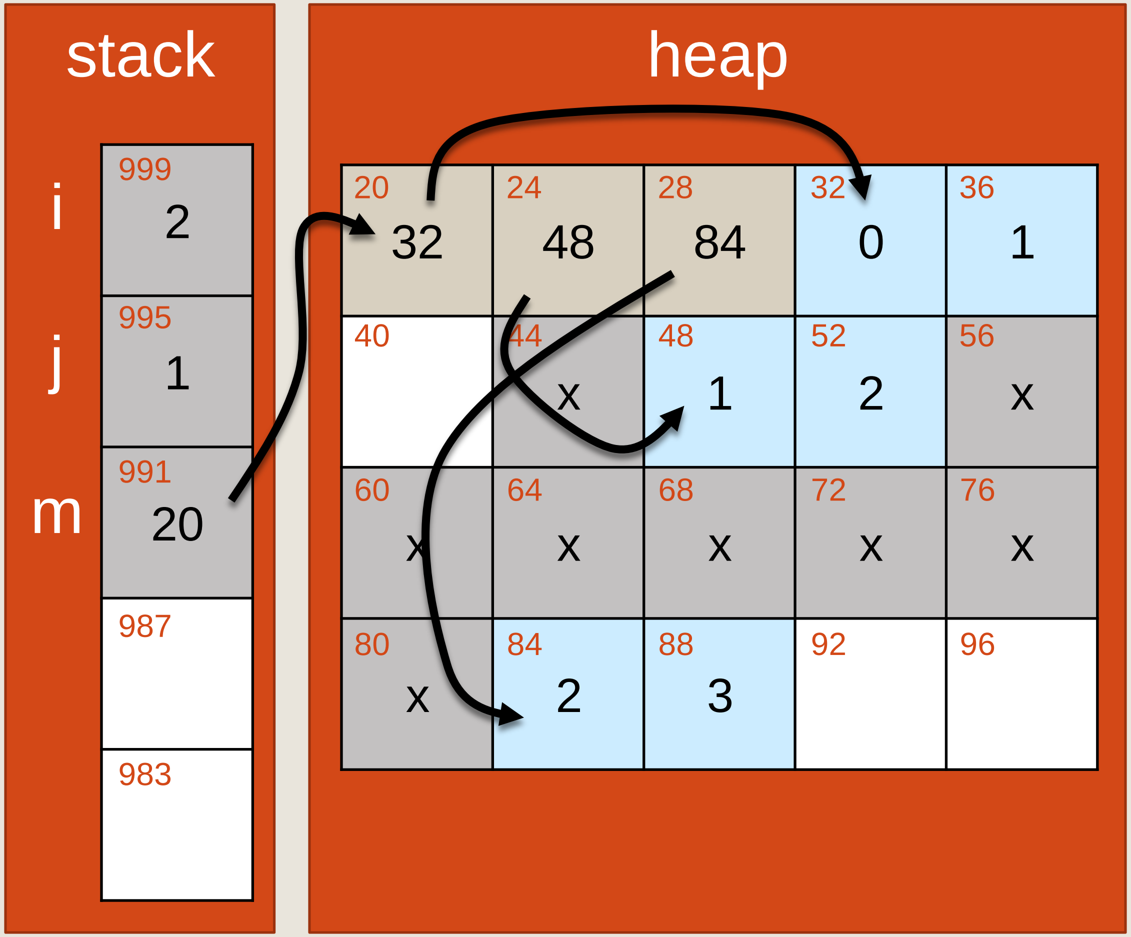

```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

Another memory leak example



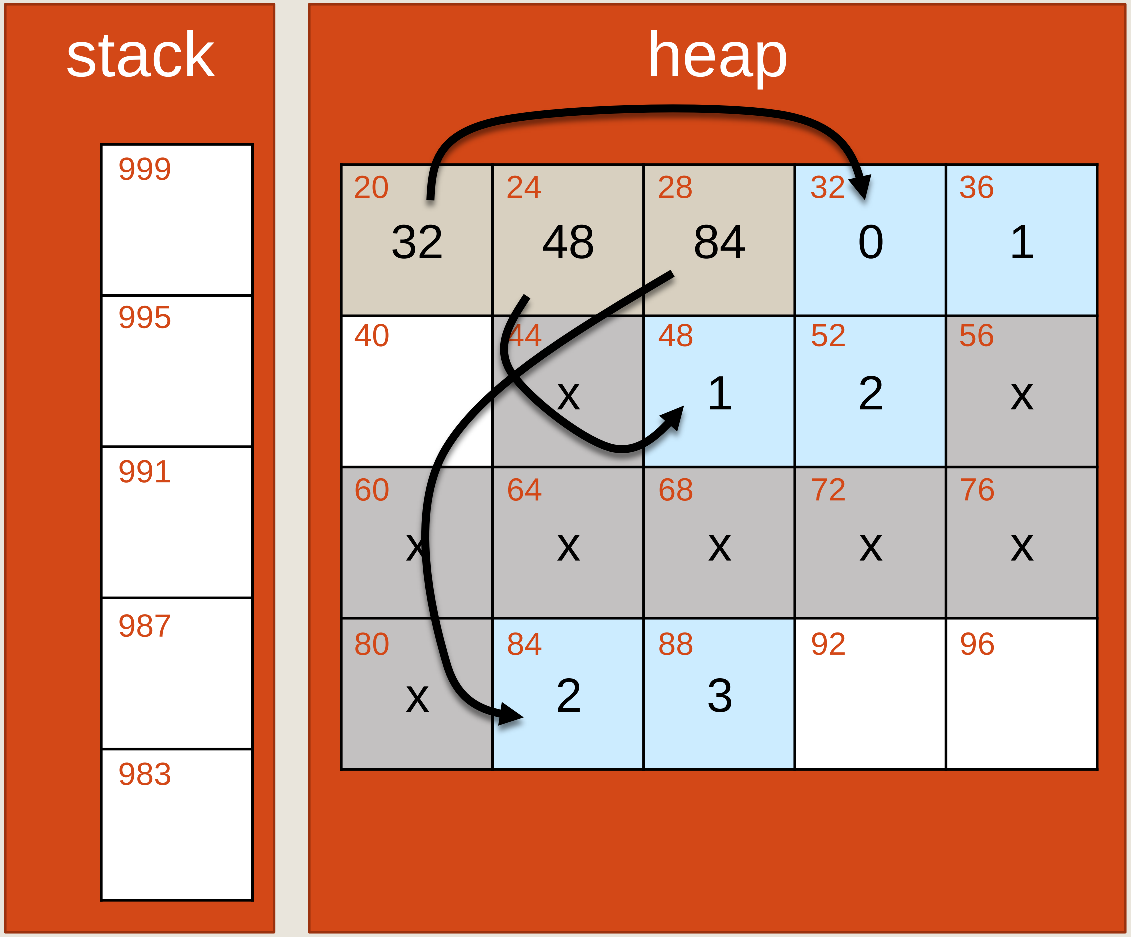

```c
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

Another memory leak example



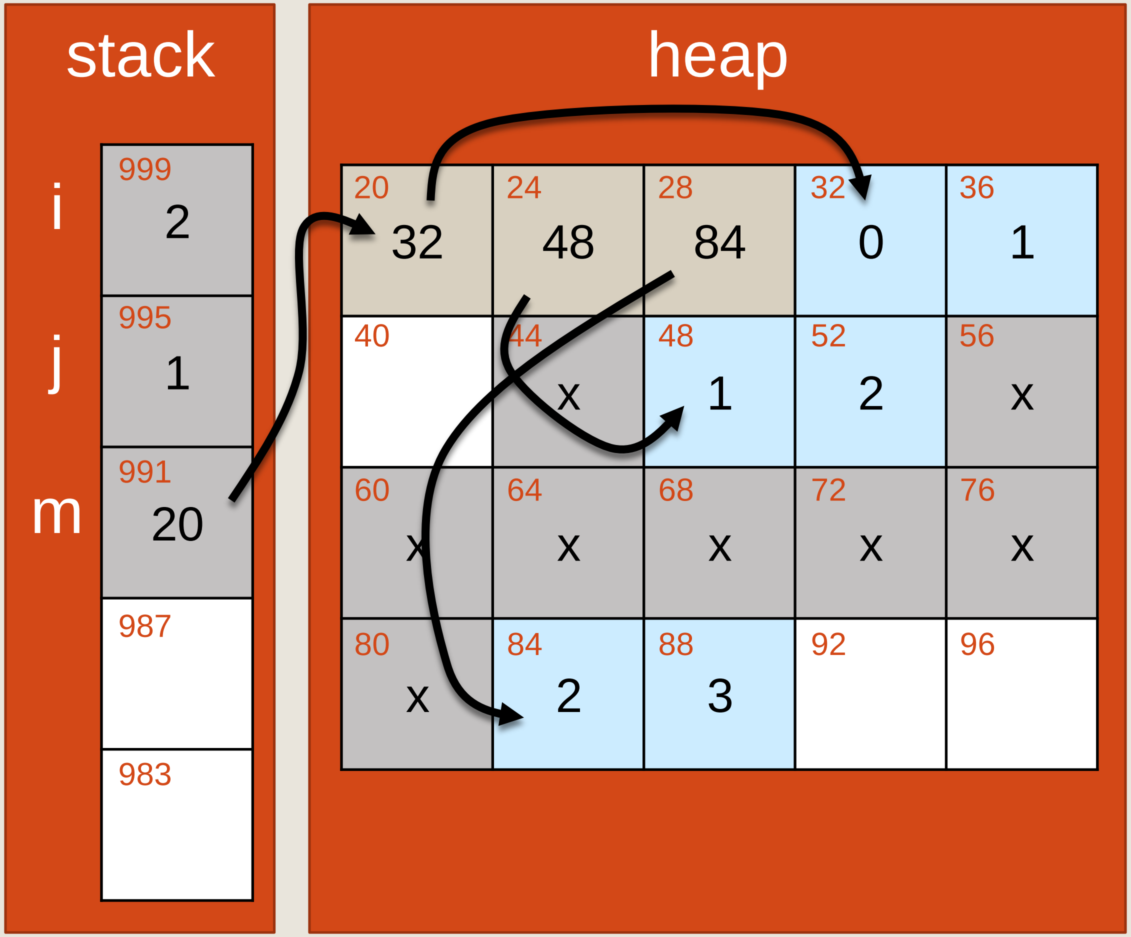

```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

Function ends with memory leak




```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

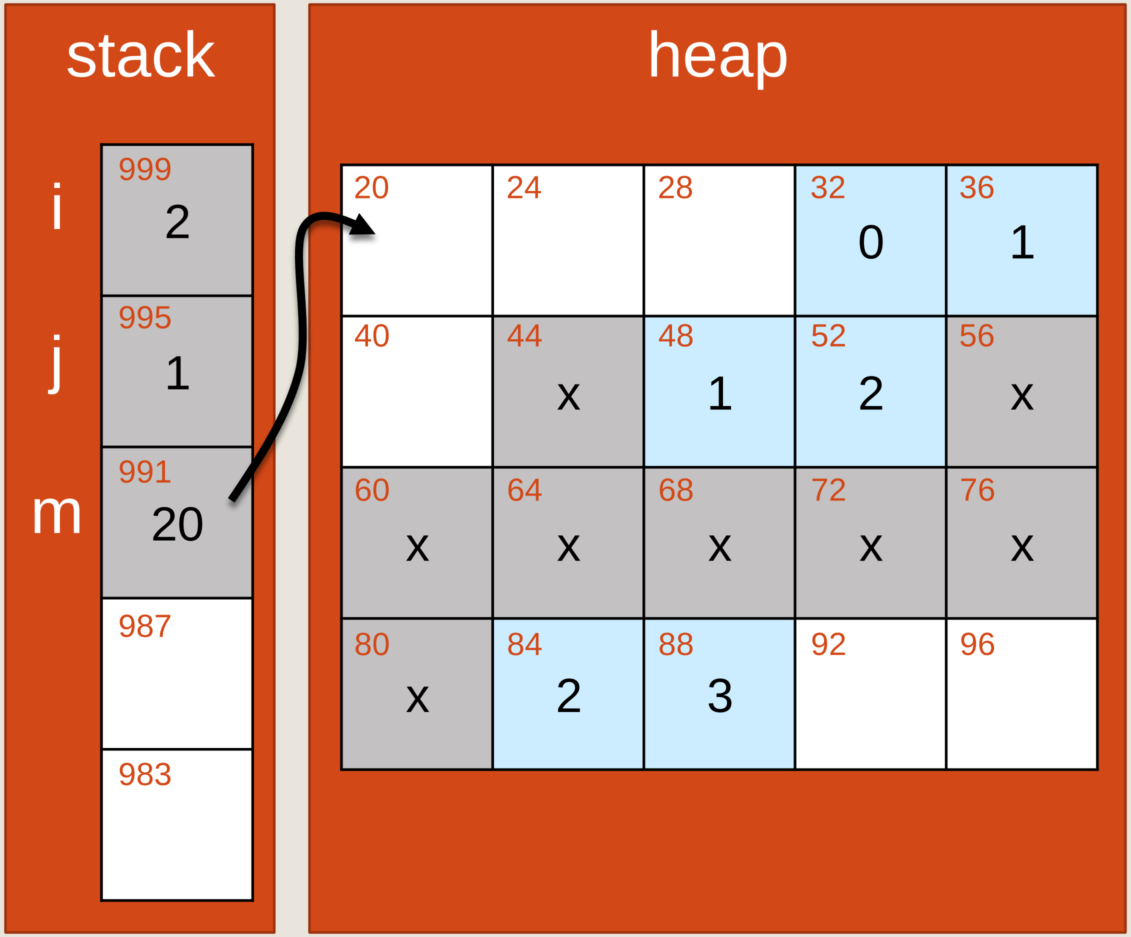
Trying to fix using free




```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

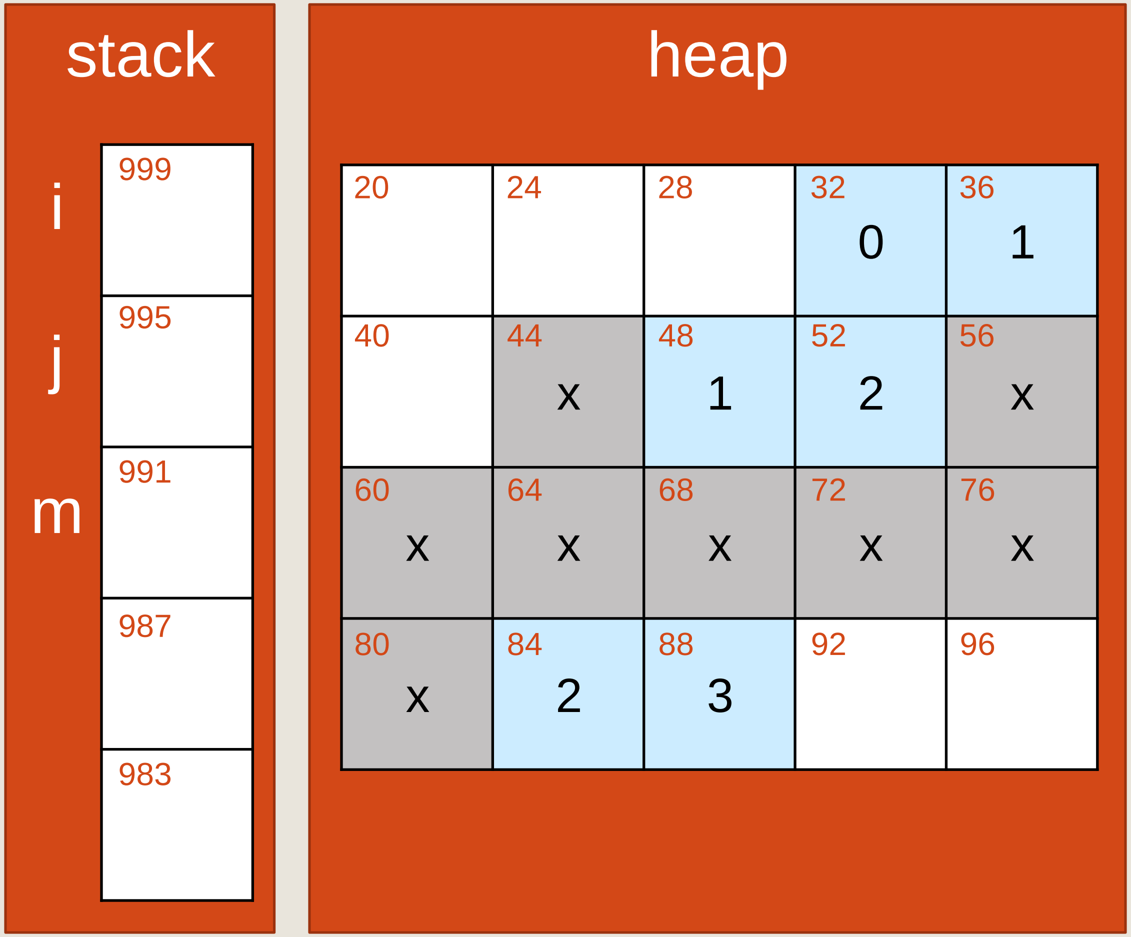
```
free(m);
```

Trying to fix using free



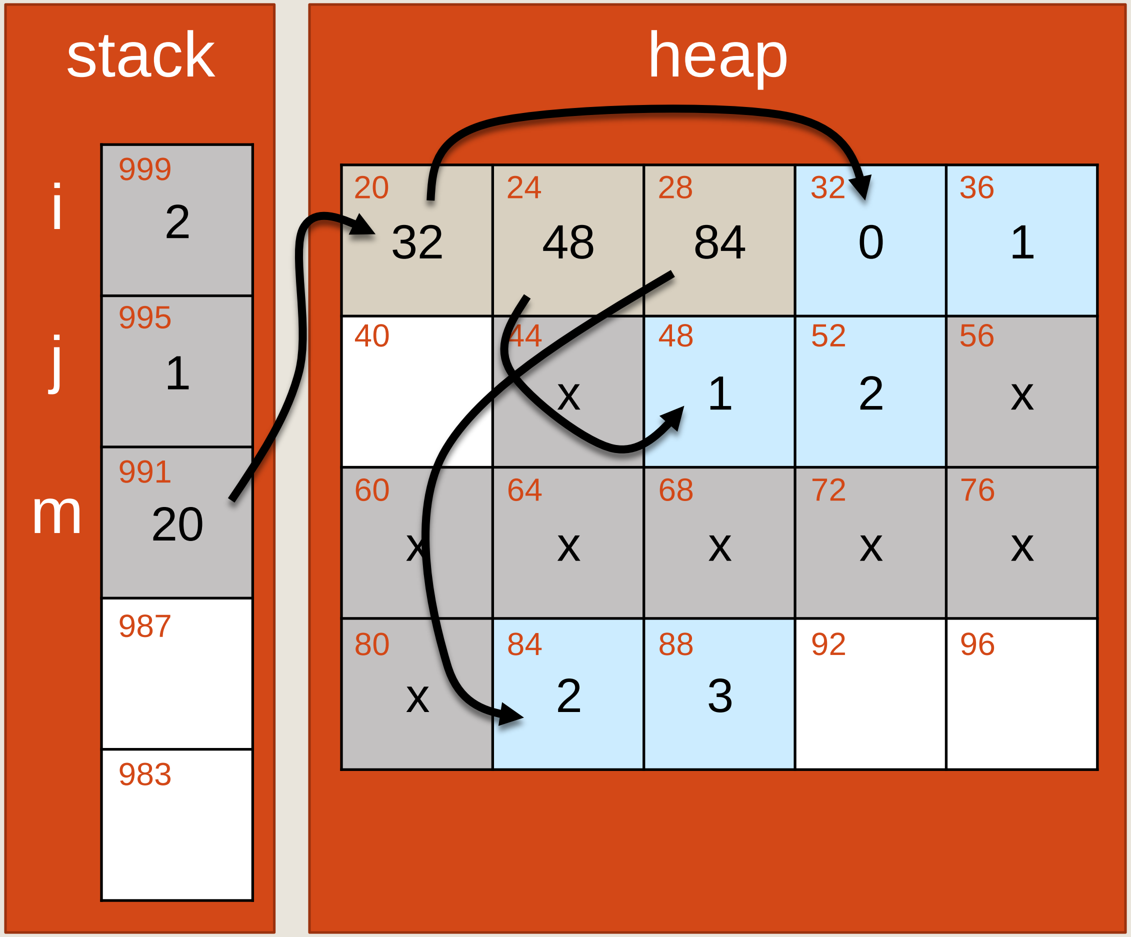

```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}

free(m);
```

Function still ends with memory leak




```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}

free(m);
```

Freeing all allocated memory

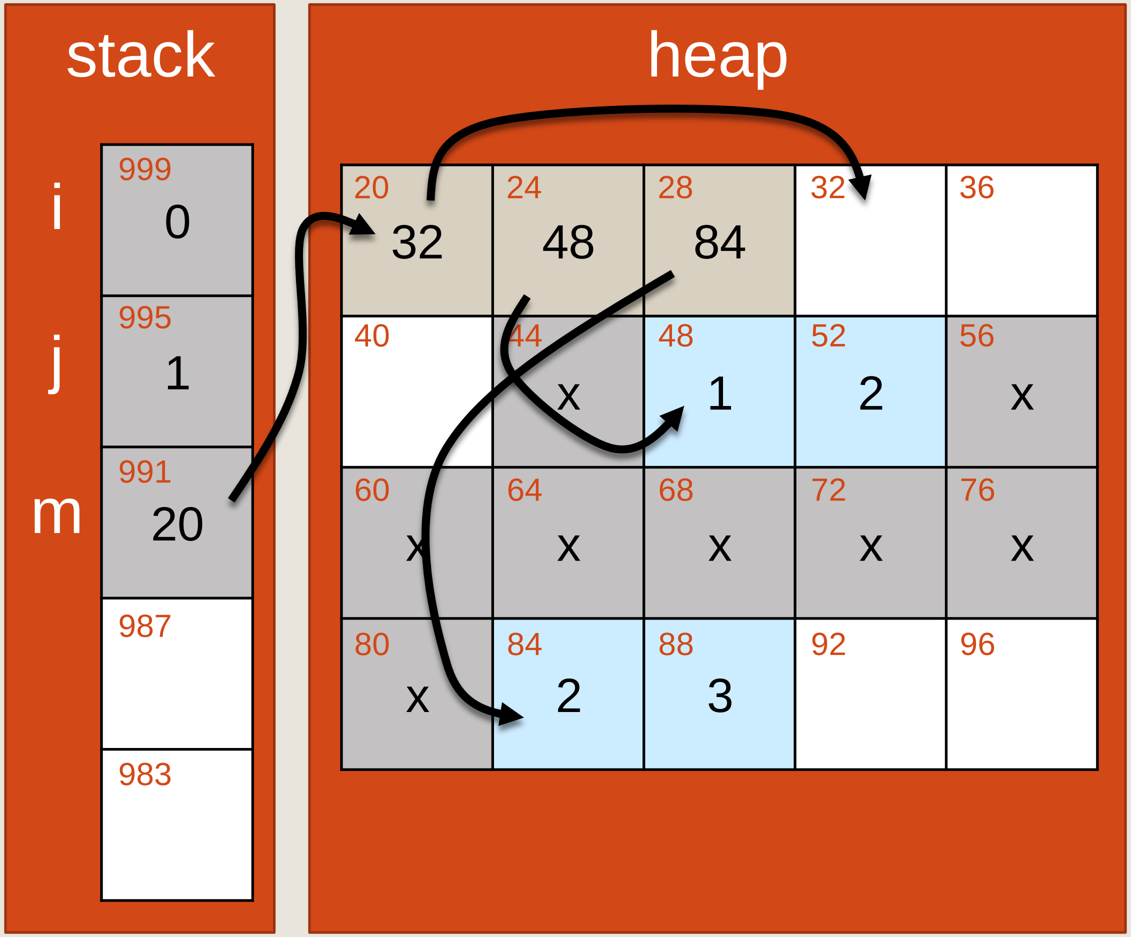



```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

```
for (i=0; i<3; i++) {
  free(m[i]);
}
```

```
free(m);
```

Freeing all allocated memory

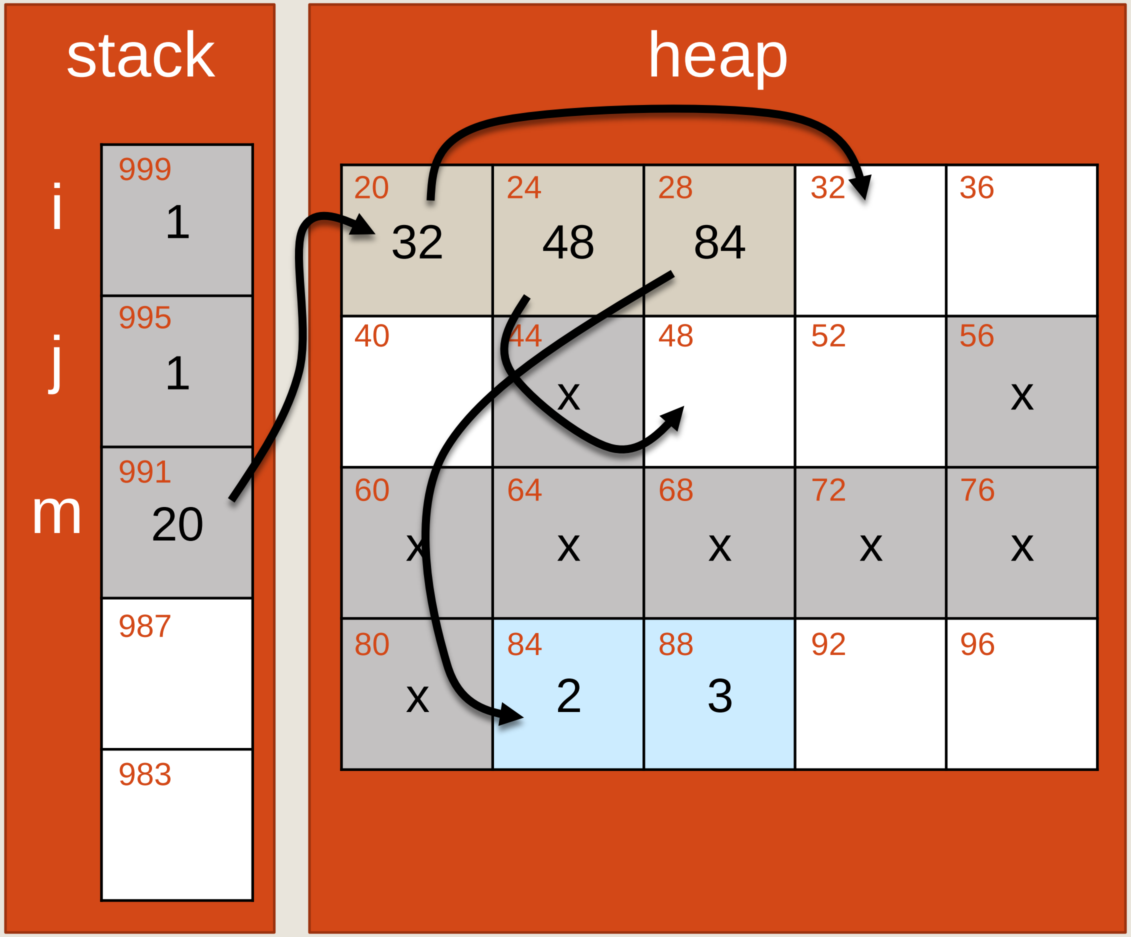



```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

```
for (i=0; i<3; i++) {
  free(m[i]);
}
```

```
free(m);
```

Freeing all allocated memory

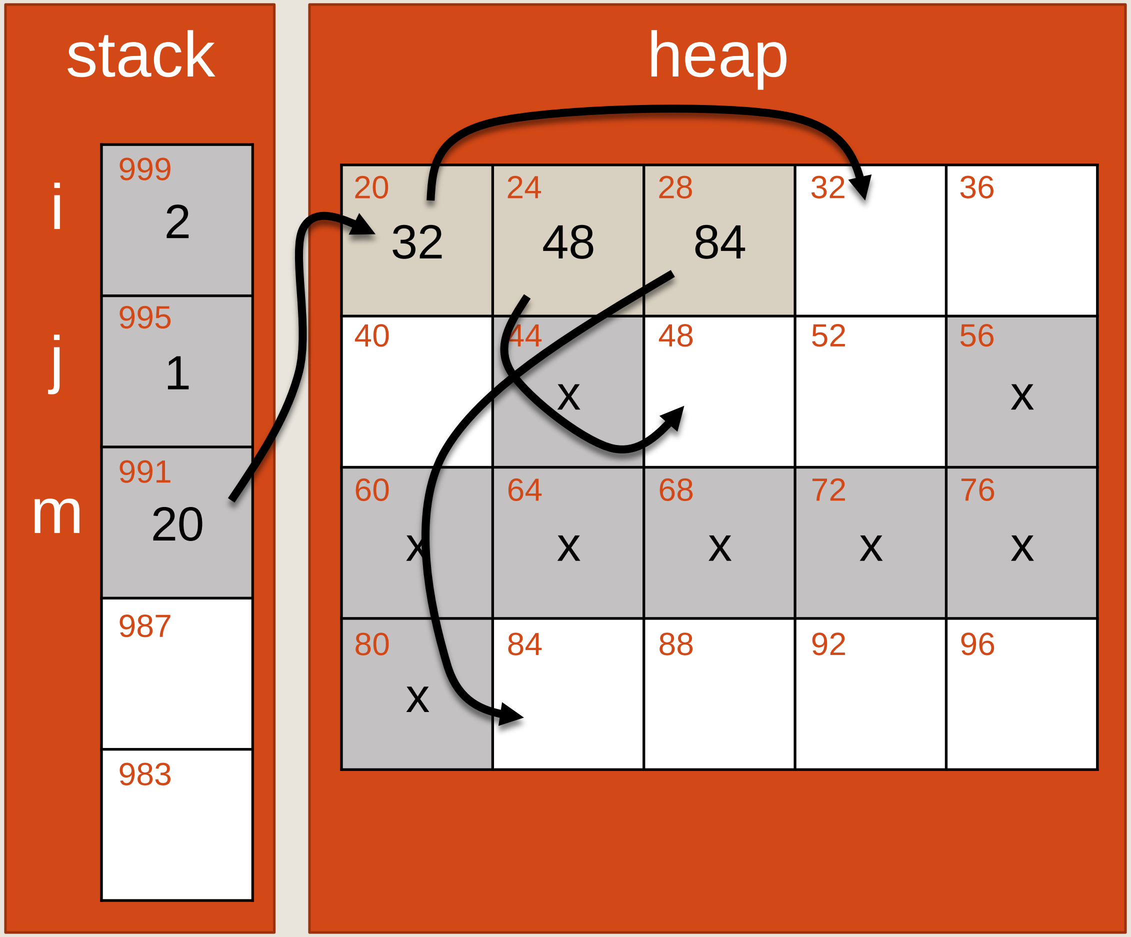



```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

```
for (i=0; i<3; i++) {
  free(m[i]);
}
```

```
free(m);
```

Freeing all allocated memory

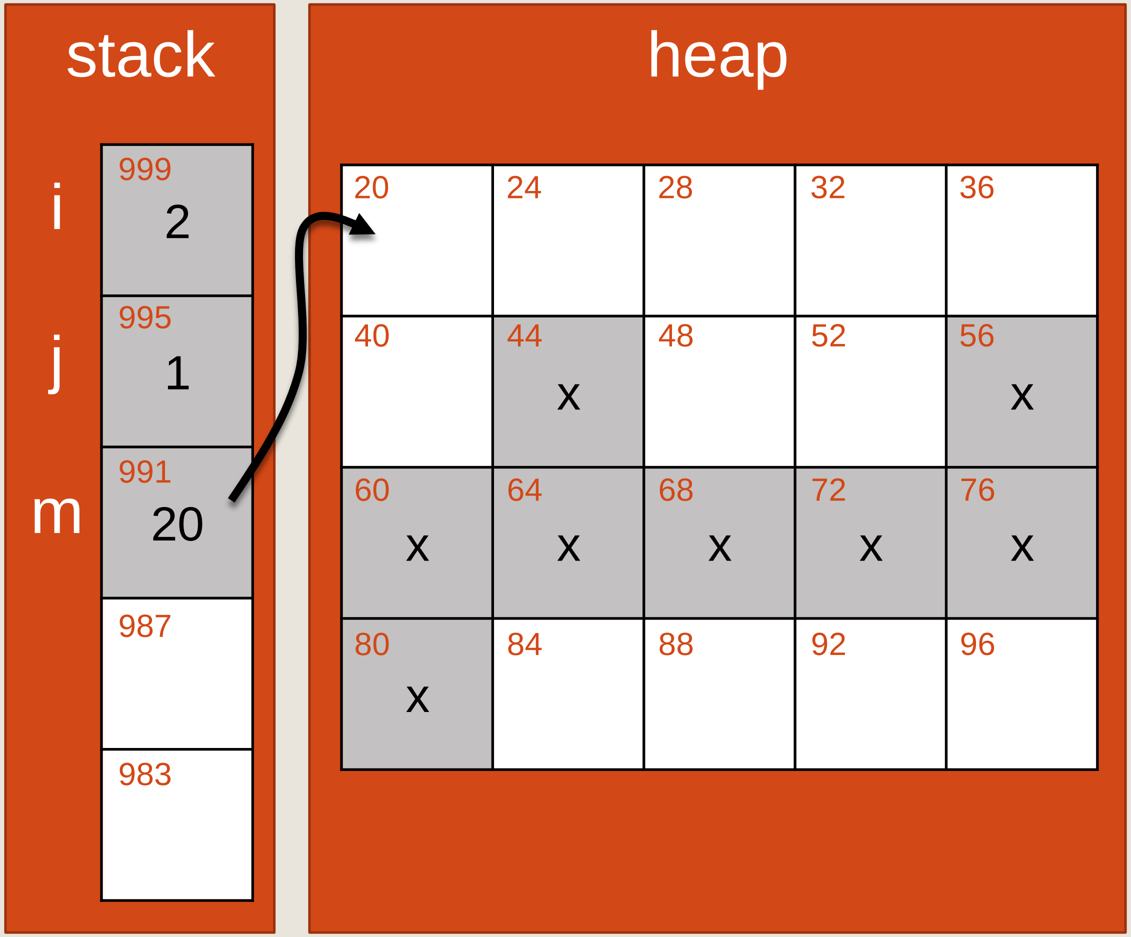



```c
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}
```

```c
for (i=0; i<3; i++) {
  free(m[i]);
}
```

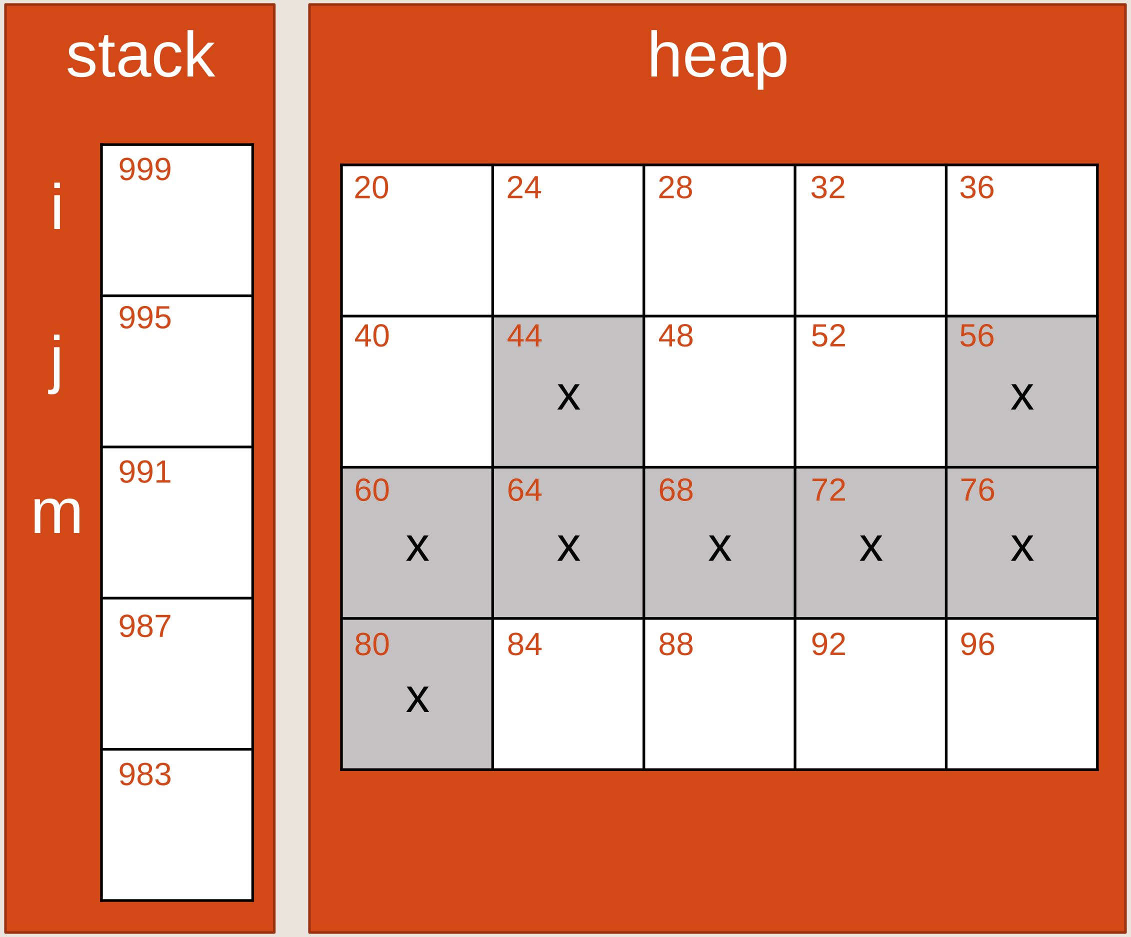
```c
free(m);
```

Freeing all allocated memory




```c
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}

for (i=0; i<3; i++) {
  free(m[i]);
}

free(m);
```

Now function ends with **no** memory leak




```
int i, j;

int **m = (int**)
  malloc(sizeof(int)*3);

for (i=0; i<3; i++) {

  m[i] = (int*)
    malloc(sizeof(int)*2);

  for (j=0; j<2; j++) {

    m[i][j] = i + j;

  }
}

for (i=0; i<3; i++) {
  free(m[i]);
}

free(m);
```

Program Design

Interfaces

A definition of a set of functions that provide a coherent module (or library)

- Data structure (e.g., list, binary tree)
- User interface (e.g., drawing graphics)
- Communication (e.g., device driver)

Interface - modularity

Hide the details of implementing the module from its usage

- Specification – “what”
- Implementation – “how”

Interface – information hiding

Hide “private” information from outside

- The “outside” program should not be able to use internal variables of the module
- Crucial for modularity

Resource management

- Define who controls allocation of memory (and other resources)

Example interface - StrStack

A module that allows to maintain a stack of strings

Operations:

- Create new
- Push string
- Pop string
- IsEmpty
- Free

Example interface - StrStack

```
#ifdef _STRSTACK_H
#define _STRSTACK_H
struct StrStack;
typedef struct StrStack StrStack;

StrStack* StrStackNew();

void StrStackFree(StrStack** stack);

// This procedure *does not* duplicate s
void StrStackPush(StrStack* stack, char* s);

// return NULL if the stack is empty
char *StrStackPop( StrStack* stack );

// Check if the stack is empty
int StrStackIsEmpty(StrStack const* stack);

#endif // _STRSTACK_H
```

Implementation of StrStack

Decision #1: data structure

- Linked list
- Array (static? dynamic?)
- Linked list of arrays
- …

We choose linked list for simplicity

Implementation of StrStack

Decision #2: Resource allocation

- Duplicated strings on stack or keep pointer to original?
- If duplicate, who is responsible for freeing them?

We choose not to duplicate --- leave this choice to user of module

Implementation of StrStack

```c
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "StrStack.h"
typedef struct StrStackLink {
   char* str;
   struct StrStackLink *next;
} StrStackLink;

struct StrStack {
   StrStackLink* top;
};

void StrStackFree(StrStack** stack) {
   while (!StrStackIsEmpty(*stack))    {
      StrStackPop(*stack);
   }
   free(*stack);
   *stack=NULL;
}
```

Implementation of StrStack

```
StrStack* StrStackNew() {
   StrStack* stack = (StrStack*) malloc(sizeof(StrStack));
   if (stack != NULL)    {  stack->top = NULL; }
   else {  printf("out of memory, cannot create stack\n"); }
   return stack;
}
int StrStackIsEmpty(StrStack const* stack) {
   assert( stack != NULL );
   return stack->top == NULL;
}
void StrStackPush(StrStack* stack, char* s) {
   assert( stack != NULL );
   StrStackLink *p = (StrStackLink*) malloc(sizeof(StrStackLink));
   if (p == NULL)
   {
      printf("out of memory, cannot push a string to stack\n");
      return;
   }
   p->str = s;
   p->next = stack->top;
   stack->top = p;
}
```

Implementation of StrStack

```
char* StrStackPop(StrStack* stack)
{
   char *s;
   StrStackLink *p;
   assert( stack != NULL );
   if (stack->top == NULL) {
      return NULL;
   }
   s = stack->top->str;
   p = stack->top;
   stack->top = p->next;
   free(p);
   return s;
}
```

Using StrStack

```
#include "StrStack.h"
char * ReadLine() {  ... } //A function to read a line

int main()
{
   char *line;
   StrStack *stack = strStackNew();
   while ((line = readline()) != NULL)
   {
      strStackPush(stack, line);
   }
   while ((line = strStackPop(stack)) != NULL)
   {
      printf("%s\n", line);
      free(line);
   }
   strStackFree(&stack);
   return 0;
}
```

Interface Principles

Hide implementation details

1. Hide data structures
2. Don't provide access to data structures that might be changed in alternative implementation
3. A "visible" detail cannot be later changed without changing code using the interface!

Interface Principles

Use small set of “primitive” actions

1. Provide to maximize functionality with minimal set of operations
2. Do not provide unneeded functions “just because you can”
3. How much functionality? Two approaches: Minimal (For few users, don't waste your time / Maximal (when many users will use it e.g. OS)

Interface Principles

Don’t reach behind the back

1. Do not use global variables unless you must.
2. Don't have unexpected side effects!
3. Use comments if you assume specific order of operations by the user (and force it).

Interface Principle

Consistent Mechanisms

1. Do similar things in a similar way

   - strcpy(dest, source)
   - memcpy(dest, source)

Interface Principle

Resource Management

1. Free resource at the same level it was allocated – the one who allocates the resource is responsible to free it
2. If you have assumptions about resources – specify this clearly

Pointers to functions

Pointers to Functions

C allow to have a pointer to a function:

```
int foo(int x) { ... }
main()
{
   // func is a pointer to a function that
   // returns an int and receives an int
   int (*func)(int);

   func = &foo;
   func = foo; // same

   int x = (*func)(7); // same as x = foo(7)
}
```

Pointers to Functions

C allow to have a pointer to a function:

```
int foo(int x) { ... }
main()
{
   // func is a pointer to a function that
   // returns an int and receives something
   int (*func)();

   func = &foo;
   func = foo; // same

   int x = (*func)(7); // same as x = foo(7)
}
```

What is this syntax?

Function declaration is the same as variable declaration:

```
int a,*pa; //int, pointer to int
int fa(), (*pfa)(); //function, pointer to function

int * pfa() // this is a function
            // returning a pointer to int
```

And `typedef` follows the same syntax:

```
typedef int ta //ta the type of an int
typedef int (*tpfa)(); // tpfa is the type of a
                       // pointer to a
                       // function returning int
```

Function pointers as function arguments

```
//<pt2Func> is a pointer to a function which returns an int and takes a float and two char
void PassPtr(int (*pt2Func)(float,char,char) )
{
      int result = (*pt2Func)(12, 'a', 'b');
}
// execute example code
void Pass_A_Function_Pointer()
{
      PassPtr(&DoIt);
}
```

Function that returns function pointer

// GetPtr1 is a function that gets const char as input and returns pointer to function, that gets two floats as input and returns a float

```
float (*GetPtr1(const char opCode))(float, float)
{
	if(opCode == '+')
		return &Plus;
	else
		return &Minus;
}
```

// define a function pointer and initialize it to NULL

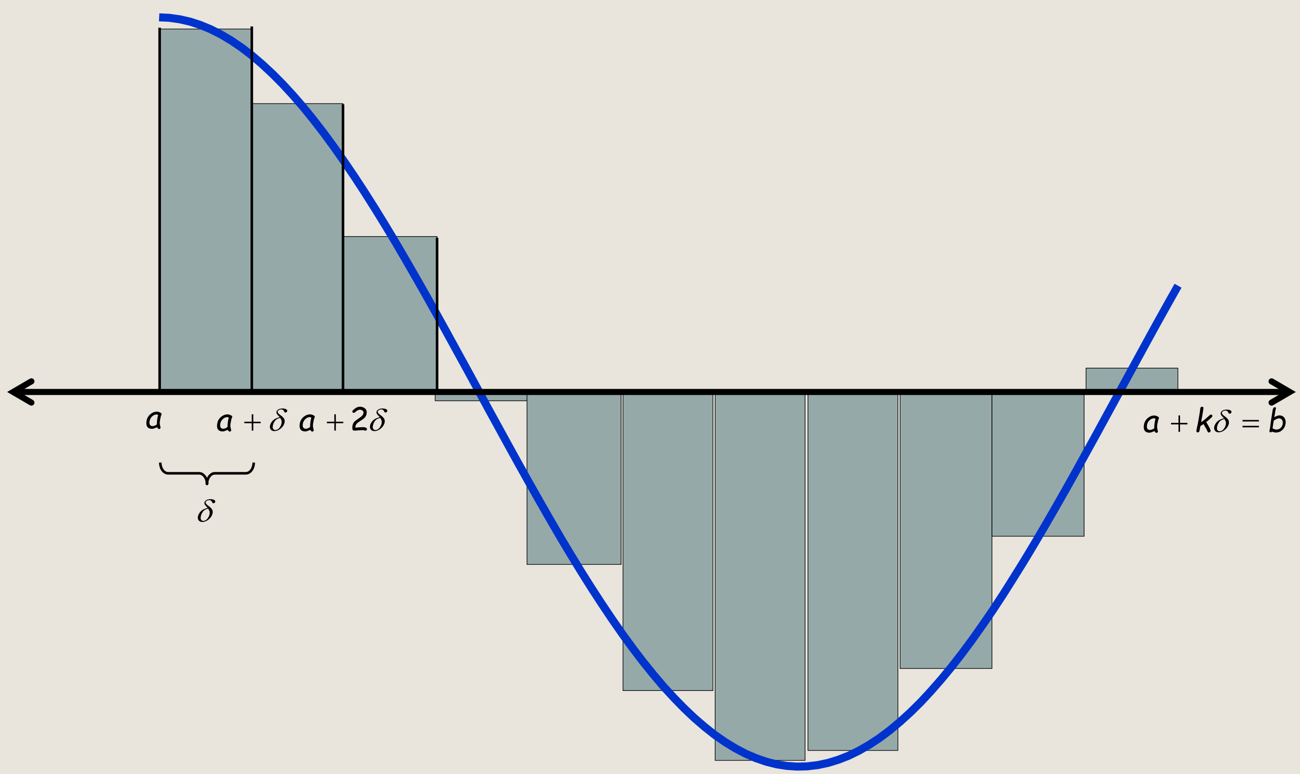
```
float (*pt2Function)(float, float) = NULL;
pt2Function = GetPtr1('+');
Float ans = (*pt2Function)(4.5f,6.5f);
```

typedef is your friend!

```
typedef float(*pt2Func)(float, float);

pt2Func GetPtr2(const char opCode)
{
        if(opCode == '+')
                return &Plus;
        else
                return &Minus;
}
```

Example: Numerical Integrator

$$\int_a^b f(x)dx \approx \sum_{i=1}^{k} \delta f\left(a + (i - \frac{1}{2})\delta\right) \qquad \delta = \frac{b-a}{k}$$



Example: Numerical Integrator

```
double numericalIntegration(
          double a, double b,
          double (*func)(double), int k )
{
   double delta = (b - a)/k;
   double sum = 0;
   for(double x = a+0.5*delta;
          x < b ; x+ =delta)
   {
      sum += (*func)(x);
   }
   return sum*delta;
}
```

Example

Suppose we implement an interface of a list of ints:

```
struct IntList;

// Allocates a new list
IntList* intListNew();
```

Example

```
typedef void (*funcInt)( int x, void* Data );

// Apply Func to each element of the list
void intListMAPCAR(
   IntList* List,
   funcInt Func,
   void* Data
);
```

Example:

```
struct IntList;
typedef struct IntList IntList;

IntList* intListNew();

void intListFree      (IntList* List );
void intListPushFront (IntList* List, int x);
void intListPushBack  (IntList* List, int x);
int  intListPopFront  (IntList* List);
int  intListPopBack   (IntList* List);
int  intListIsEmpty   (IntList const* List);

typedef void (*funcInt)( int x, void* Data );
void intListMAPCAR( IntList* List,
                    funcInt Func, void* Data );
```

Implementation of MAPCAR

```
void intListMAPCAR(
     IntList* List, funcInt Func, void* Data)
{
   IntListNode* p;
   for (p=List->start; p!=NULL; p=p->next)
   {
      (*Func)(p->value, Data);
   }
}
```

Usage if MAPCAR

```
typedef struct ListStats {
   int n;
   int sum;
   int sumOfSquares;
} ListStats;

void RecordStatistics(int x, void* Data) {
   ListStats *s = (ListStats*) Data;
   s->n++;
   s->sum += x;
   s->sumOfSquares += x * x;
}

void intListStats(IntList* List, double* avg, double* var) {
   ListStats stats =  {  0, 0, 0  };
   intListMAPCAR(List, RecordStatistics, &Stats);
   if (stats.n > 0)  {
      *avg = stats.sum / (double) stats.n;
      *var = stats.sumOfSquares / (double) stats.n - (*avg) * (*avg);
   }
   else {
      *avg = 0;
      *var = 0;
   }
}
```

“Generic” interface

Pointers to functions provide a way to write code that receives functions as arguments

MAPCAR is a uniform way of performing computations over list elements - the given function provides the different functional element

Example: qsort

Library procedure:

```
void qsort(
  void* base, size_t n, size_t size,
  int (*compare)(void const*, void const*)
);
// base - start of an array
// n - number of elements
// size - size of each element
// compare - comparison function
```

Using qsort

```
int compareInt(void const *p, void const *q)
{
    int a = *(int const*)p;
    int b = *(int const*) q;
    if( a < b )
    {
        return -1;
    {
    return a > b;
}

int array[10] = { ...} ;
qsort( array, 10, sizeof(int), compareInt );
```

Generic data-structures

Generic data-structures

Generic data-structures are data-structures that can hold data of any type (or, at least, of several types).

- The specific type that the instance of the data-structure holds is determined during run-time.
- The main tool C provides for generic data-structures implementation is:

`void*`

memcpy

Before we begin to discuss implementation of generic stack, let us introduce the function `memcpy`.

Prototype:

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

- `memcpy` copies a block of memory of specific size from one address to another address.
- `memcpy` doesn’t know the *type* of variable(s) being copied.
- The main challenges:
- how to iterate void*.
  - No pointer-arithmetics is defined for void*
- How to derefrence the pointers

Possible implementation of memcpy

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

  char *d = (char*) destination;
  char *s = (char*) source;
  int i;
  for(i=0;i<num;++i)
  {
    //pointer arithmetics for char* is done with
      //units of sizeof(char) == 1 byte
    d[i]=s[i];
  }
}
```

Back to generic stack

We would like our stack to:

- hold any type. (Same type to all of the stack nodes).
- allocate its own memory for the data it holds.

We would like to support the following operations:

- create new stack.
- pop element from the stack head.
- push element to the stack head.
- check if the stack is empty.
- free the stack.

Generic stack underlying data structures:

```
typedef struct Node
{
    void * _data; //pointer to anything
    struct Node * _next;
} Node;

typedef struct Stack
{
    Node * _top;
    size_t _elementSize;//we will need that for
                         //memcpy
} Stack;
```

Generic stack alloc/free:

```
Stack* stackAlloc(size_t elementSize)
{
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    stack->_top = NULL;
    stack->_elementSize = elementSize;
    return stack;
}
void freeStack(Stack** stack){
    Node* p1;
    Node* p2;
    if (!(*stack == NULL)){
      p1= (*stack)->_top;
      while(p1){
        p2= p1;
        p1= p1->_next;
        free(p2->_data);
        free(p2);
      }
      free(*stack);
      *stack = NULL;
    }
}
```

Generic stack push/pop

```
void push(Stack* stack,void *data){
  //you should check allocation success
  Node* node = (Node*)malloc(sizeof(Node));
  node->_data = malloc(stack->_elementSize);
  memcpy(node->_data, data, stack->_elementSize);
  node->_next = stack->_top;
  stack->_top = node;
}
void pop(Stack* stack,void *headData){
  if(stack == NULL){/*print error message and exit*/}
  if(stack->_top == NULL){
    printf("stack is empty\n");
    return;
  }
  Node *node = stack->_top;
  memcpy(headData, node->_data,stack->_elementSize);
  stack->_top = node->_next;
  free(node->_data);
  free(node);
}
```

Using generic stack:

```
int main()
{
  int i, num = 10;
  printf("Generating list with %d ints\n", num);
  Stack *stack = stackAlloc(sizeof(int));
  for(i = 1; i <= num; i++) {
    push(stack,&i);
  }
  for(i = 1; i <= num-2; i++) {
    int headData;
    pop(stack,&headData);
    printf("top value is: %d\n",headData);
  }
  freeStack(&stack);
  return 0;
}
```

Libraries

http://www.adp-gmbh.ch/cpp/gcc/create_lib.html

Libraries

Library is a collection of functions that you may want to use

- written and compiled by you
- or by someone else

Examples:

- C’s standard libraries
- Math library
- Graphic libraries

Libraries may be composed of many different object files

Libraries

2 kinds of libraries:

Static libraries:

- linked with your executable at compilation time
- standard unix suffix: .a (windows: .lib)

Shared libraries:

- loaded by the executable at run-time
- standard unix suffix: .so (windows: .dll)

static vs. shared

Shared libraries pros:

1. Smaller executables
2. Multiple processes share the code
3. No need to re-compile executable when libraries are changed
4. The same executable can run with different libraries

Static libraries pros:

1. Independent of the presence/location of the libraries
2. Independent of the versions of the libraries
3. Less linking overhead on run-time

Using a utilities library

```
#include <utils.h> // Library header
int main ()
{
    foo(); // foo is a function of the
          // 'utils' library
    ...
}
```

Compiling with static libraries

Compilation:

```
gcc -Wall -c –I /usr/lib/include/
    main.c –o main.o
```

Linking:

```
gcc main.o -L /usr/lib/bin/
   -lutils -o app
```

Static libraries – creating your own

Creating the library **`libutils.a`**:

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

- **`ar`** is like tar – archive of object files
- **`rcs`** are 3 relevant flags (read `ar` man pages),

of which ‘`s`’ indicates: create ‘symbol-table’ for the linker.

Using the static library **`libutils.a`**:

```
gcc main.o -L. -lutils -o prog
```

This links to the code in `libutils.a`

Libraries in makefile

```
libutils.a: ${LIBOBJECTS}
    ar rcs libutils.a ${LIBOBJECTS}


...


OBJECTS = main.o another.o
CC = gcc
prog: ${OBJECTS} libutils.a
 ${CC} ${OBJECTS} –L. –lutils –o prog
```

Order is important – put libraries at end

```
gcc main.o -L. -lutils driver.o
     -o app
```

If `driver` tries to use references from `libutils.a` they may not be linked!

The linking process: Objects vs. static libraries

objects:

- The whole object file linked to the executable, even when its functions not used.
- Two function implementations – will cause error.

Libraries:

- Just symbols (functions) which not found in the obj files are linked.
- Two function implementations – first in the obj file, and second in library – The first will be used.
- Order is important – compiler may discard unused references when linking the library.

Dynamic Libraries

Library creation:

Compilation:

```
gcc -Wall –fPIC -c utils.c
```

Linking:

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

- PIC – position independent code.
- On windows you need `__declspec(dllimport)` (feel free to read more about it…)

Dynamic Libraries

Usage:

Linking to:

```
 gcc –Wall main.c -L. –lutils
```

When running, you will need to set

`LD_LIBRARY_PATH=.`

So the shared library can be found, unless it is in the system’s path (the already set value)

Dynamic Libraries

- Why do we link at compile time to dynamically linked library?

- Not real linking, just to check that linking is possible.
- Actual linking is done in run time:
  => Need to know how to find in runtime. Should be in the dynamic library search path (e.g. c:\windows\system), or set **`LD_LIBRARY_PATH`**.

Errors handling

The problem:

```
include <stdio.h>
void sophisticatedAlgorithm (char* name)
{
   FILE * fd = fopen (name); // using the file
                             // for an algorithm
   // ...
}
int main()
{
   char name[100];
   scanf ("%s", name);
   sophisticatedAlgorithm (name);
   // ...
}
```

OOP (java / C++) solution:

```
try
 {
    FileInputStream fstream = new
                   FileInputStream(name);
    DataInputStream in = new
                   DataInputStream(fstream);
    while (in.available() !=0)
    {
        System.out.println (in.readLine());
    }
    in.close();
 }
 catch (Exception e)
 {
   System.err.println("File input error");
 }
```

How can it be done in C?

Errors types:

- Bugs:
  - Deterministic errors.
  - Not dependant on the program inputs.
  - You assert they will never happen.
- Exceptions:
  - Originate from the program inputs and environment.
    - Input streams
    - Memory allocations
    - ...
  - May happened from time to time

Catching bugs -- assert

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

If the program violates the condition, then

```
assertion "x >= 0" failed: file "Sqrt.c",
line 7 <exception>
```

The exception allows to catch the event in the debugger

Using assert:

- Terminates the program continuation.
- Good for debugging and logic examination.
- User of library function can not decide what to do in case of an error.
- Discarded in NDEBUG mode

C exception handling strategies:

Detecting the errors:

1. Catch the exception before it occurred.
2. Use function return value to indicate errors.
3. Use global variables to indicate errors occurred and their identity.
4. Develop an ‘exception-catching- like’ mechanism

   (will not be discussed in this course).

Handling the errors:

- May include printing error massages.
- May include program termination.

Printing error messages:

- Use the *standard errors stream (stderr)*

- Relevant functions (examples in the following slides):
  - `fprintf(stderr, “format string”, …)`
  - `perror`
  - `strerror (with errno)`

- `stdout` and `stderr` can be redirected separately:
  - `~% (myProg > outputFile) >& errorFile`

Return status

- ‘0’ -- success
- other values – failure (most common: ‘1’/ ’-1’)

- `stdlib.h` defines the macros:
  - `#define EXIT_SUCCESS 0`
  - `#define EXIT_FAILURE 1`

exit()

- `exit(int status)` terminates the program in case of an exception:

```
#include <stdio.h>      /* fprintf, fopen */
#include <stdlib.h>     /* exit, EXIT_FAILURE */
int readFile (){
  FILE * pFile;
  pFile = fopen ("myfile.txt","r");
  if (pFile==NULL){
    fprintf (stderr, "Error opening file");
    exit (EXIT_FAILURE);
  }
  else{
    /* file operations here */
  }
  return EXIT_SUCCESS;
}
int main(){
  int status = readFile();
  return status;}
```

Find the error before it occurred

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int dividend = 20;
   int divisor = 0;
   int quotient;
   if( divisor == 0){
      fprintf(stderr, "Division by zero! Exiting...\n");
      return 1;
   }
   quotient = divide(dividend, divisor);
   fprintf(stderr, "Value of quotient : %d\n", quotient );
   return 0;
}
```

Return values:

```
#include <stdio.h>
int sophisticatedAlgorithm (char* name)
{
  FILE * fd = fopen (name);
  if( in == NULL )
  {
     return -1; // indicate an abnormal
                // termination of the
               // function
  }
 // do your sophisticated stuff here
  return 0; // indicate a normal
            // termination of the function
}
```

Special return values indicate errors:

```
int main()
{
    if(sophisticatedAlgorithm(name) == -1)
    {
        // the exceptional case
    }
    else
    {
        // the normal case
    }
}
```

Return values:

User of a library function can decide what to do in case of an error.

But:

- We may have no *free* value to indicate an error
- We need a separate value for each error type.
- Requires checking after each function call.
  - No separation of regular code from the errors checking

Modify a global variable

- In case no return value is free, errors are indicated by a global variable.

```
int g_divisionError;
int divide(int dividend, int divisor){
   g_divisionError = 0;
   if( divisor == 0){
      g_divisionError = 1;
      return 1;
   }
   return dividend / divisor;
}
int main(){
  int c = divide(20,0);
  if( g_divisonError == 1){
      fprintf(stderr, "Division by zero! Exiting...\n");
      return EXIT_FAILURE;
}
```

Modify a local variable using a pointer

- In case no return value is free, we can use a combination of return value (usually for error indication), and an address of a given variable for return value.

```
int divide(int dividend, int divisor, int *quotient){
    if( divisor == 0){
        return 1;
    }
    *quotient = dividend / divisor;
    return 0;
}
int main(){
  int c;
  int div_error = divide(20,0,&c);
  if(div_error == 1){
      fprintf(stderr, "Division by zero! Exiting...\n");
      return EXIT_FAILURE;
}
```

The standard library approach:

Combination of return value and global variable to indicate errors:

The idea: Separate between function return code and error description.

- Function return just 0 in case of success or -1 in case of error.
- A global variable holds the specific error code (or message) describes the occurred error.

Example:

```
#include <stdio.h> // for perror
#include <stdlib.h>
#include <errno.h> // for the global variable
                   // errno
#include <string.h> // for strerror
const char *FILE_NAME =
"/tmp/this_file_does_not_exist.haha";
```

Example:

```
int main( int argc, char **argv )
{
    int fd = 0;
    fd = open( FILE_NAME, O_RDONLY, 0644 );
    if( fd < 0 )
    {
        // Error, as expected.
        perror( "Error opening file" );
        printf( "Error opening file: %s\n",
                strerror( errno ) );
    }
    return EXIT_SUCCESS;
}
```

C exceptions:

google: C exceptions will lead to many useful C libraries that implement some kind of exceptions, very similar to java/c++