Heap-Allocated Memory and Structs

Lecture Outline

- **Heap-allocated Memory**
  - **`malloc()`, `calloc()`, `realloc()` and `free()`**
  - **Memory leaks**
- `structs` and `typedef`

Memory Allocation So Far

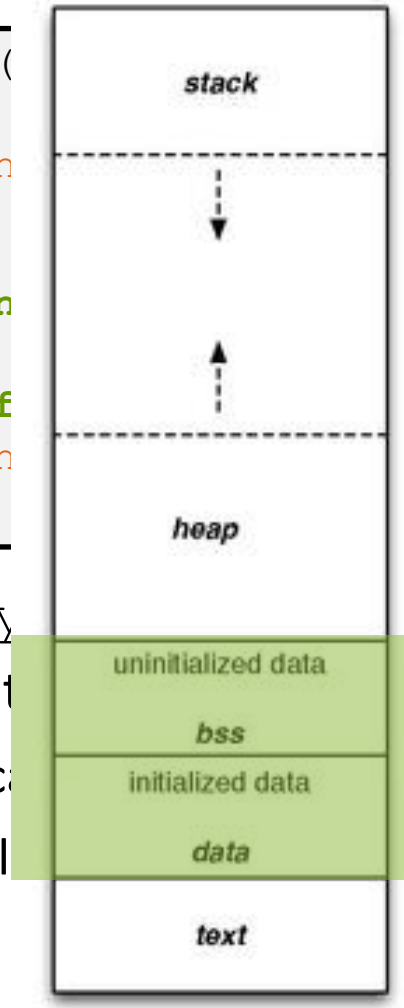
- So far, we have seen two kinds of memory allocation:

```c
int counter = 0;     // global var

int main(int argc, char** argv) {
  counter++;
  printf("count = %d\n",counter);
  return EXIT_SUCCESS;
}
```

```c
int foo(
  int x
  return
}

int main
  int y
  printf
  return
}
```

local var

* argv) {
local var



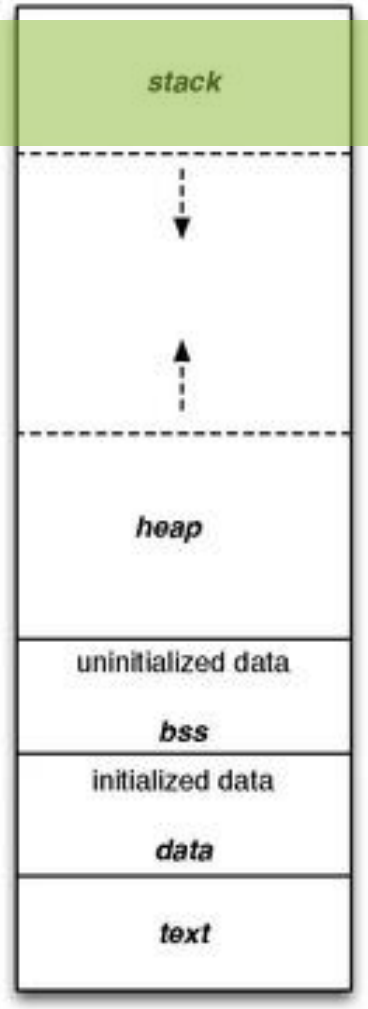
- `counter` is ***statically***-allocated
  - Allocated when program is loaded
  - Deallocated when process gets reaped
  - Stored in the data Segment

- `a, x, y` ***ly***-allocat
  - Alloc is called
  - Deall on returns

Memory Allocation So Far

- So far, we have seen two kinds of memory allocation:

```
int cour                          lobal var

int mair                        * argv) {
  counte
  printf                        counter);
  returr
}
```



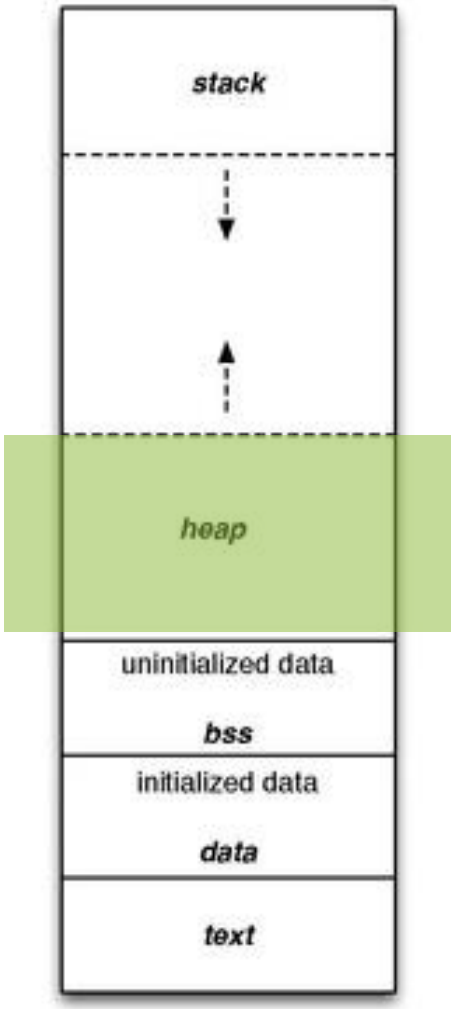

```
int foo(int a) {
  int x = a + 1;      // local var
  return x;
}

int main(int argc, char** argv) {
  int y = foo(10);    // local var
  printf("y = %d\n",y);
  return EXIT_SUCCESS;
}
```

- coun … allocated
  - Alloc … n is loaded
  - Deall … ss gets reap…
  - Store … f Data Segm…

- a, x, y are ***automatically***-allocated
  - Allocated when function is called
  - Deallocated when function returns
  - Store in stack

Dynamic Allocation

- Situations where static and automatic allocation aren't sufficient:
  - We need memory that persists across multiple function calls but not the whole lifetime of the program
  - We need more memory than that can fit on the Stack
  - We need memory whose size is not known in advance to the caller

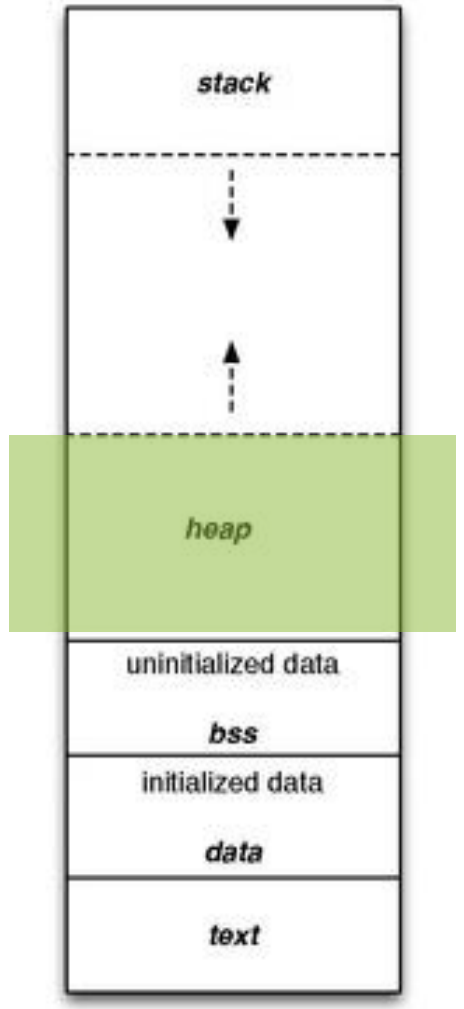


Dynamic Allocation

- Situations where static and automatic allocation aren't sufficient:
  - We need memory that persists across multiple function calls but not the whole lifetime of the program
  - We need more memory than that can fit on the Stack
  - We need memory whose size is not known in advance to the caller

```
// this is pseudo-C code
char* ReadFile(char* filename) {
  int size = GetFileSize(filename);
  char* buffer = AllocateMem(size);

  ReadFileIntoBuffer(filename, buffer);
  return buffer;
}
```

Example: Reading from a file, which has an unknown size at compilation time.



Dynamic Allocation

- What we want is ***dynamically***-allocated memory
  - Your program explicitly requests a new block of memory
    - The language allocates it at runtime, perhaps with help from OS
  - Dynamically-allocated memory persists until either:
    - Your code explicitly deallocated it (*manual memory management*)
    - A garbage collector collects it (*automatic memory management*)

- C requires you to manually manage memory
  - Gives you more control, but causes headaches

Where is it going to be stored?

```
int i;
int main()
{
    int j;
    int *k = (int *) malloc (sizeof(int));
}
```

i is global variable and it is uninitialized so it is stored in Data Section
j is local in main() so it is stored in stack frame
*k is dynamically allocated so it is stored on Heap Segment

Aside: NULL

- `NULL` is a memory location that is guaranteed to be invalid
  - In C on Linux, `NULL` is `0x0` and an attempt to dereference `NULL` *causes a segmentation fault*
- Useful as an indicator of an uninitialized (or currently unused) pointer or allocation error

```
int* p = NULL;
```

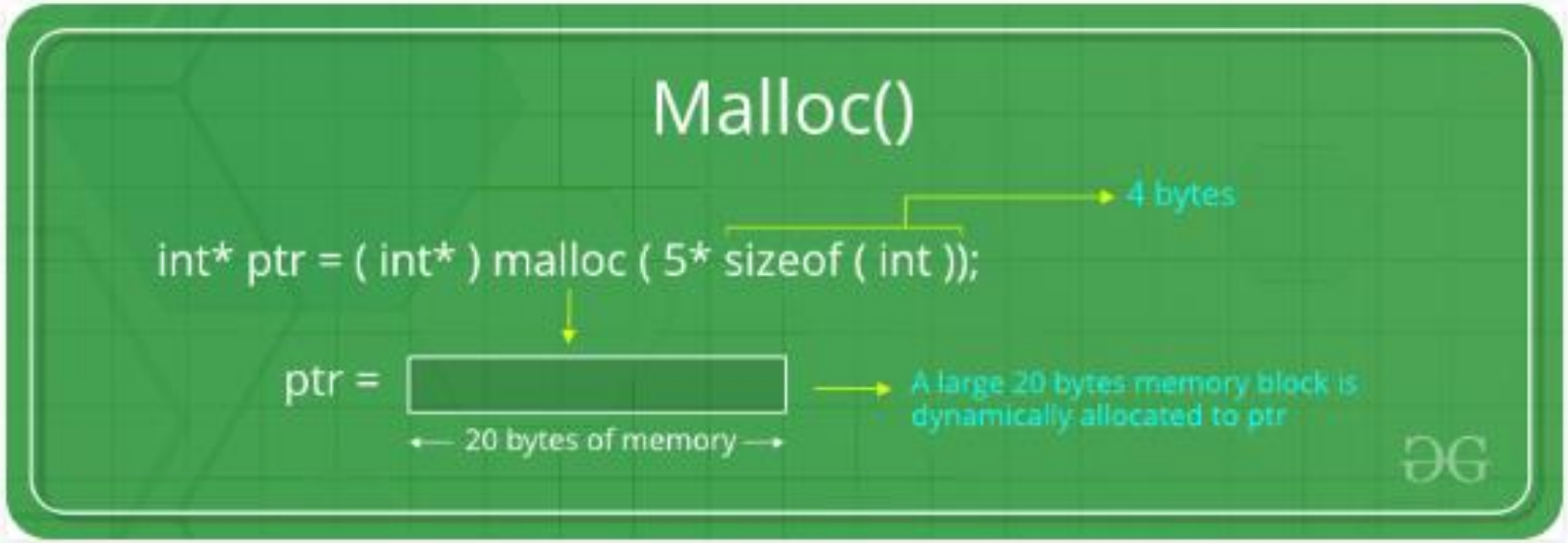
malloc()

- General usage: `var = (type*) malloc(size in bytes)`
  - Malloc returns void*. It is a good practice to explicitly cast.
- **malloc** allocates a block of memory of the requested size
  - Returns a pointer to the first byte of that memory
    - And returns NULL if the memory allocation failed!
  - You should assume that the memory initially contains garbage
  - You'll typically use sizeof to calculate the size you need

```
// allocate a 10-float array
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL) {
  // print error message
}
...   // do stuff with arr
```

https://ide.geeksforgeeks.org/5ageCRPcV7

malloc()



- https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/
- will allocate 20 bytes of memory.
- the pointer ptr holds the address of the first byte in the allocated memory
- Example: https://ide.geeksforgeeks.org/beIOPag6jV

calloc()

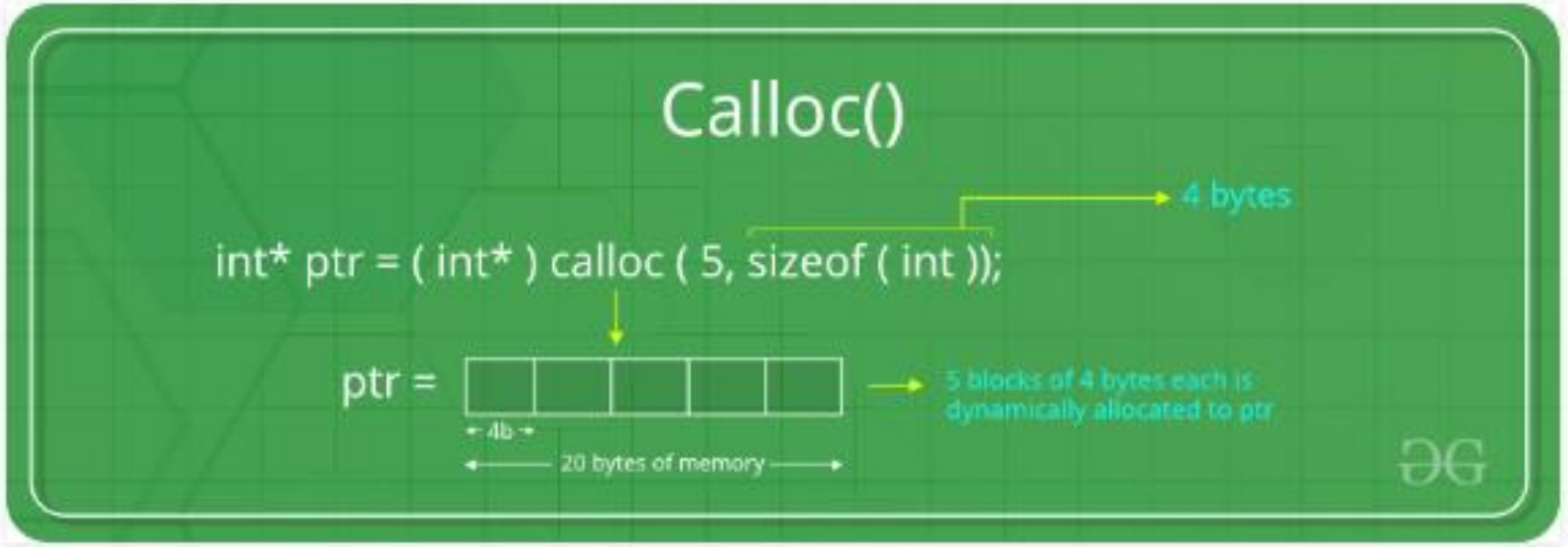
- General usage:

```
var = (type*) calloc(num, bytes per element)
```

- Like **malloc**, but also zeros out the block of memory
  - Helpful when zero-initialization wanted
  - Slightly slower; but useful for non-performance-critical code
  - **malloc** and **calloc** are found in stdlib.h

```
// allocate a 10-double array
double* arr = (double*) calloc(10, sizeof(double));
if (arr == NULL) {
  //error
}
...   // do stuff with arr
```

calloc()



https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

ptr = (float*) calloc(5, sizeof(float));

allocates contiguous space in memory for 5 elements each with the size of float and initializes to 0

Example: https://ide.geeksforgeeks.org/oadeFmYcaS

What is the return type of malloc() or calloc()

- **(A)** void *
  (B) Pointer
  (C) void **
  (D) int *

realloc()

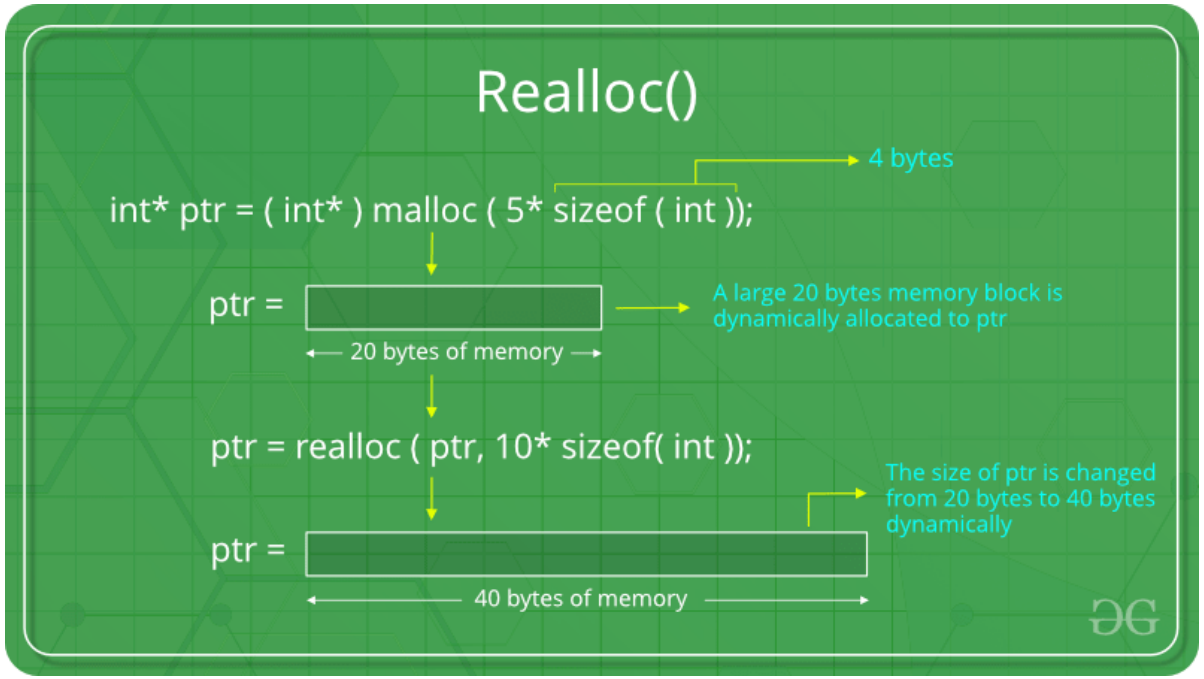
- General usage:

```
var = (type*) realloc(ptr, newSize)
```

- ptr is reallocated with the newsize ‘newSize’
  - Helpful when size of previously allocated memory is no more sufficient.

```
// allocate a 10-double array
 // Dynamically re-allocate memory using realloc()
        float* arr = (float*) malloc(10*sizeof(float));
        arr = (float* )realloc(arr, 20 * sizeof(float);
// do stuff with arr
```

realloc()



https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'

Example: https://ide.geeksforgeeks.org/SY5zO9TBDU

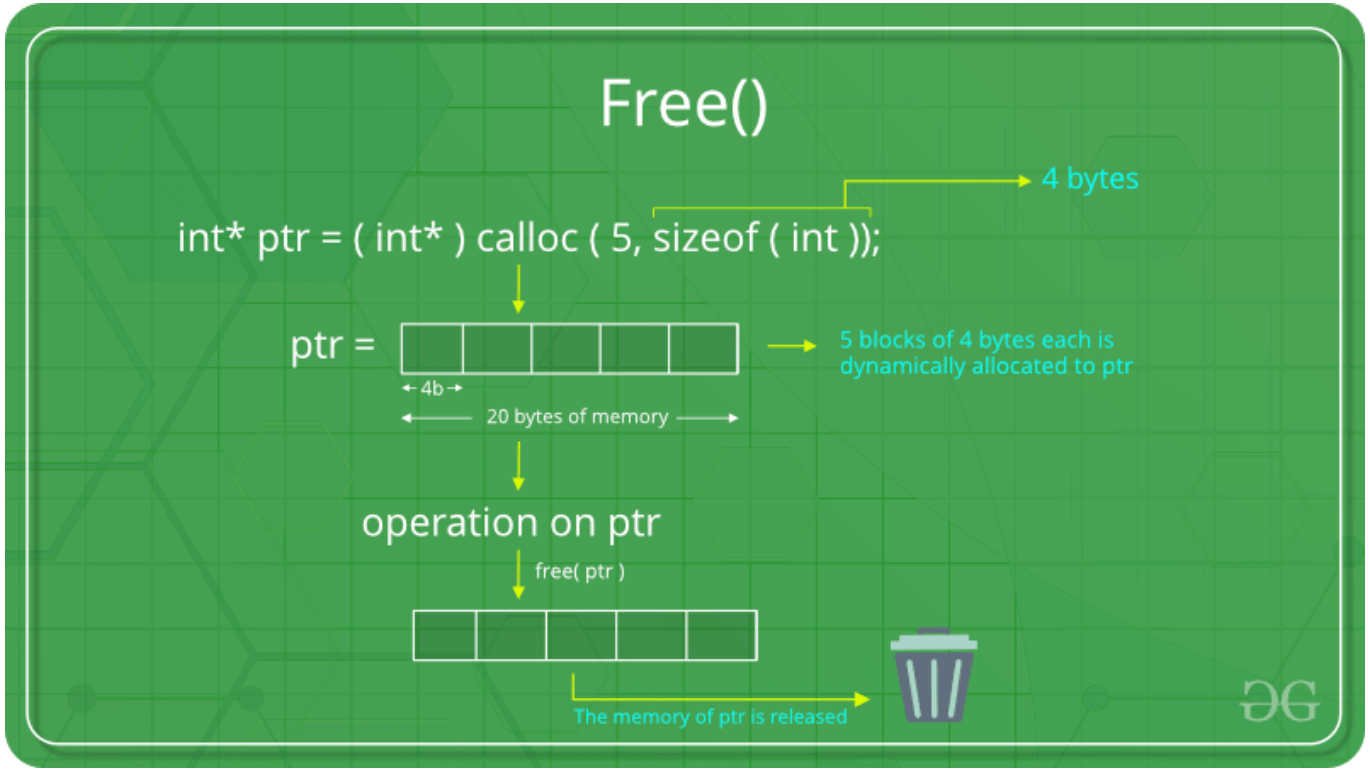
free()

- Usage: `free(pointer);`

- Deallocates the memory pointed-to by the pointer
  - Pointer *must* point to the first byte of heap-allocated memory (*i.e.* something previously returned by **malloc** or **calloc**)
  - Freed memory becomes eligible for future allocation
  - Pointer is unaffected by call to free (Still pointing to same memory locations, which is now free for future allocation)

```
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
  return errcode;
...             // do stuff with arr
free(arr);
arr = NULL;     // OPTIONAL but good practice
```

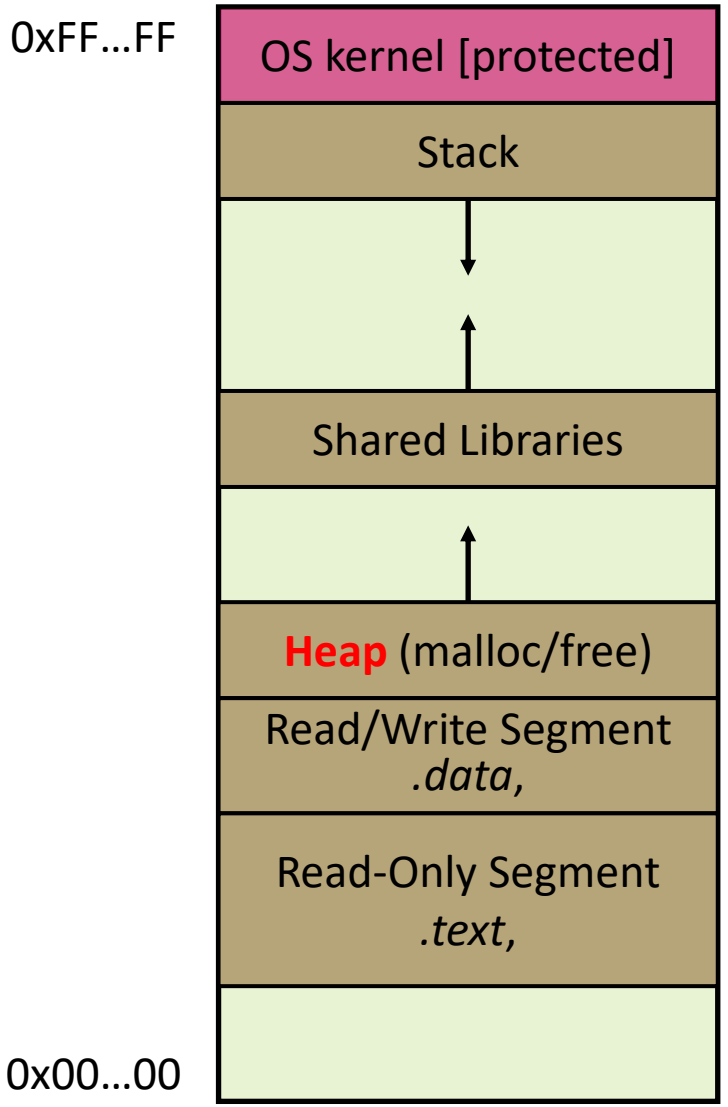
free()



- https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/
- free(ptr);
- ptr keeps pointing to the same memory, but memory is made available for future allocation.

The Heap

- The Heap is a large pool of available memory used to hold dynamically-allocated data
  - `malloc` allocates chunks of data in the Heap;
  - `free` deallocates those chunks
  - `malloc` maintains bookkeeping data in the Heap to track allocated blocks

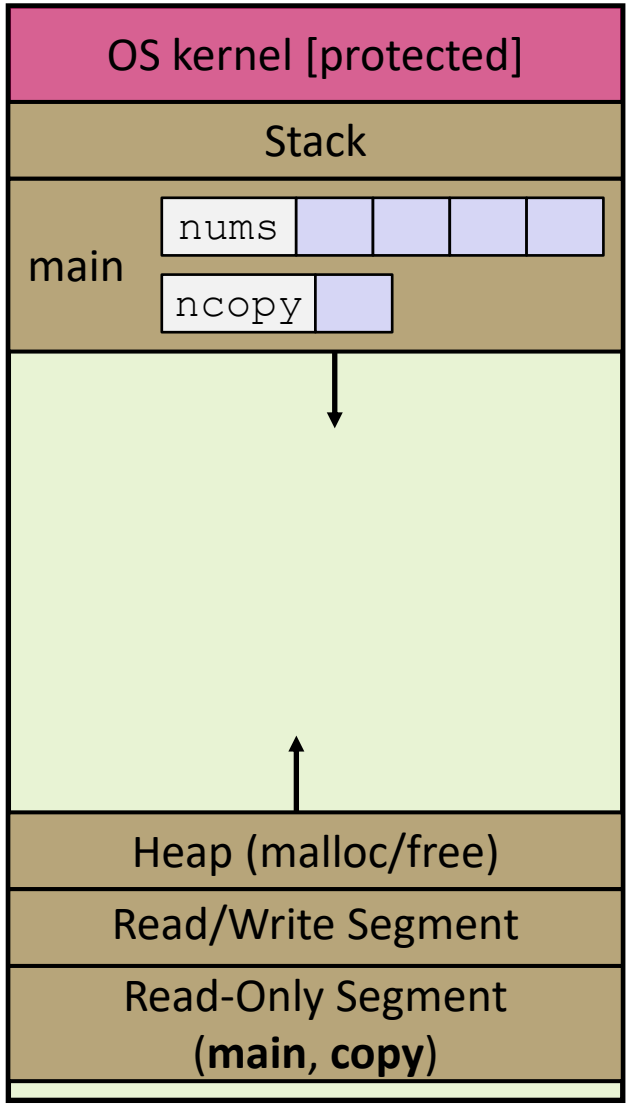


Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

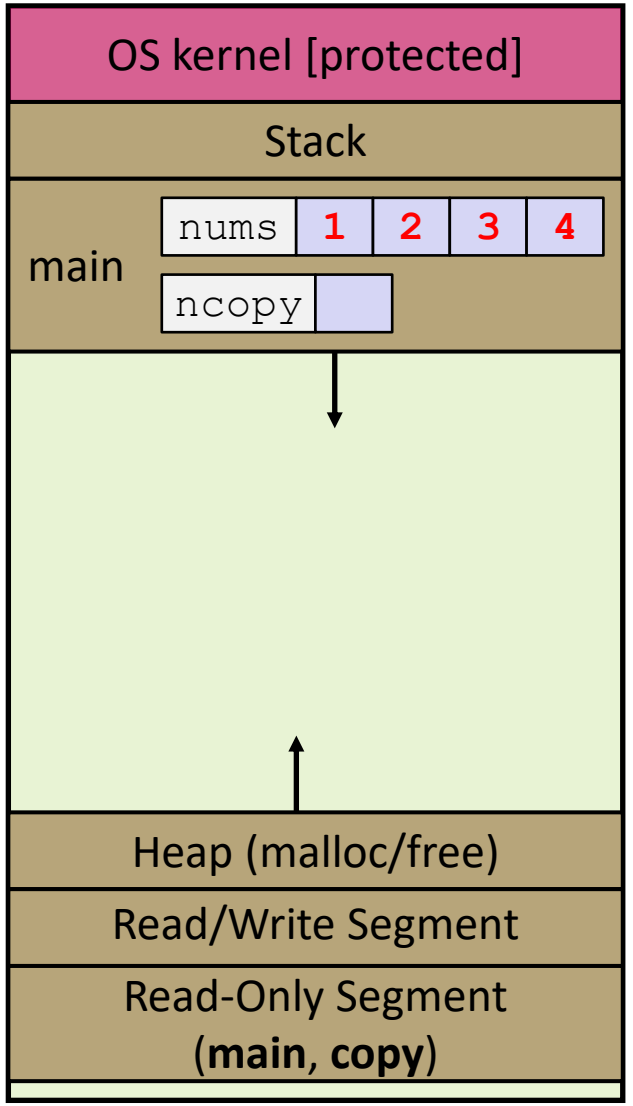
```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*)malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example

arraycopy.c

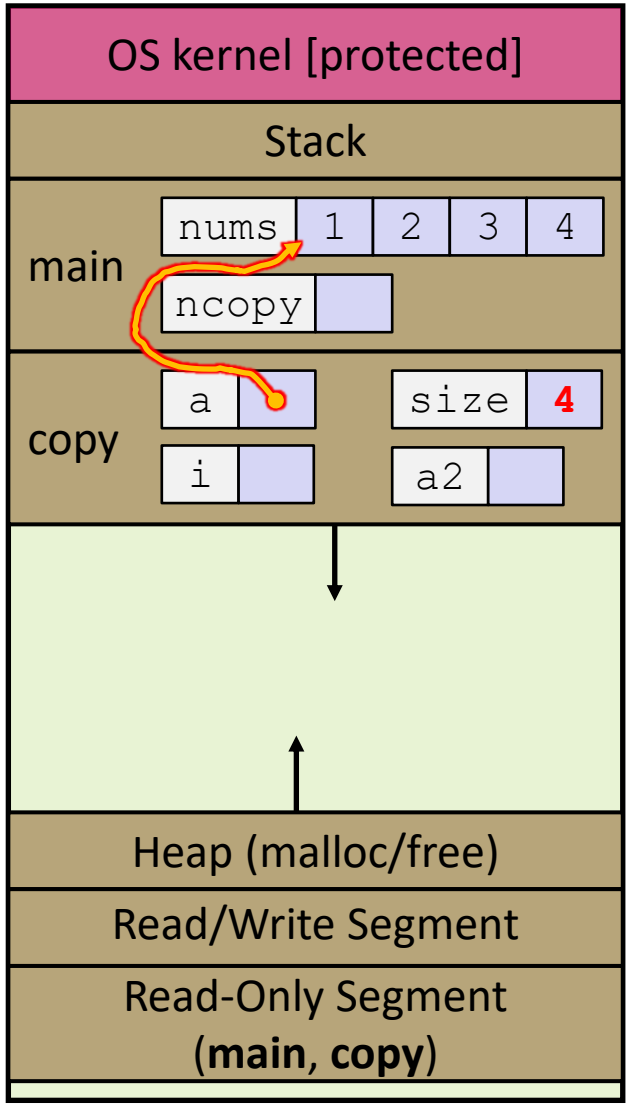
```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example

arraycopy.c

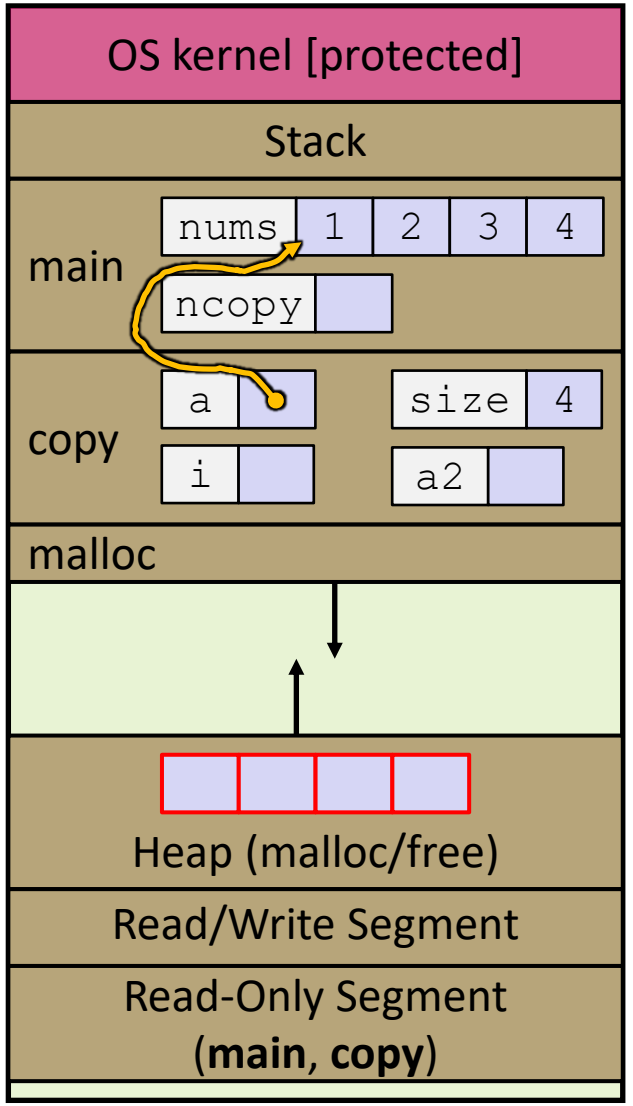
```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example

arraycopy.c

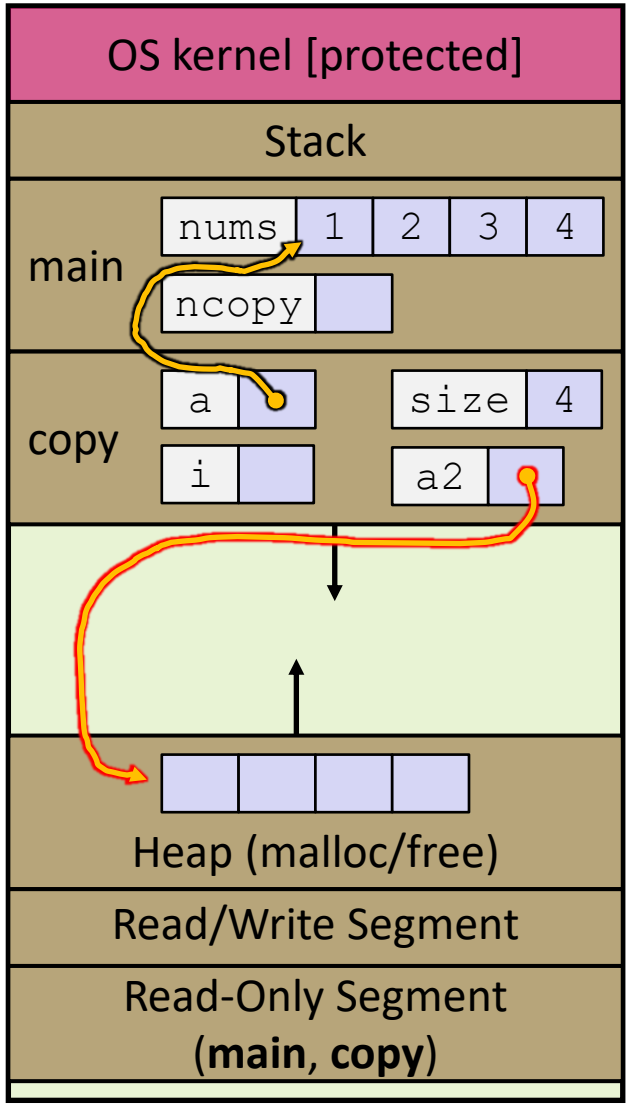
```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example

arraycopy.c

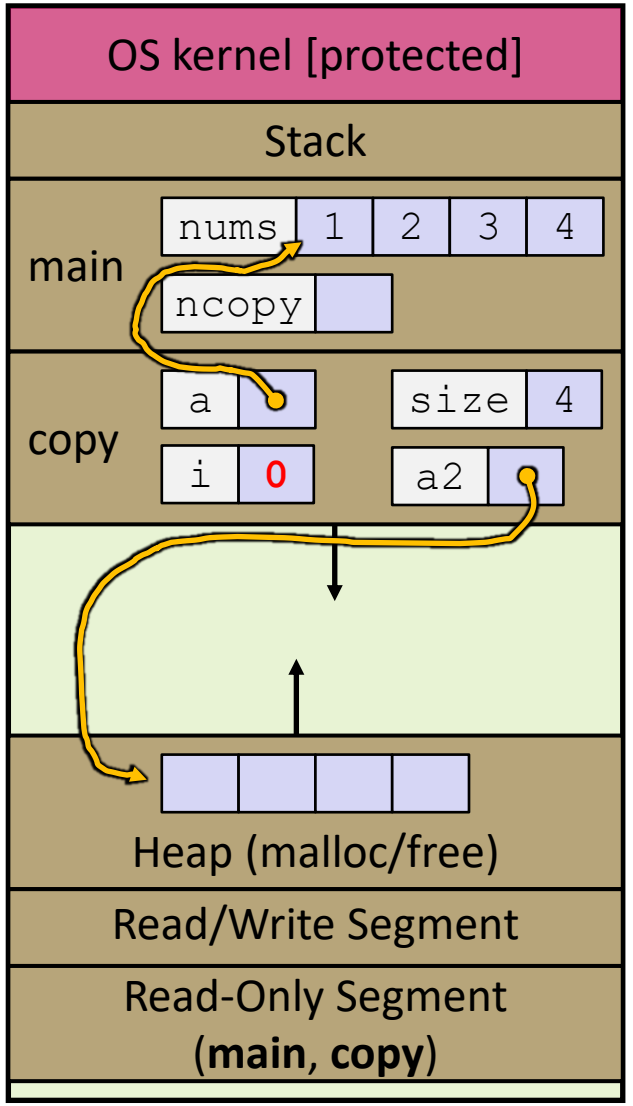
```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example

arraycopy.c

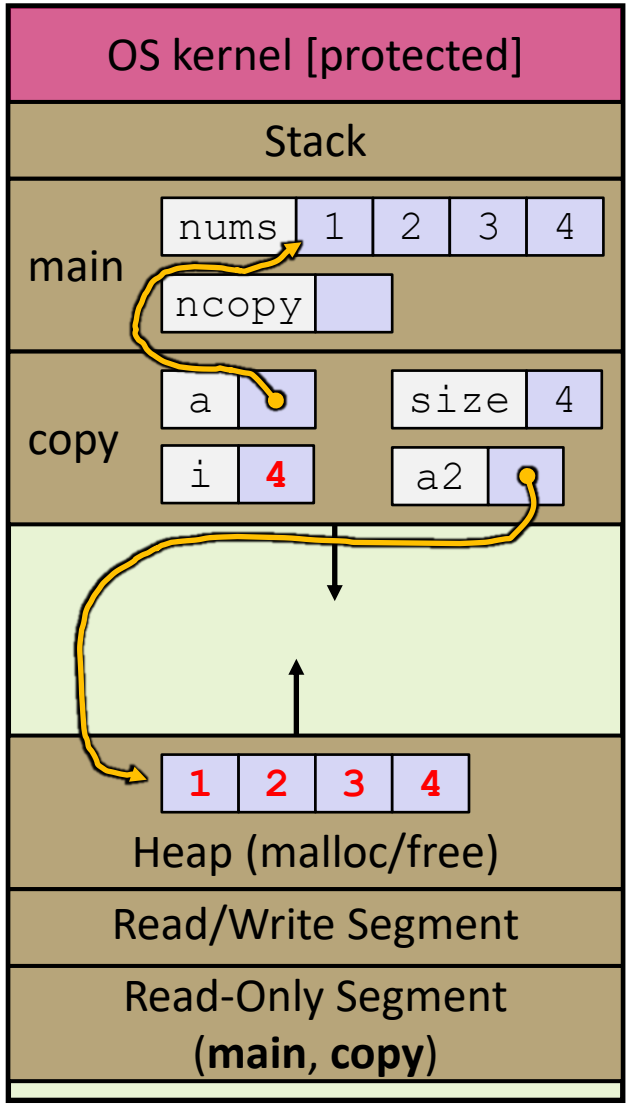
```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example

arraycopy.c

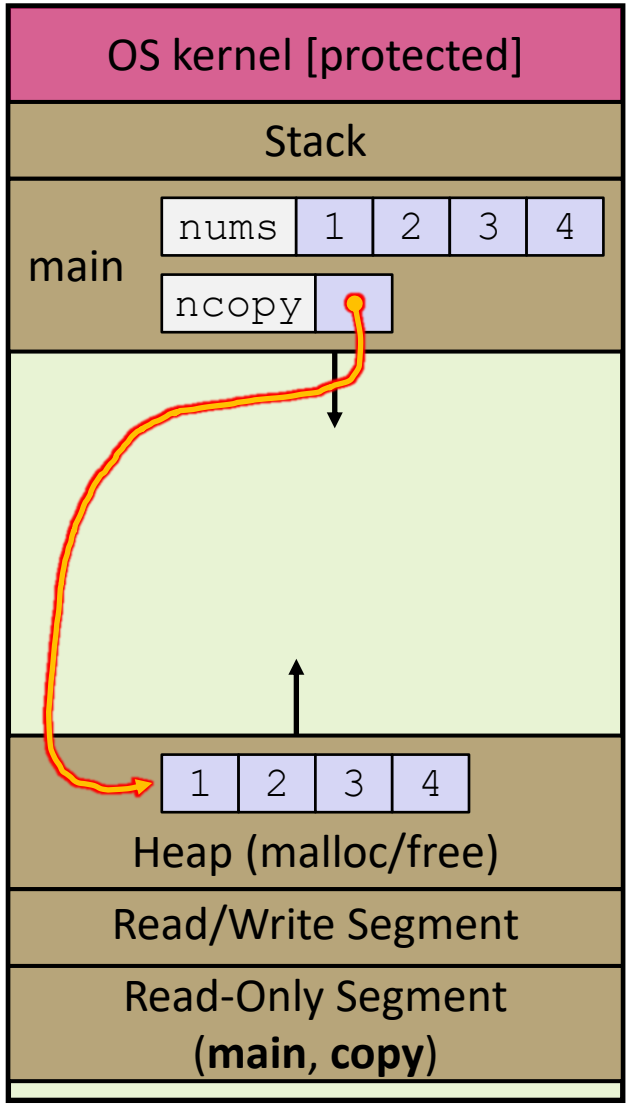
```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example

arraycopy.c

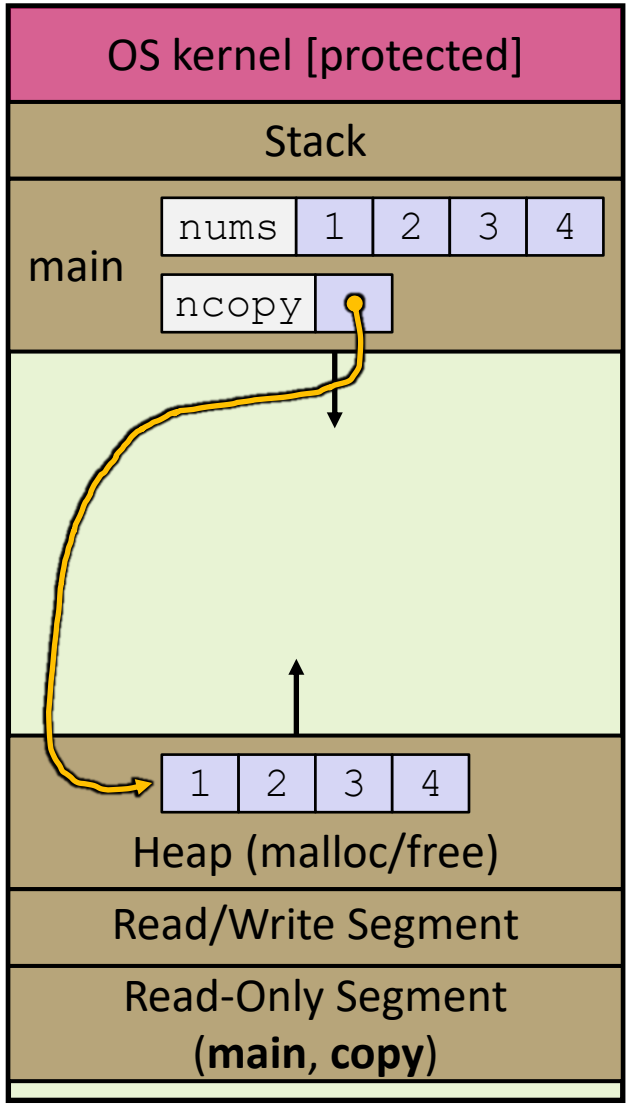
```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example

arraycopy.c

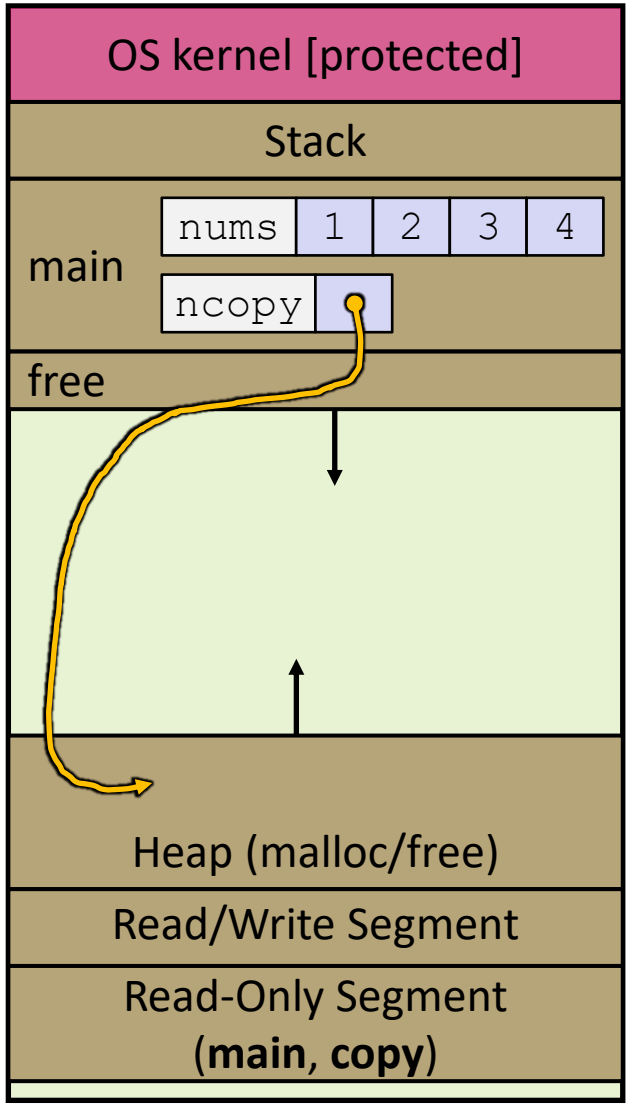
```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example

arraycopy.c

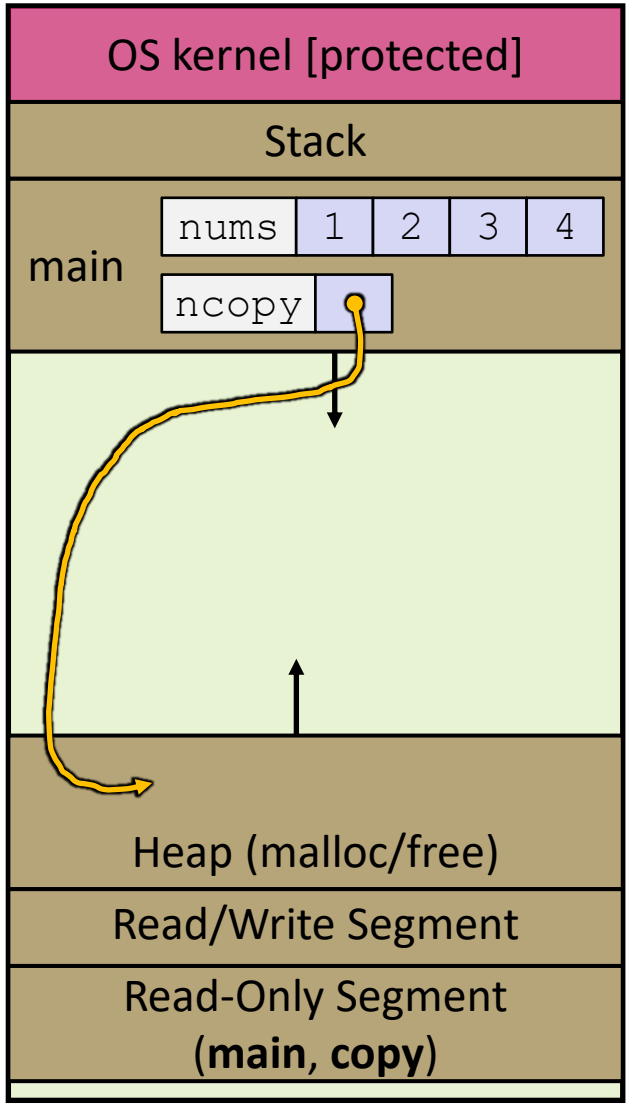
```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = (int*) malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Memory Corruption

- There are all sorts of ways to corrupt memory in C

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

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;   // assign past the end of an array (a[2])
  b[0] += 2;  // assume malloc zeros out memory, using garbage values
  c = b+3;    // pointer pass allocated block(b[3])
  free(&(a[0]));  // free something not malloc'ed (in stack)
  free(b);
  free(b);    // double-free the same block (memory management data
//structure corrupts.
  b[0] = 5;   // use a freed pointer

  return 0;
}
```

Memory Corruption

- There are all sorts of ways to corrupt memory in C

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

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;   // assign past the end of an array (a[2])
  b[0] += 2;  // assume malloc zeros out memory, using garbage values
  c = b+3;    // pointer pass allocated block(b[3])
  free(&(a[0]));  // free something not malloc'ed (in stack)
  free(b);
  free(b);    // double-free the same block (memory management data
//structure corrupts.
  b[0] = 5;   // use a freed pointer

  return 0;
}
```

Memory Leak

- A memory leak occurs when code fails to deallocate dynamically-allocated memory that is no longer used
  - *e.g.* forget to `free` malloc-ed block, lose/change pointer to malloc-ed block

- What happens: program's VM footprint will keep growing
  - This might be OK for *short-lived* program, since all memory is deallocated when program ends
  - Usually has bad repercussions for *long-lived* programs
    - Might slow down over time (*e.g.* lead to VM thrashing)
    - Might exhaust all available memory and crash
    - Other programs might get starved of memory

Memory leak example

```c
/* Function with memory leak */
#include <stdlib.h>

void f()
{
    int *ptr = (int *) malloc(sizeof(int))

    /* Do some work */

    return; /* Return without freeing ptr*
}
```

Fixing memory leak

```c
/* Function without memory leak */
#include <stdlib.h>;

void f()
{
    int *ptr = (int *) malloc(sizeof(int))

    /* Do some work */

    free(ptr);
    return;
}
```

Lecture Outline

- Heap-allocated Memory
  - `malloc()`, `calloc()`, `realloc()` and `free()`
  - Memory leaks
- **Structs, Unions, Enumerations and typedef**

Structured Data

- A `struct` is a C datatype that contains a set of fields
  - Similar to a Java class, but with no methods or constructors
  - Useful for defining new structured types of data
  - Behave similarly to primitive variables

- Generic declaration:

```
struct tagname {
  type1 name1;
  ...
  typeN nameN;
};
```

```
// the following defines a new
// structured datatype called
// a "struct Point"
struct Point {
  float x, y;
};

// declare and initialize a
// struct Point variable
struct Point origin = {0.0,0.0};
```

How to declare structure variables

```
// A variable declaration with structure declaration.
struct Point
{
    int x, y;
} p1;  // The variable p1 is declared with 'Point'
```

- With structure declaration

```
// A variable declaration like basic data types
struct Point
{
    int x, y;
};

int main()
{
    struct Point p1;  // The variable p1 is declared like a normal variable
}
```

- As separate declaration

How to not initialize structure members?

- Structure members cannot be initialized within declaration
  - There is no memory allocated yet when a datatype is declared
  - Memory is allocated when variables are created

```
struct Point
{
    int x = 0;  // COMPILER ERROR
    int y = 0;  // COMPILER ERROR
};
```

Initializing structure members?

```
3  struct Point
4  {
5    int x, y;
6  }p1 = {0,1}, p2 = {2,3};
7
8  int main()
9  {
10     printf ("p1.x = %d, p1.y = %d\n", p1.x, p1.y);
11     printf ("p2.x = %d, p2.y = %d", p2.x, p2.y);
12     return 0;
13 }
```

```
p1.x = 0, p1.y = 1
p2.x = 2, p2.y = 3
```

https://ide.geeksforgeeks.org/M4eruAvq89

Designated Initialization

- Designated Initialization allows structure members to be initialized in any order. This feature has been added in C99 standard

```c
#include<stdio.h>

struct Point
{
    int x, y, z;
};

int main()
{
    // Examples of initializtion using designated initializatio
    struct Point p1 = {.y = 0, .z = 1, .x = 2};
    struct Point p2 = {.x = 20};

    printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);
    printf ("x = %d", p2.x);
    return 0;
}
```

```
x = 2, y = 0, z = 1
x = 20
```

https://ide.geeksforgeeks.org/71ADKY1vrq

Array of structures

- we can create an array of structures.

```c
#include<stdio.h>

struct Point
{
    int x, y;
};

int main()
{
    // Create an array of structures
    struct Point arr[10];

    // Access array members
    arr[0].x = 10;
    arr[0].y = 20;

    printf("%d %d", arr[0].x, arr[0].y);
    return 0;
}
```

```
10 20
```

https://ide.geeksforgeeks.org/3lbz1OgkBf

Structure Pointers

- If we have a pointer to structure, members are accessed using arrow (->) operator.

```c
#include<stdio.h>

struct Point
{
    int x, y;
};

int main()
{
    struct Point p1 = {1, 2};

    // p2 is a pointer to structure p1
    struct Point *p2 = &p1;

    // Accessing structure members using structure pointer
    printf("%d %d", p2->x, p2->y);
    return 0;
}
```

```
1 2
```

https://ide.geeksforgeeks.org/8cCdU7Ypi8

Structure Pointers

- Use “ . ” to refer to a field in a struct
- Use “ –> ” to refer to a field from a struct pointer
  - Dereferences pointer first, then accesses field

```
struct Point {
  float x, y;
};

int main(int argc, char** argv) {
  struct Point p1 = {0.0, 0.0};  // p1 is stack allocated
  struct Point* p1_ptr = &p1;

  p1.x = 1.0;
  p1_ptr->y = 2.0;  // equivalent to (*p1_ptr).y = 2.0;
  return 0;
}
```

https://onlinegdb.com/HkktYZtPS

Copy by Assignment

- You can assign the value of a struct from a struct of the same type – *this copies the entire contents!*

```
struct Point {
  float x, y;
};

int main(int argc, char** argv) {
  struct Point p1 = {0.0, 2.0};
  struct Point p2 = {4.0, 6.0};

  printf("p1: {%f,%f}  p2: {%f,%f}\n", p1.x, p1.y, p2.x, p2.y);
  p2 = p1;
  printf("p1: {%f,%f}  p2: {%f,%f}\n", p1.x, p1.y, p2.x, p2.y);
  return 0;
}
```

Nested Structures

- Structure can be used inside another structure as a member.

```c
#include <stdio.h>
struct date
{
    int day;
    int month;
    int year;
};

struct Employee
{
    char ename[20];
    int ssn;
    float salary;
    struct date dob;
};
```

```c
int main()
{
  struct Employee Ahmet;
  Ahmet.dob.day = 1;
  Ahmet.dob.month = 10;
  Ahmet.dob.year = 1000;

  printf("%d / %d / %d",
         Ahmet.dob.day, Ahmet.dob.month, Ahmet.dob.year );
}
```

```
1 / 10 / 1000
```

https://ide.geeksforgeeks.org/z5Z5q4SnxK

Structures can also be declared as nested

```c
#include <stdio.h>

struct Employee
{
    char ename[20];
    double salary;
    struct date
        {
            int date;
            int month;
            int year;
        }dob;
}emp = {"Ahmet",1000,{1,1,1000}};

int main(int argc, char *argv[])
{
    printf("\nEmployee Name    : %s",emp.ename);
    printf("\nEmployee Salary  : %f",emp.salary);
    printf("\nEmployee DOB     : %d/%d/%d", \
        emp.dob.date,emp.dob.month,emp.dob.year);
    return 0;
}
```

```
Employee Name    : Ahmet
Employee Salary  : 1000.000000
Employee DOB     : 1/1/1000
```

https://ide.geeksforgeeks.org/MgUB1l10ao

What can we use as members of structures?

- Any primitive data type (char, unsigned, int, float, etc.)
- Other structures
- Arrays
- Pointers
- Unions

Pointers as struct members

1

```
void main(){
    struct msg{
        char *p1;
        char *p2;
    } myptrs;

    myptrs.p1 = "Teach Yourself C";
    myptrs.p2 = "SAMS";
}
```

2

```
void main(){
    struct msg{
        char *p1;
        char *p2;
    } myptrs;

    myptrs.p1 = malloc(20*sizeof(char));
    myptrs.p1 = "Teach Yourself C";
    myptrs.p1 = malloc(10*sizeof(char));
    myptrs.p2 = "SAMS";
}
```

3

```
void main(){
    struct msg{
        char p1[20];
        char p2[10];
    } myptrs = {"Teach Yourself C", "SAMS"};
}
```

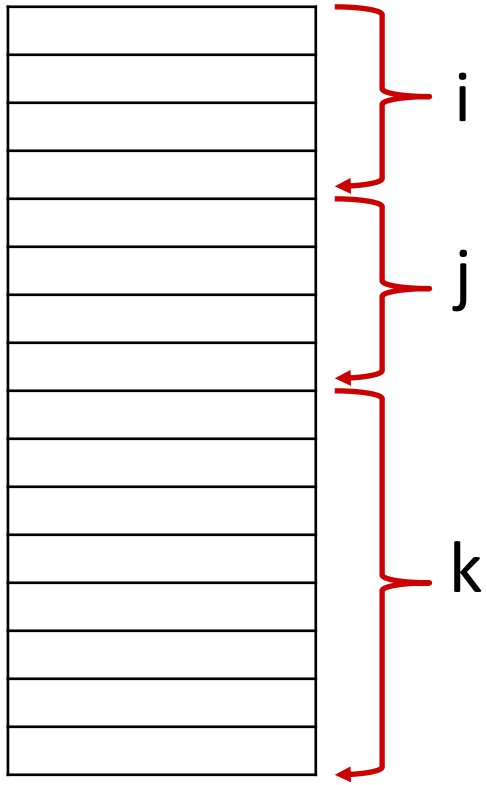
Unions

- Unions look like structures however compiler allocates only enough space for the largest of the members.

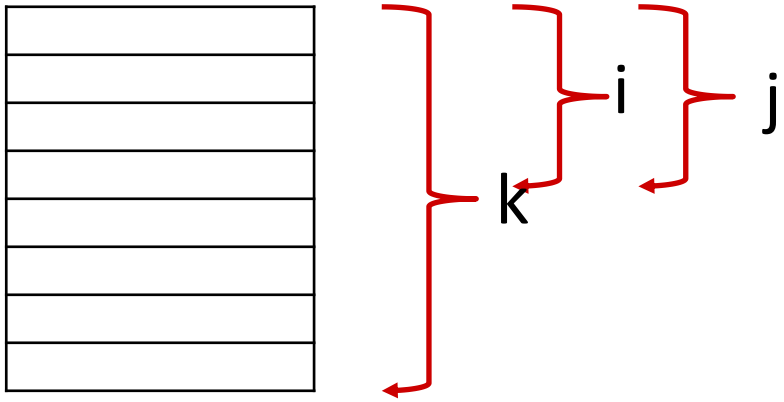
- Assigning a new value to one member alters the values of other members as well

Struct vs union

```
struct {
        int i;
        int j;
        double k;
} u;
```




```
union {
        int i;
        int j;
        double k;
} u;
```



Use of Unions

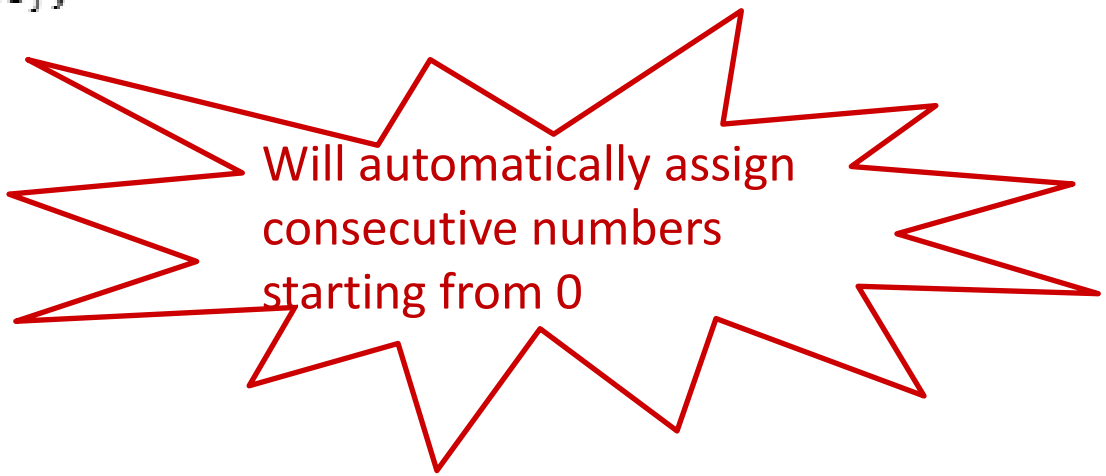
- Maybe used to save space for some data structures
- Creating mixed data structures, e.g. Creating an array, where each element is a different type.
  - Still will not know for each element which variable is initialized
- Solution: put union in a struct and create a “tag field”

```
struct data{
        int kind; //tag field
        union {
                int i;
                double d;
        } u;

} number;
```

Enumeration

- Allows you to associate integral constants with names

```c
// Another example program to demonstrate working
// of enum in C
#include<stdio.h>

enum year{Jan, Feb, Mar, Apr, May, Jun, Jul,
        Aug, Sep, Oct, Nov, Dec};

int main()
{
int i;
for (i=Jan; i<=Dec; i++)
    printf("%d ", i);

return 0;
}
```




```
0 1 2 3 4 5 6 7 8 9 10 11
```

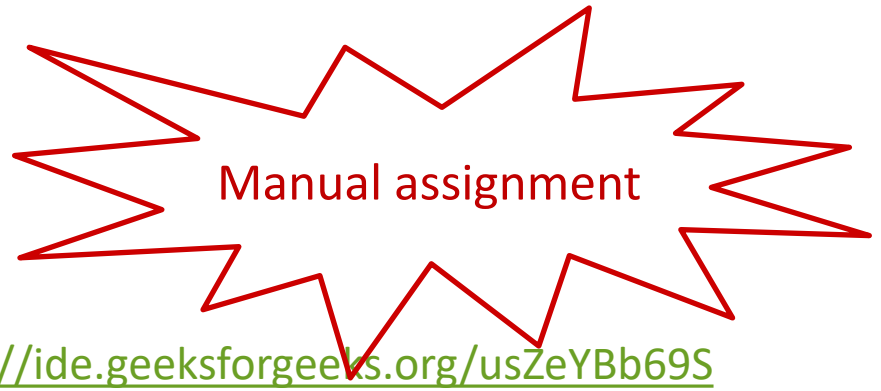
https://ide.geeksforgeeks.org/L5L1TZke0h

Enumeration

- Two enum names can have same value.

```c
#include <stdio.h>
enum State {Working = 1, Failed = 0, Freezed = 0};

int main()
{
printf("%d, %d, %d", Working, Failed, Freezed);
return 0;
}

```

```
1, 0, 0
```



https://ide.geeksforgeeks.org/usZeYBb69S

Enumerations as “Tag Fields”

```c
#include <stdio.h>
int main()
{
    struct data{
        enum {INT_KIND, DOUBLE_KIND} kind; //tag field
        union {
            int i;
            double d;
        } u;

    } number;


    number.kind =DOUBLE_KIND;
    number.u.d = 2.3;

   if(number.kind == DOUBLE_KIND)
     printf("Number is: %lf", number.u.d);
}
```

```
Number is: 2.300000
```

https://ide.geeksforgeeks.org/MMaqUwXYEP

typedef

- Generic format: `typedef type name;`
- Allows you to define new data type *names/synonyms*
  - Both `type` and `name` are usable and refer to the same type
  - Be careful with pointers – `*` before `name` is part of `type`!

```
// make "superlong" a synonym for "unsigned long long"
typedef unsigned long long superlong;

// make "str" a synonym for "char*"
typedef char* str;

// make "Point" a synonym for "struct point_st { ... }"
// make "PointPtr" a synonym for "struct point_st*"
typedef struct point_st {
  superlong x;
  superlong y;
} Point, *PointPtr;  // similar syntax to "int n, *p;"

Point origin = {0, 0};
```

Typedef extended example

https://onlinegdb.com/B1LkDkyNI

When you should define your own types

- you are working on a system that
  - will have a lot of code ...
  - last a long time ...
  - need to be ported ...
  - have multiple revisions ...
  - have a maintenance life time ...
  - rely on standards ...
- System-specific types afford you flexibility to alter the foundational data structures and types quickly and apply to large code bases.

Dynamically-allocated Structs

- You can **`malloc`** and **`free`** structs, just like other data type
  - `sizeof` is particularly helpful here

```
typedef struct point_st {
  superlong x;
  superlong y;
} Point, *PointPtr;


PointPtr AllocComplex(superlong x, superlong y) {
  PointPtr retval = (PointPtr) malloc(sizeof(Point));
  if (retval != NULL) {
    retval->x = x;
    retval->y = y;
  }
  return retval;
}
```

Structs as Arguments

- Structs are passed by value, like everything else in C
  - Entire struct is copied – where?
  - To manipulate a struct argument, pass a pointer instead

```c
typedef struct point_st {
  int x, y;
} Point, *PointPtr;

void DoubleXBroken(Point p)    {  p.x *= 2; }

void DoubleXWorks(PointPtr p) { p->x *= 2; }

int main(int argc, char** argv) {
  Point a = {1,1};
  DoubleXBroken(a);
  printf("(%d,%d)\n", a.x, a.y);    // prints: ( 1 , 1  )
  DoubleXWorks(&a);
  printf("(%d,%d)\n", a.x, a.y);    // prints: ( 2  , 1 )
  return 0;
}
```

https://onlinegdb.com/S1FqAWFwB

Pass Copy of Struct or Pointer?

- Value passed: passing a pointer is cheaper and takes less space unless struct is small
- Field access: indirect accesses through pointers are a bit more expensive and can be harder for compiler to optimize
- For small structs passing a copy of the struct can be faster and often preferred if function only reads data; for large structs use pointers

Structs vs Java Classes

- No Data Hiding: C Structures do not permit data hiding. Structure members can be accessed by any function, anywhere in the scope of the Structure.
- Functions inside Structure: C structures do not permit functions inside Structure
- Static Members: C Structures cannot have static members inside their body
- Access Modifiers: C Programming language do not support access modifiers. So they cannot be used in C Structures.
- Construction creation in Structure: Structures in C cannot have constructor inside Structures

Extra Exercise

- Write a program that defines:
  - A new structured type Point
    - Represent it with `floats` for the x and y coordinates
  - A new structured type Rectangle
    - Assume its sides are parallel to the x-axis and y-axis
    - Represent it with the bottom-left and top-right Points
  - A function that computes and returns the area of a Rectangle
  - A function that tests whether a Point is inside of a Rectangle

Detecting memory leaks, other memory issues

- We will use Valgrind at the lab. Don't forget to use –g option when building your program, otherwise Valgrind will not be able to provide useful information.
- https://valgrind.org/

```
HEAP SUMMARY:
    in use at exit: 40 bytes in 1 blocks
  total heap usage: 1 allocs, 0 frees, 40 bytes allocated

40 bytes in 1 blocks are definitely lost in loss record 1 of 1
   at 0x4C29EA3: malloc (vg_replace_malloc.c:309)
   by 0x40073E: main (dynamic.c:26)

LEAK SUMMARY:
   definitely lost: 40 bytes in 1 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```