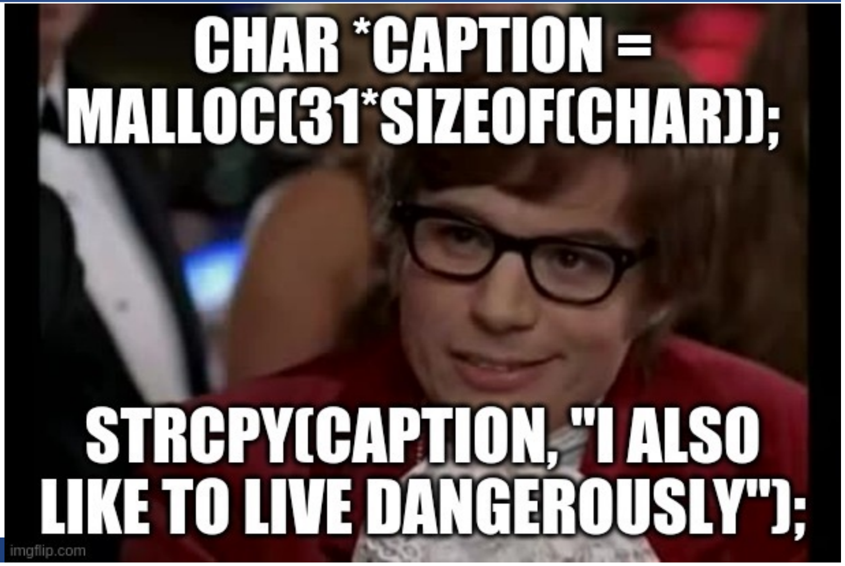
CMPSC 311 - Introduction to Systems Programming

Memory Management

Professors:

Suman Saha

(Slides are mostly by *Professor Patrick McDaniel* and *Professor Abutalib Aghayev*)



Box and arrow diagrams

boxarrow.c

```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];

  printf("&x: %p;  x: %d\n", &x, x);
  printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
  printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
  printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);
  return 0;
}
```

address	name	value
&x	x	value
&arr[0]	arr[0]	value
&arr[1]	arr[1]	value
&arr[2]	arr[2]	value
&p	p	value

Box and arrow diagrams

boxarrow.c

```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];

  printf("&x: %p;  x: %d\n", &x, x);
  printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
  printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
  printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);
  return 0;
}
```

address	name	value
&x	x	1
&arr[0]	arr[0]	2
&arr[1]	arr[1]	3
&arr[2]	arr[2]	4
&p	p	&arr[1]

Box and arrow diagrams

PennState

boxarrow.c

```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];

  printf("&x: %p;  x: %d\n", &x, x);
  printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
  printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
  printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);
  return 0;
}
```

address	name	value
0xbfff2dc	x	1
0xbfff2d0	arr[0]	2
0xbfff2d4	arr[1]	3
0xbfff2d8	arr[2]	4
0xbfff2cc	p	0xbfff2d4

Box and arrow diagrams

boxarrow.c

```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];

  printf("&x: %p;  x: %d\n", &x, x);
  printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
  printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
  printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);
  return 0;
}
```

address	name	value

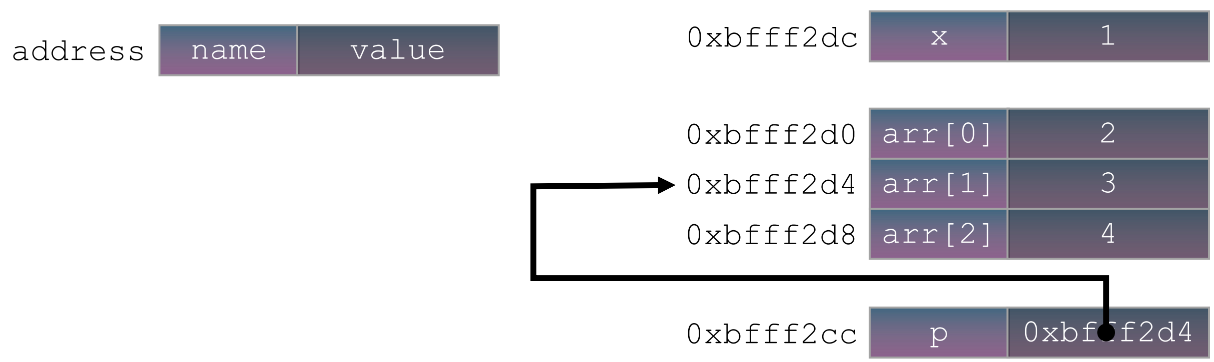
0xbfff2dc	x	1
0xbfff2d8	arr[2]	4
0xbfff2d4	arr[1]	3
0xbfff2d0	arr[0]	2
0xbfff2cc	p	0xbfff2d4

main( )'s stack frame

Box and arrow diagrams

boxarrow.c

```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];

  printf("&x: %p;  x: %d\n", &x, x);
  printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
  printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
  printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);
  return 0;
}
```



Double pointers

- Question: “what’s the difference between a (char *) and a (char **)?

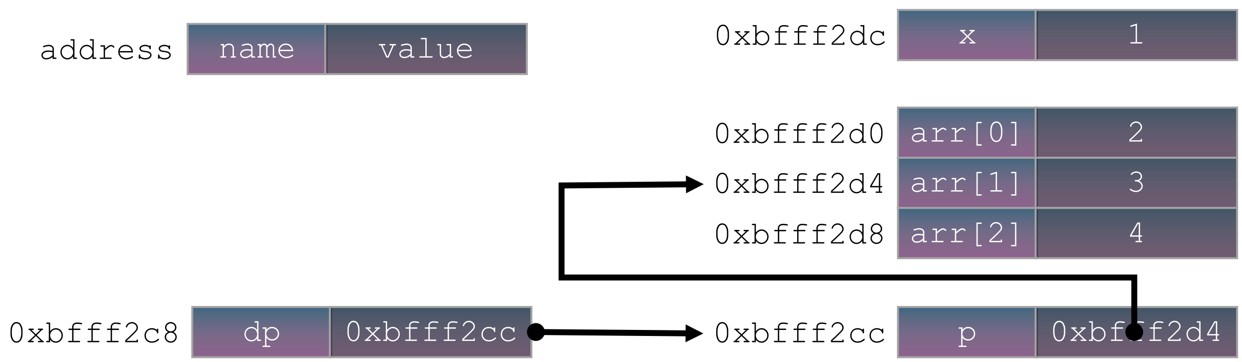
```
int main(int argc, char **argv) {
  char hi[6] = {'h', 'e', 'l',
                'l', 'o', '\0'};
  char *p, **dp;

  p = &hi[0];
  dp = &p;

  printf("%c %c\n", *p, **dp);
  printf("%p %p %p\n", p, *dp, hi);
  p += 1;
  printf("%c %c\n", *p, **dp);
  printf("%p %p %p\n", p, *dp, hi);
  *dp += 2;
  printf("%c %c\n", *p, **dp);
  printf("%p %p %p\n", p, *dp, hi);
  return 0;
}
```

exercise0.c

Box and arrow diagrams

boxarrow2.c

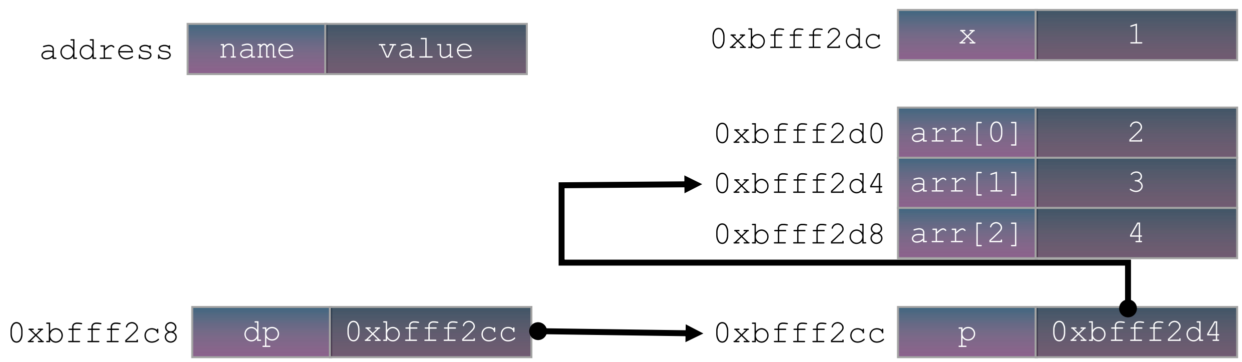
```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];
  int **dp = &p;

  **dp += 1;
  p += 1;
  **dp += 1;
  return 0;
}
```



Box and arrow diagrams

boxarrow2.c

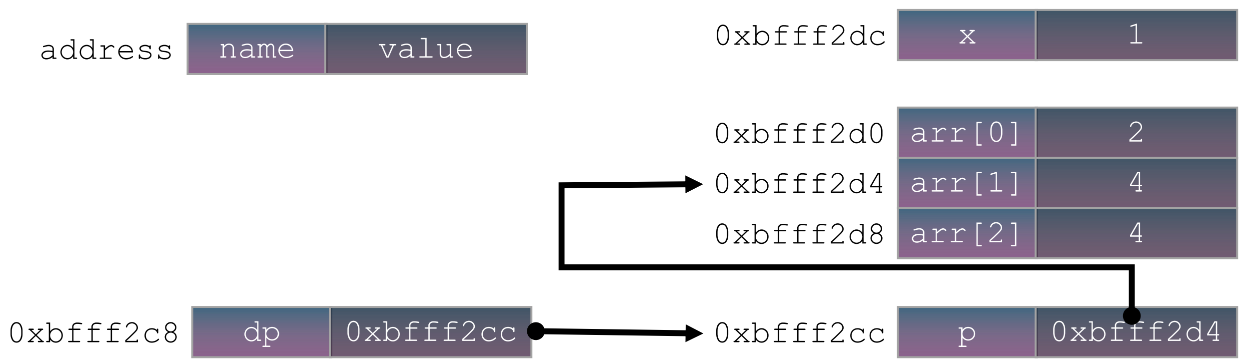
```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];
  int **dp = &p;

  **dp += 1;
  p += 1;
  **dp += 1;
  return 0;
}
```



Box and arrow diagrams

boxarrow2.c

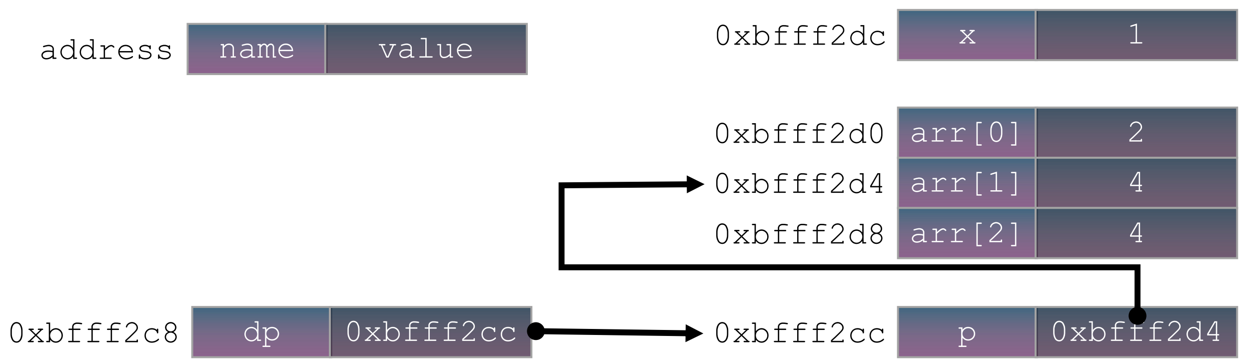
```
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];
  int **dp = &p;

  **dp += 1;
  p += 1;
  **dp += 1;
  return 0;
}
```



Box and arrow diagrams

boxarrow2.c

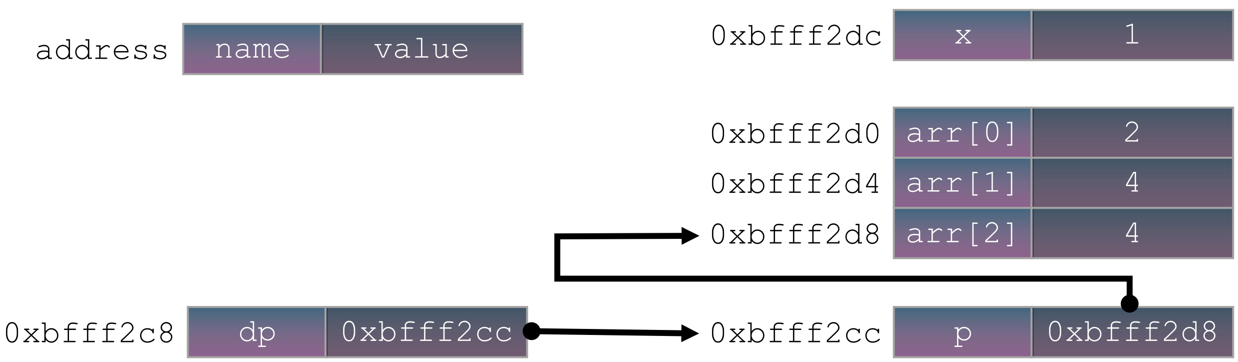
```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];
  int **dp = &p;

  **dp += 1;
  p += 1;
  **dp += 1;
  return 0;
}
```



Box and arrow diagrams

boxarrow2.c

```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];
  int **dp = &p;

  **dp += 1;
  p += 1;
  **dp += 1;
  return 0;
}
```

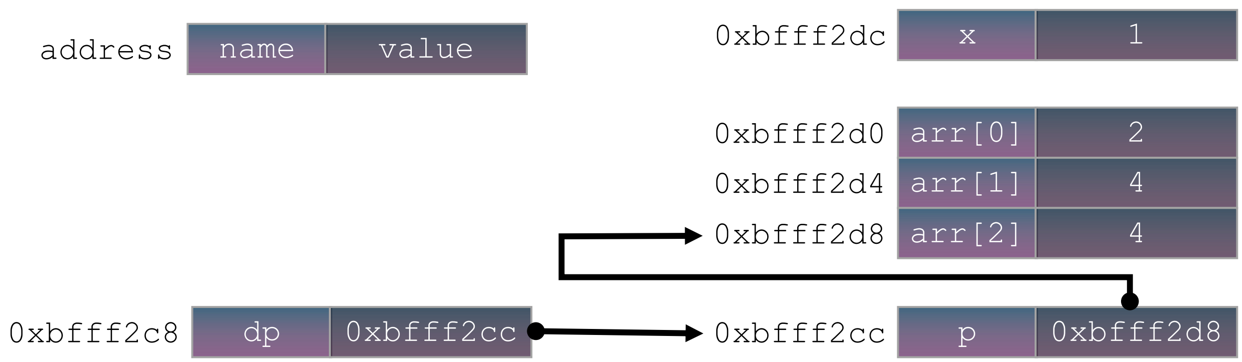


Box and arrow diagrams

PennState

boxarrow2.c

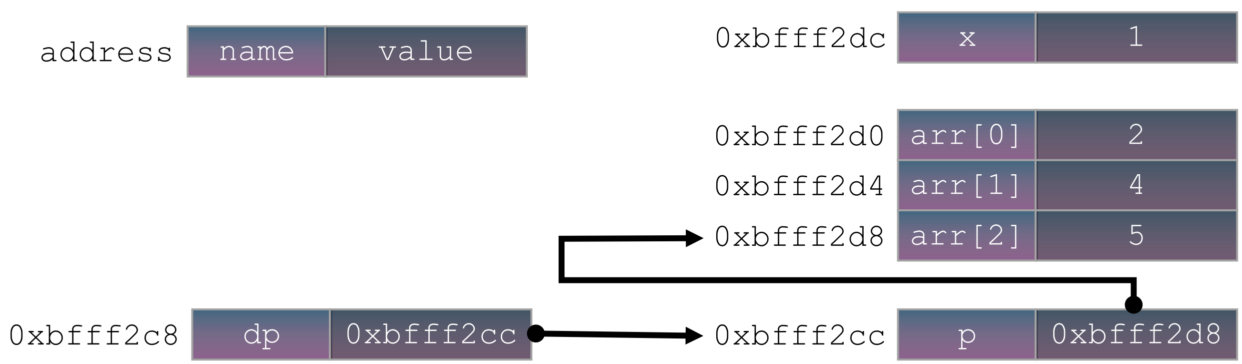
```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];
  int **dp = &p;

  **dp += 1;
  p += 1;
  **dp += 1;
  return 0;
}
```



Box and arrow diagrams

boxarrow2.c

```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];
  int **dp = &p;

  **dp += 1;
  p += 1;
  **dp += 1;
  return 0;
}
```



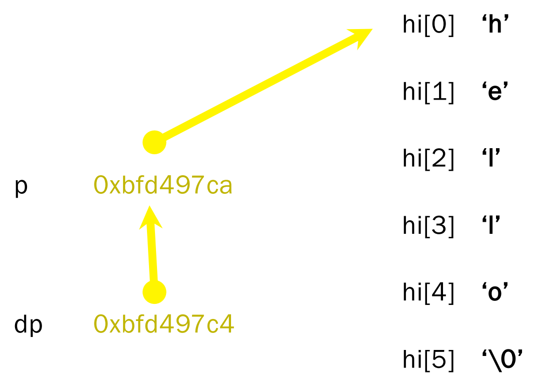
Double pointers

- Question: “what’s the difference between a (char *) and a (char **)?

```
int main(int argc, char **argv) {
  char hi[6] = {'h', 'e', 'l',
                'l', 'o', '\0'};
  char *p, **dp;

  p = &hi[0];
  dp = &p;

  printf("%c %c\n", *p, **dp);
  printf("%p %p %p\n", p, *dp, hi);
  p += 1;
  printf("%c %c\n", *p, **dp);
  printf("%p %p %p\n", p, *dp, hi);
  *dp += 2;
  printf("%c %c\n", *p, **dp);
  printf("%p %p %p\n", p, *dp, hi);
  return 0;
}
```

exercise0.c

Exercise: draw / update the box-and-arrow diagram for this program as it executes

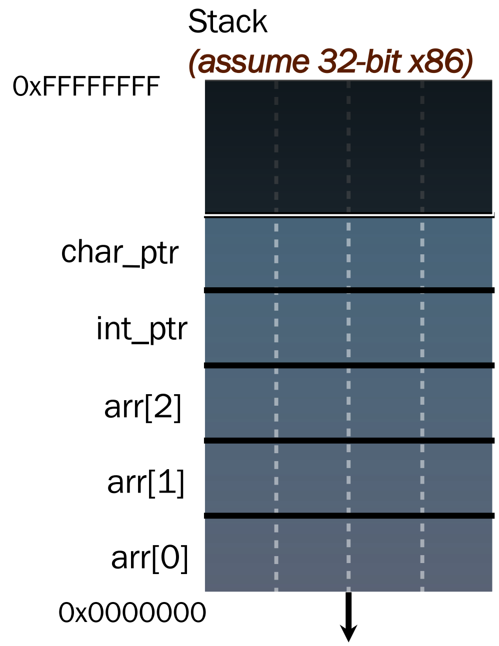


Pointer Arithmatic

- Pointers are typed
  - int *int_ptr;   vs.   char *char_ptr;
  - pointer arithmetic obeys those types
  - i.e., when you add 1 to a pointer, you add sizeof() that type

Sample Pointer Arithmetic

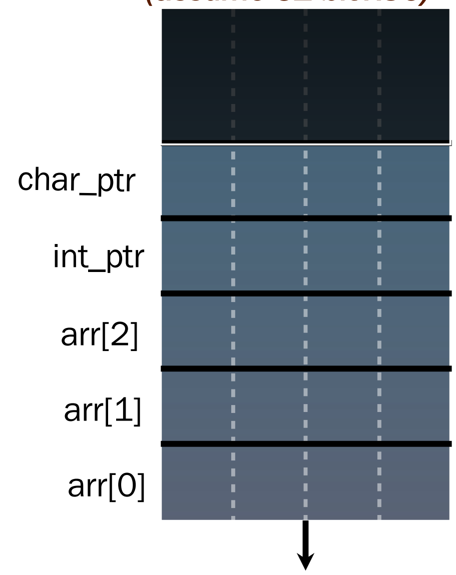
```c
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```



Sample Pointer Arithmetic

```
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```

Stack
(assume 32-bit x86)



(x86 is little endian)

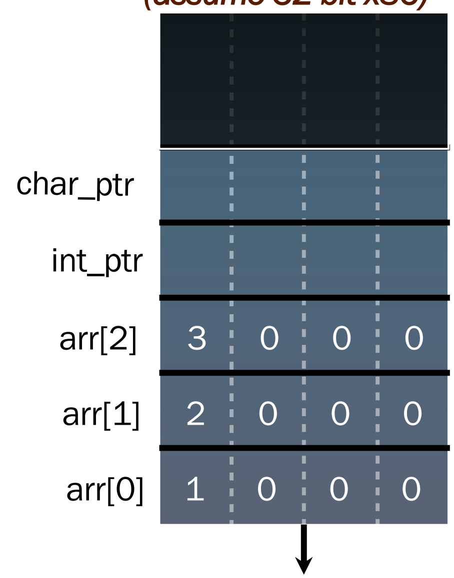
pointerarithmetic.c

Sample Pointer Arithmetic

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```

Stack

(assume 32-bit x86)



pointerarithmetic.c

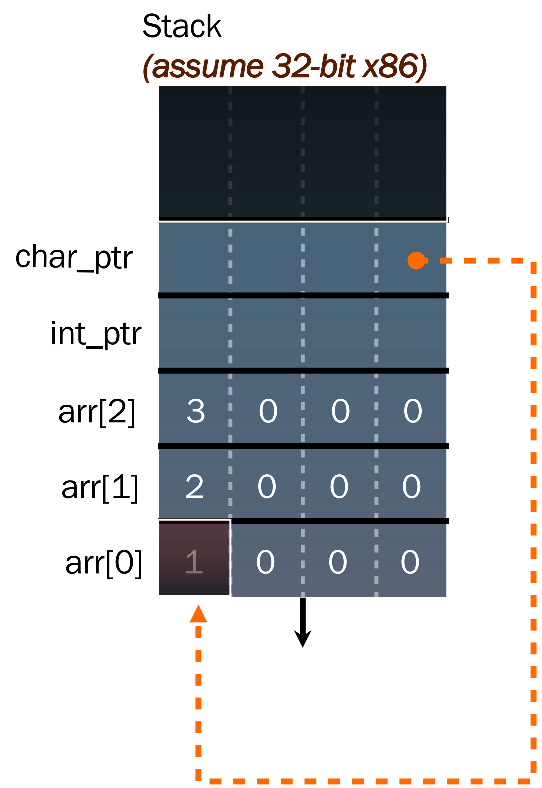
Sample Pointer Arithmetic

```
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```

pointerarithmetic.c

Sample Pointer Arithmetic

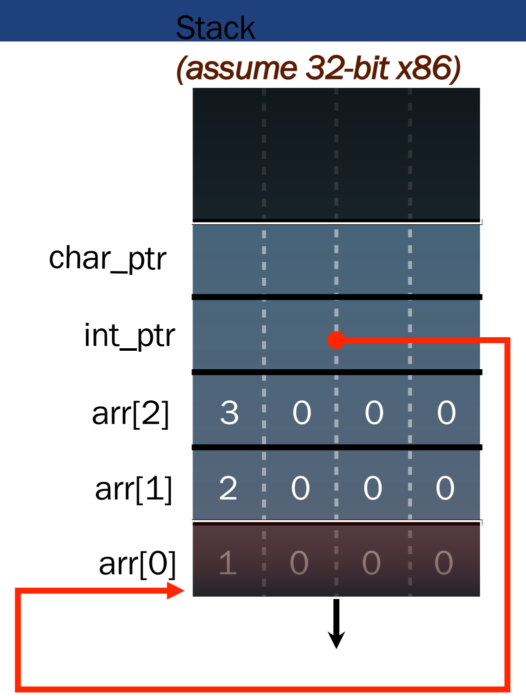
```c
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```



pointerarithmetic.c

Sample Pointer Arithmetic

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```

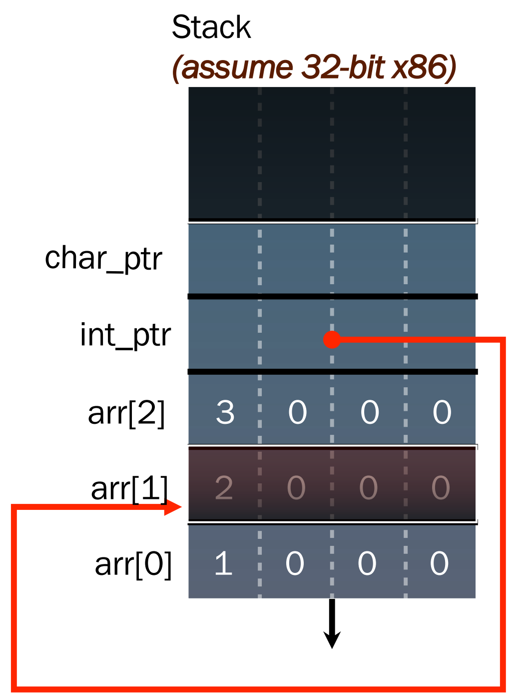



```
int_ptr: 0xbffff2ac;  *int_ptr: 1
```

pointerarithmetic.c

Sample Pointer Arithmetic

```
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```




```
int_ptr: 0xbffff2ac;  *int_ptr: 1
```

pointerarithmetic.c

Sample Pointer Arithmetic

```
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```


```
int_ptr: 0xbffff2ac;  *int_ptr: 1
int_ptr: 0xbffff2b0;  *int_ptr: 2
```

pointerarithmetic.c

Sample Pointer Arithmetic

PennState

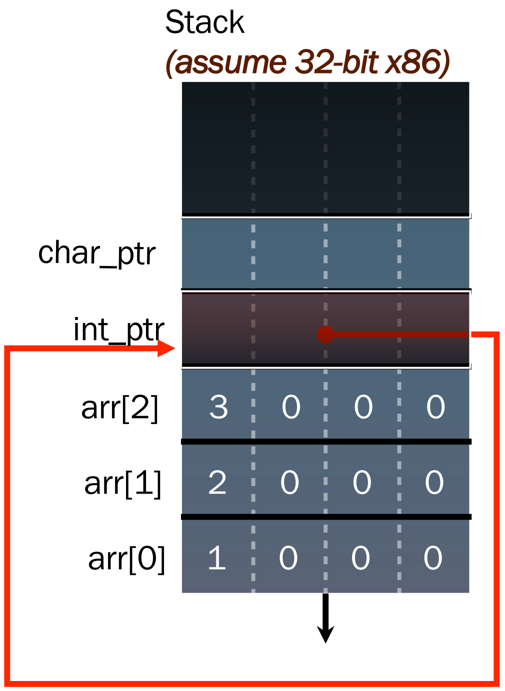
```c
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```


```
int_ptr: 0xbffff2ac;  *int_ptr: 1
int_ptr: 0xbffff2b0;  *int_ptr: 2
```

pointerarithmetic.c

Sample Pointer Arithmetic

```
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```



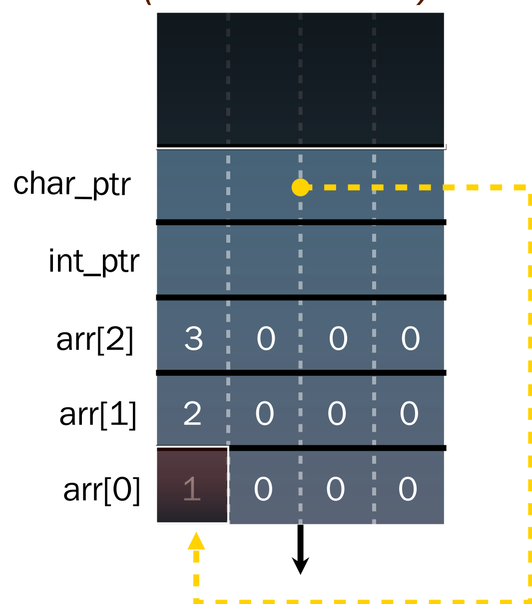

```
int_ptr: 0xbffff2ac;  *int_ptr: 1
int_ptr: 0xbffff2b0;  *int_ptr: 2
int_ptr: 0xbffff2b8;  *int_ptr: -1073745224
```

Sample Pointer Arithmetic

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```

Stack

(assume 32-bit x86)

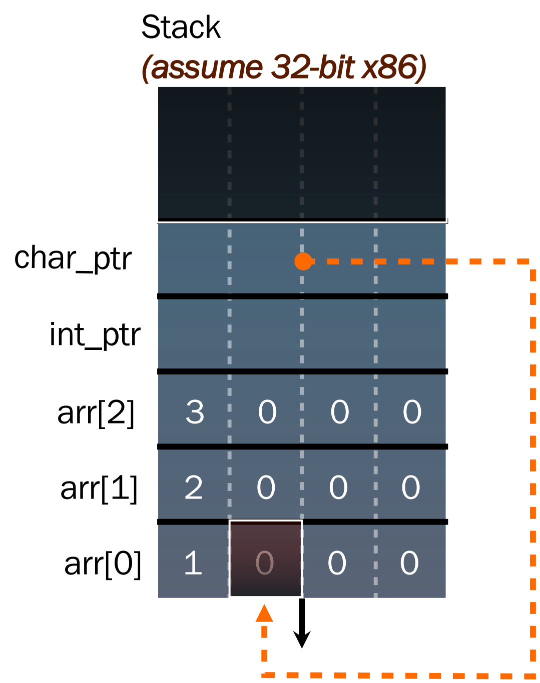



```
char_ptr: 0xbffff2ac;  *char_ptr: 1
```

pointerarithmetic.c

Sample Pointer Arithmetic

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```

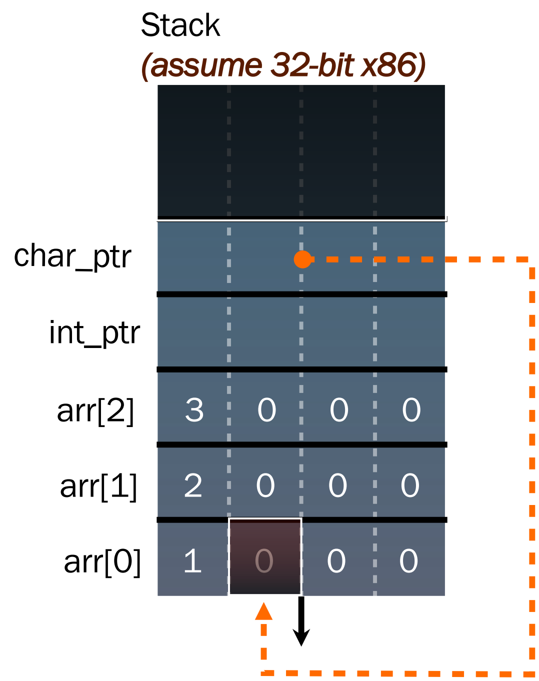



```
char_ptr: 0xbffff2ac;  *char_ptr: 1
```

ointerarithmetic.c

Sample Pointer Arithmetic

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```



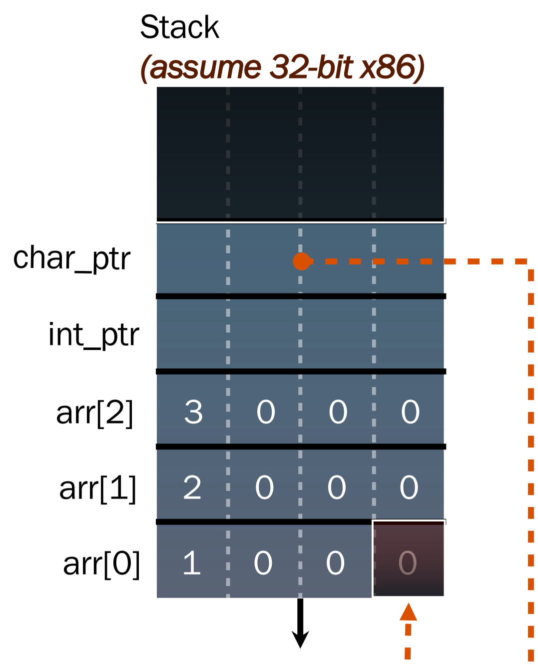

```
char_ptr: 0xbffff2ac;  *char_ptr: 1
char_ptr: 0xbffff2ad;  *char_ptr: 0
```

pointerarithmetic.c

Sample Pointer Arithmetic

```
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```

pointerarithmetic.c

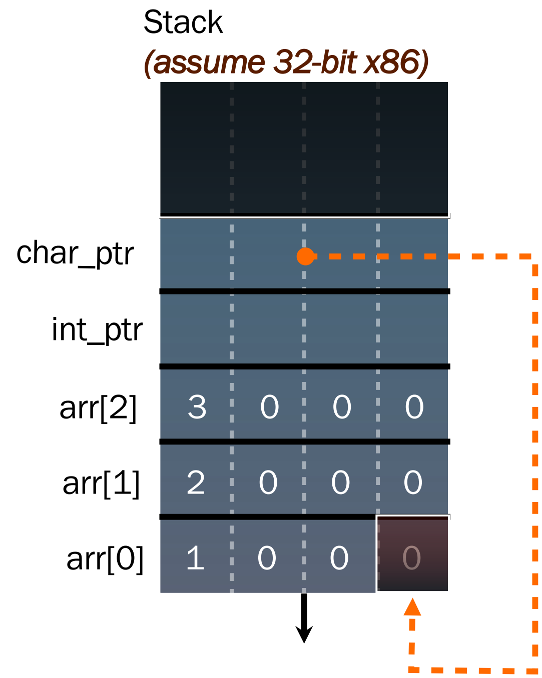



```
char_ptr: 0xbffff2ac;  *char_ptr: 1
char_ptr: 0xbffff2ad;  *char_ptr: 0
```

Sample Pointer Arithmetic

PennState

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int arr[3] = {1, 2, 3};
  int *int_ptr = &arr[0];
  char *char_ptr = (char *) int_ptr;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 1;
  printf("int_ptr: %p;  *int_ptr: %d\n",
         int_ptr, *int_ptr);
  int_ptr += 2;  // uh oh
  printf("int_ptr: %p;   *int_ptr: %d\n",
        int_ptr, *int_ptr);

  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 1;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  char_ptr += 2;
  printf("char_ptr: %p;  *char_ptr: %d\n",
         char_ptr, *char_ptr);
  return 0;
}
```




```
char_ptr: 0xbffff2ac;  *char_ptr: 1
char_ptr: 0xbffff2ad;  *char_ptr: 0
char_ptr: 0xbffff2af;  *char_ptr: 0
```

pointerarithmetic.c

Buffers

- We have used the term “buffer” quite a lot, but never really defined it.
  - One way to think of it is that a buffer is just a memory region that has some use
  - Typically is referenced (maintained) through a pointer

```
char *buffer;
```

Definition: a buffer is a temporary holding place.

Void buffers?

- Often you will see buffers defined as `void *`.
  - This is a general purpose pointer (pointer to a raw address)
  - There is no data type for this, thus the compiler has no idea what this pointing to.
    - Use casting to coerce a type for use

```
char arr[4] = {'a', 'b', 'c', 'd'};
void *ptr = arr;
printf( "As pointer : %p\n", ptr);
printf( "As character : %c\n", *((char *)ptr));
printf( "As 32 bit int : %d\n", *((int32_t *)ptr));
```

```
As pointer : 0x7fff51a2a7fc
As character : a
As 32 bit int : 1684234849
```

Copying memory

PennState

- `memcpy` copies one memory region to another
  - Copy from “source” buffer to “destination” buffer
  - The size must be explicit (because there is no terminator)

```
char buf1[] = { 0, 1, 2, 3 };
char buf2[4] = { 0, 0, 0, 0 };

printf( "Before\n" );
for (i=0; i<4; i++) {
  printf( "buf1[i] = %1d, buf2[i] = %1d\n",
          (int)buf1[i], (int) buf2[i] );
}

memcpy( buf2, buf1, 4 ); // Copy the buffers

printf( "After\n" );
for (i=0; i<4; i++) {
  printf( "buf1[i] = %1d, buf2[i] = %1d\n",
          (int) buf1[i], (int) buf2[i] );
}
```

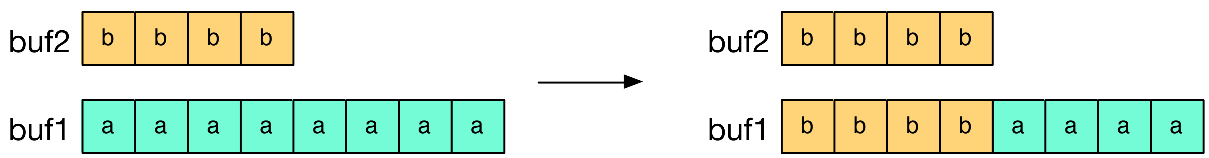
`memcpy(dest, src, n)`
is kinda like `dest = src`

```
Before
buf1[i] = 0, buf2[i] = 0
buf1[i] = 1, buf2[i] = 0
buf1[i] = 2, buf2[i] = 0
buf1[i] = 3, buf2[i] = 0
After
buf1[i] = 0, buf2[i] = 0
buf1[i] = 1, buf2[i] = 1
buf1[i] = 2, buf2[i] = 2
buf1[i] = 3, buf2[i] = 3
```

Copying memory

- `memcpy` copies one memory region to another
  - Copy from “source” buffer to “destination” buffer
  - The size must be explicit (because there is no terminator)

```
char buf1[8] = {'a', 'a', 'a', 'a',
                'a', 'a', 'a', 'a', };
char buf2[4] = {'b', 'b', 'b', 'b', };
memcpy(buf1, buf2, 4);
```

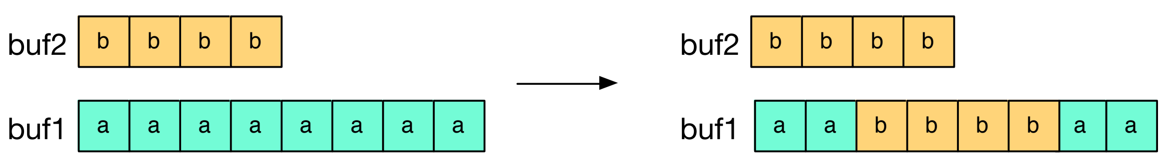


Copying memory

- `memcpy` copies one memory region to another
  - Copy from “source” buffer to “destination” buffer
  - The size must be explicit (because there is no terminator)

```
char buf1[8] = {’a’, ’a’, ’a’, ’a’,
                ’a’, ’a’, ’a’, ’a’, };
char buf2[4] = {’b’, ’b’, ’b’, ’b’, };
memcpy(&buf1[2], buf2, 4);
```

Buffer splicing!



Filling memory

- `memset` fills memory with a given constant byte

```
void *memset(void *buf, int c, size_t n);
```

```
char a[4] = {0, 1, 2, 3};
memset(arr, 0, 4);
// what are the contents of a now?

int b[4] = {0, 1, 2, 3};
memset(b, 0, 4);
// what are the contents of b now?
```

Memory comparison …

- We often want to compare buffers to see if they match or are byte-wise smaller or larger
- `memcmp` compares first `n` bytes of buffers

```
memcmp(buf1, buf2, n);
```

- The comparison functions return
  - negative integer if buf1 is less than buf2
  - 0 if buf1 is equal to buf2
  - positive integer is buf1 greater than buf2

Memcmp example

PennState

```
int i, j, x;
char cmps[4][2] = {
    { 0x0, 0x0 }, { 0x1, 0x0 },
    { 0x0, 0x1 }, { 0x9, 0x0 } };

for (i=0; i<4; i++) {
    for (j=0; j<4; j++) {
            x = memcmp( &cmps[i][0], &cmps[j][0], 2
    );
        printf( "compare %1d%1d with %1d%1d = %d\n",
                cmps[i][0], cmps[i][1],
                cmps[j][0], cmps[j][1], x );
    }
}
```

```
compare 00 with 00 = 0
compare 00 with 10 = -1
compare 00 with 01 = -256
compare 00 with 90 = -9
compare 10 with 00 = 1
compare 10 with 10 = 0
compare 10 with 01 = 1
compare 10 with 90 = -8
compare 01 with 00 = 256
compare 01 with 10 = -1
compare 01 with 01 = 0
compare 01 with 90 = -9
compare 90 with 00 = 9
compare 90 with 10 = 8
compare 90 with 01 = 9
compare 90 with 90 = 0
```

Memory allocation

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

```c
// a global variable
int counter = 0;

int main(int argc, char **argv)
{
  counter++;
  return 0;
}
```

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

int main(int argc, char **argv) {
  int y = foo(10);  // local var
  return 0;
}
```

`counter` is *statically* allocated

- allocated when program is loaded
- deallocated when program exits

`a,x,y` are *automatically* allocated

- allocated when function is called
- deallocated when function returns

We need more flexibility

- Sometimes we want to allocate memory that:
  - persists across multiple function calls but for less than the lifetime of the program
  - is too big to fit on the stack
  - is allocated and returned by a function and its size is not known in advance to the caller (this is called *dynamic* memory)

```
// (this is pseudo-C-code)
char *ReadFile(char *filename) {
  int   size = FileSize(filename);
  char *buffer = AllocateMemory(size);
  ReadFileIntoBuffer(filename, buffer);
  return buffer;
}
```

Dynamic allocation

- What we want is *dynamically allocated memory*
  - your program explicitly requests a new block of memory
    - the language runtime allocates it, perhaps with help from OS
  - dynamically allocated memory persists until:
    - your code explicitly deallocates it *[manual memory management]*
      - *C, C++, Rust*
    - a garbage collector collects it *[automatic memory management]*
      - *Java, Python, Go*
  - C requires you to manually manage memory
    - gives you more control, but causes headaches
    - <u>*C has no garbage collection*</u>

Garbage collection

- In some languages like Java, you can dynamically allocate objects using the built in "`new`" function of the Java runtime environment

```
String str1 = new String("This is a text string");
```

- Stays in memory until an invisible process behind the scenes frees.
  - This invisible process is called garbage collection
  - Typically done in the background by a background process
- Pros: a large class of memory bugs avoided
  - dereferencing dangling pointers, memory leaks, double freeing of memory, …
- Cons: unpredictable performance
  - garbage collection can start at an arbitrary time and slow down the program

C and malloc

- `malloc` allocates a block of memory of the given size

  `void *malloc(size_t size)`

  - returns a `void` pointer to the first byte of that memory
  - no need to cast – void pointer is automatically promoted
    - `malloc` returns `NULL` if the memory could not be allocated
  - you should assume the memory initially contains garbage
  - you'll typically use `sizeof` to calculate the size you need

```
// allocate a 10-float array
float *arr = malloc(10*sizeof(float));
if (arr == NULL)
  return errcode;
arr[0] = 5.1;
```

  - (aside: Linux overcommits memory by default, malloc never fails)

C and calloc

- Similar to malloc, but zeroes out allocated memory

```
void *calloc(size_t nmemb, size_t bytes);
```

  - Returns an array of `nmemb` members, each of size bytes
  - Memory is zeroed out (all bytes have the value 0x0)
  - slightly slower; preferred for non-performance-critical code
  - malloc and calloc are found in *stdlib.h*

```
// allocate a 10 long-int array
long *arr = calloc(10,sizeof(long));
if (arr == NULL)
  return errcode;
arr[0] = 5L;  // etc.
```

Deallocation

- Releases the memory pointed-to by the pointer

```
void free(void *ptr);
```

  - pointer must point to the first byte of heap-allocated memory
    - i.e., something previously returned by malloc( ) or calloc( )
  - after free( )ing a block of memory, that block of memory might be returned in some future malloc( ) / calloc( )
  - it's good form to set a pointer to NULL after freeing it
    - otherwise we get a dangling pointer

```
long *arr = calloc(10*sizeof(long));
if (arr == NULL)
  return errcode;
// .. do something ..
free(arr);
arr = NULL;
```

Dynamically allocated `structs`

- You can `malloc()` and `free()` structs, as with other types
  - `sizeof()` is particularly helpful here

```c
typedef struct {
  double real;  // real component
  double imag;  // imaginary component
} Complex, *ComplexPtr;

ComplexPtr AllocComplex(double real, double imag) {
  Complex *retval = malloc(sizeof(Complex));
  if (retval != NULL) {
    retval->real = real;
    retval->imag = imag;
  }
  return retval;
}
```

Realloc (re-allocation)

- `realloc` changes a previous allocation (resizing it)

  `void *realloc(void *ptr, size_t size)`

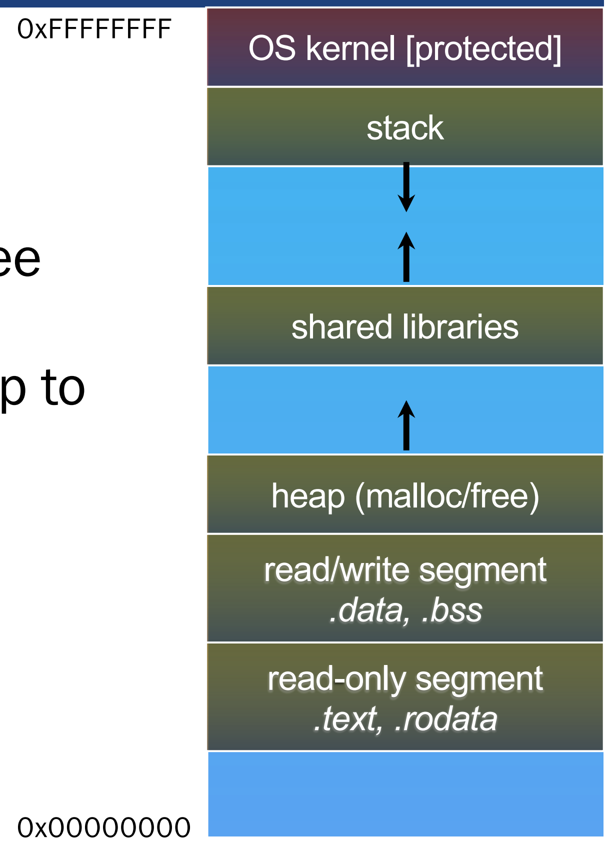
  - Resizes the previous allocation in place, if possible
  - If it can't, it creates a new allocation and copies as much data as it can
  - returns `NULL` if the memory could not be allocated

```
// allocate a 10-float array
char *buf, *rbuf;
buf = malloc(2); // allocate 2 byte array
memset(buf, 0xa, 2);
printf("buf = %x %x\n", buf[0], buf[1]);
rbuf = realloc(buf, 4); // resize to 4 bytes
printf("rbuf = %x %x %x %x\n", rbuf[0], rbuf[1],
         rbuf[2], rbuf[3]);
```

```
buf = a a
rbuf = a a 0 0
```

Heap

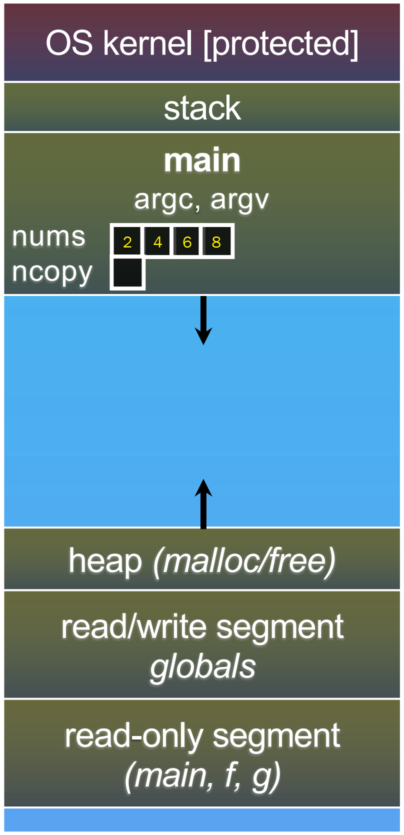
- The heap (aka "free store")
  - is a large pool of unused memory that is used for dynamically allocated data
    - `malloc` allocates chunks of data in the heap, free deallocates data
    - `malloc` maintains book-keeping data in the heap to track allocated blocks
- malloc and friends are ordinary functions in the standard C library (or libc)
  - you may implement them in future classes



Heap + stack

```c
#include <stdlib.h>

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

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

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

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

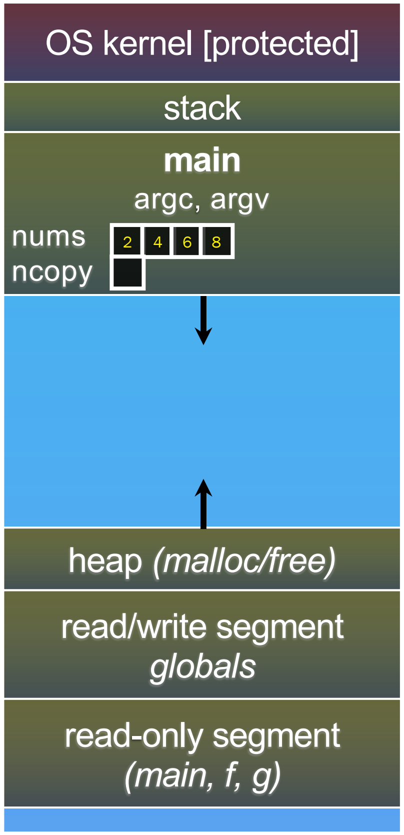
arraycopy.c



Heap + stack

```c
#include <stdlib.h>

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

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

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

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

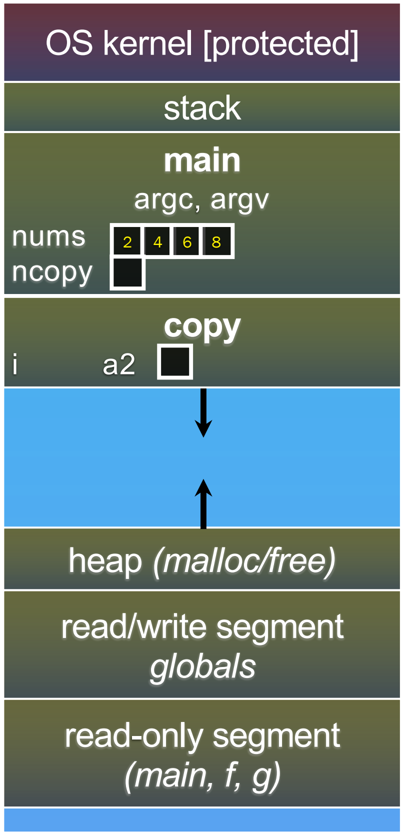
arraycopy.c



Heap + stack

```
#include <stdlib.h>

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

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

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

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

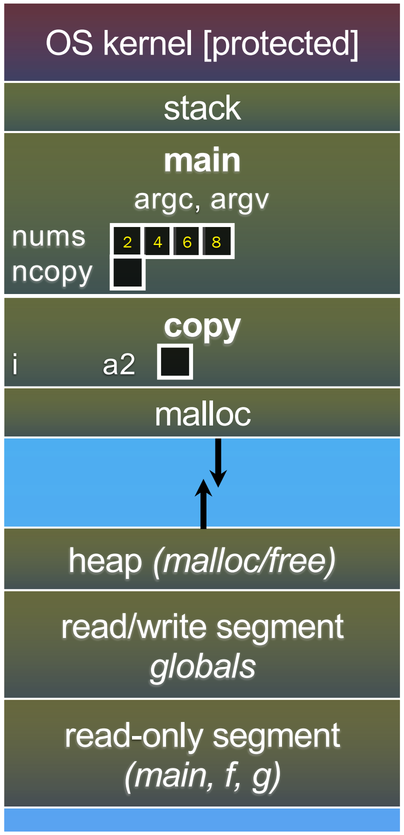
arraycopy.c



Heap + stack

```c
#include <stdlib.h>

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

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

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

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

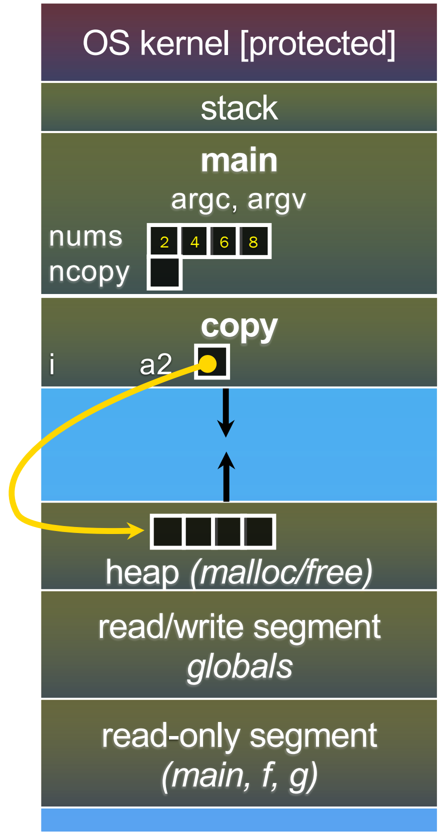
arraycopy.c



Heap + stack

```c
#include <stdlib.h>

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

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

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

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

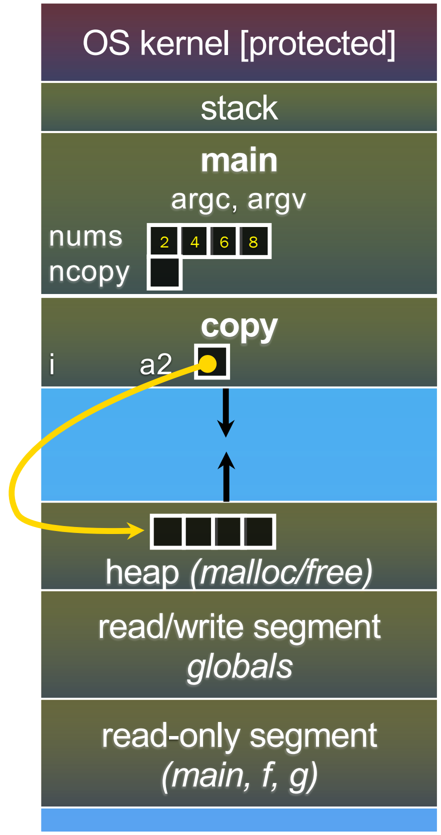
arraycopy.c



Heap + stack

```c
#include <stdlib.h>

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

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

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

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

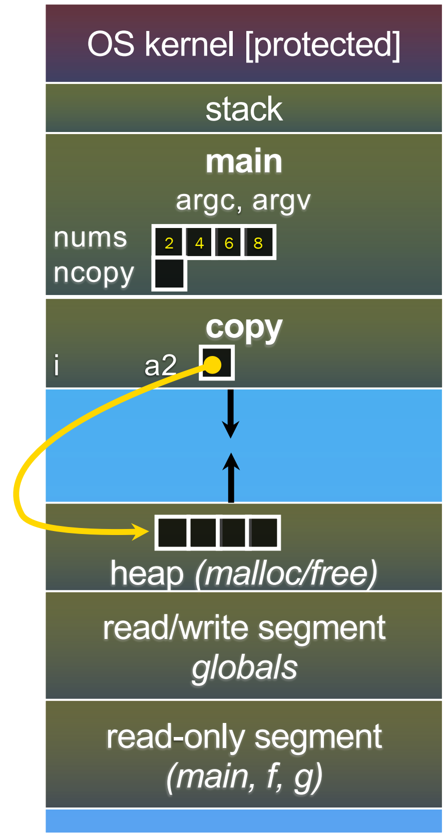
arraycopy.c



Heap + stack

```c
#include <stdlib.h>

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

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

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

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

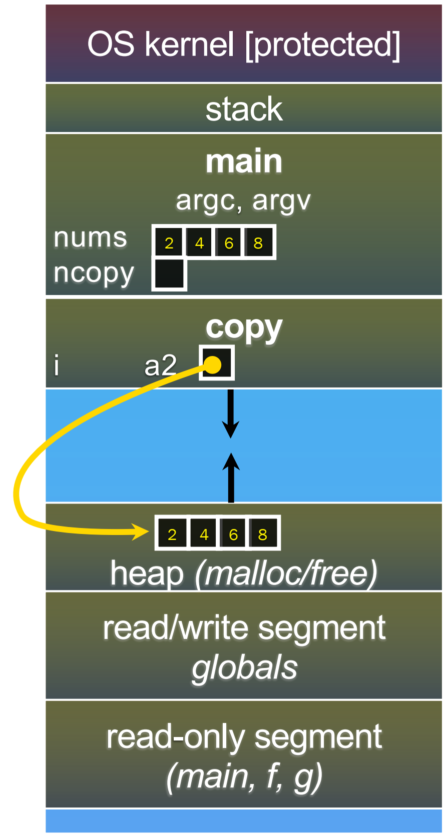
arraycopy.c



Heap + stack

```
#include <stdlib.h>

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

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

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

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

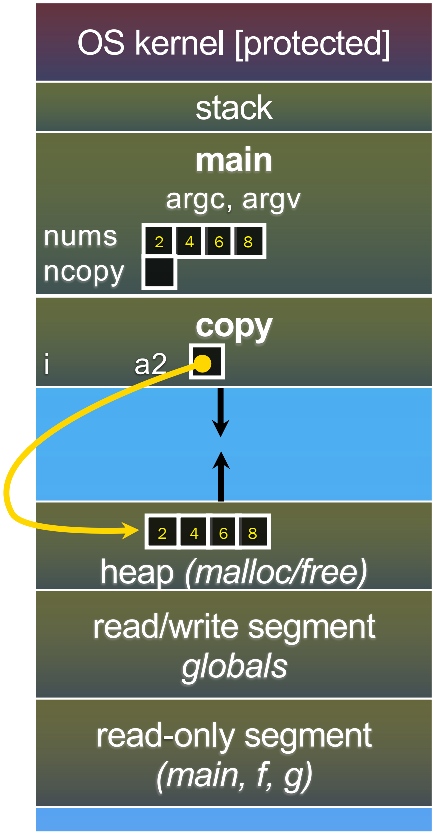
arraycopy.c



Heap + stack

```c
#include <stdlib.h>

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

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

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

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

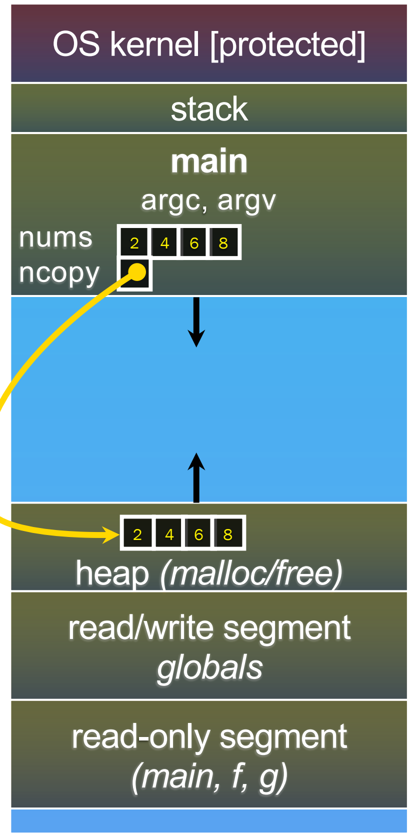
arraycopy.c



Heap + stack

```c
#include <stdlib.h>

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

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

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

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

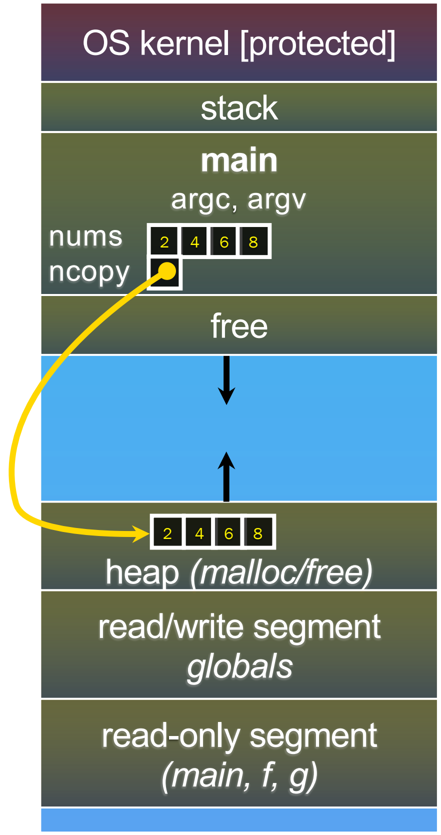
arraycopy.c



Heap + stack

```c
#include <stdlib.h>

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

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

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

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

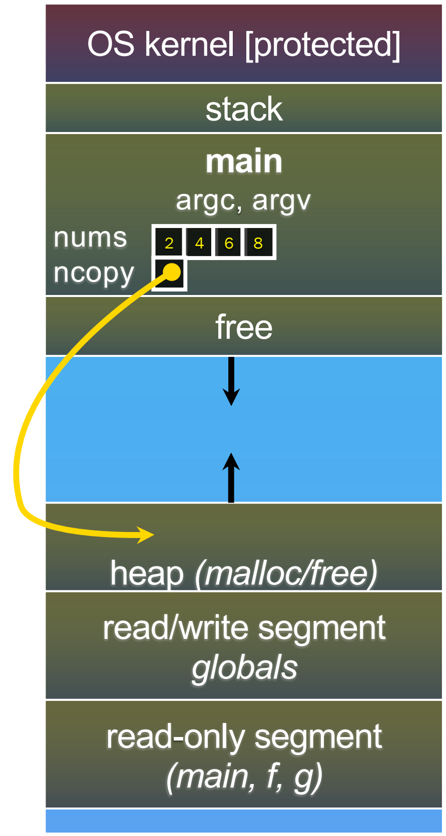
arraycopy.c



Heap + stack

```c
#include <stdlib.h>

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

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

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

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

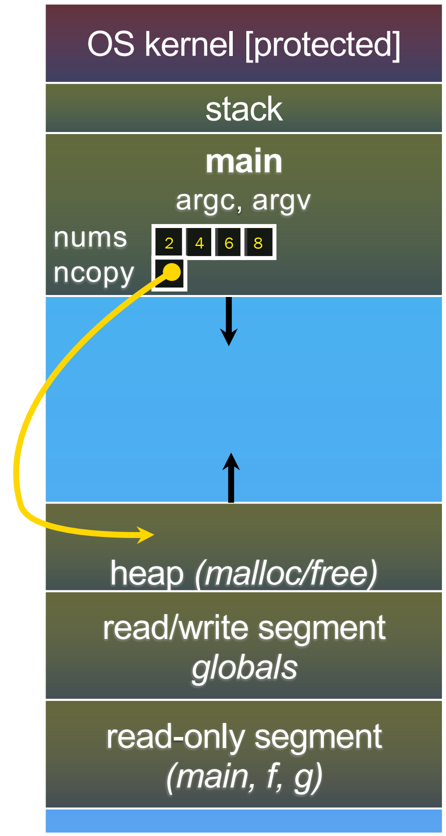
arraycopy.c



Heap + stack

```
#include <stdlib.h>

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

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

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

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

arraycopy.c



NULL

- NULL: a guaranteed-to-be-invalid memory location
  - in C on Linux:
    - NULL is 0x00000000
    - an attempt to deference NULL causes a segmentation fault
  - that's why you should NULL a pointer after you have free()d it
    - it's better to have a segfault than to corrupt memory!

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int *p = NULL;
  *p = 1;  // causes a segmentation fault
  return 0;
}
```

Memory corruption

- All sorts of ways to corrupt memory in C

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

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

  b[2] = 5;    // assign past the end of an array
  b[0] += 2;   // assume malloc zeroes out memory
  c = b+3;     // mess up your pointer arithmetic
  free(a);     // free() something not malloc()'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a free()'d pointer

  // any many more!
  return 0;
}
```

memcorrupt.c

Memory leak

- A memory leak happens when code fails to deallocate dynamically allocate memory that will no longer be used

```
// assume we have access to functions FileLen,
// ReadFileIntoBuffer, and NumWordsInString.

int NumWordsInFile(char *filename) {
  char *filebuf = malloc(FileLen(filename)+1);
  if (filebuf == NULL)
    return -1;

  ReadFileIntoBuffer(filename, filebuf);

  // leak! we never free(filebuf)
  return NumWordsInString(filebuf);
}
```

Implications of a leak?

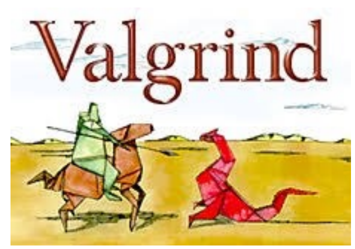
- A program's virtual memory will keep growing
  - for short-lived programs, this might be OK
  - for long-lived programs, this usually has bad repercussions
    - might slow down over time (VM thrashing)
      - *potential "DoS attack" if a server leaks memory*
    - might exhaust all available memory and crash
    - other programs might get starved of memory
  - in some cases, you might prefer to leak memory than to corrupt memory with a buggy `free()`

Memory Debuggers

- Tools for finding memory corruption/leak bugs, buffer overflows…
  - Purify
  - Valgrind
  - AddressSanitizer
  - …

Purify:
Fast Detection of Memory Leaks and Access Errors

Reed Hastings and Bob Joyce
Pure Software Inc.



- AddressSanitizer is built into your compiler
  - gcc –fsanitize=address foo.c –o foo

- Let's do a quick demo

AddressSanitizer: A Fast Address Sanity Checker

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitry Vyukov
Google
{kcc,bruening,glider,dvyukov}@google.com

Summary: dynamic memory interface

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

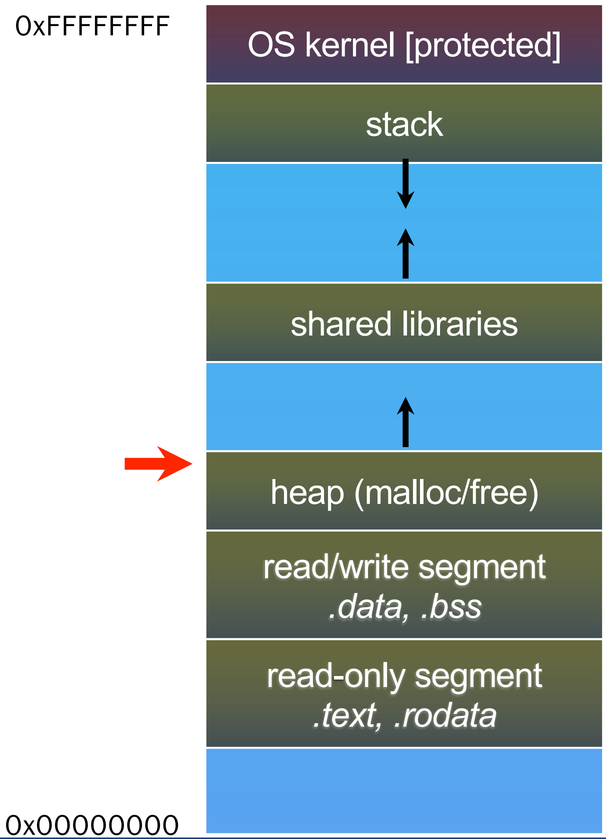
- Allocators are user-space libraries. You can write one yourself. (And you will.)
- Example allocators
  - dlmalloc (Doug Lea's malloc – you are using its derivative)
  - tcmalloc (thread-caching malloc by Google)
  - jemalloc (Jason Evan's malloc – used by FreeBSD)

Whence virtual memory?

- Every program begins with a certain amount of memory in its heap?
  - The top of the heap is known and the *program break*.
  - Functions like `malloc()` and `free()` handle the management of the heap by obtaining and releasing memory.
    - You don't see it because it is being handled for you.

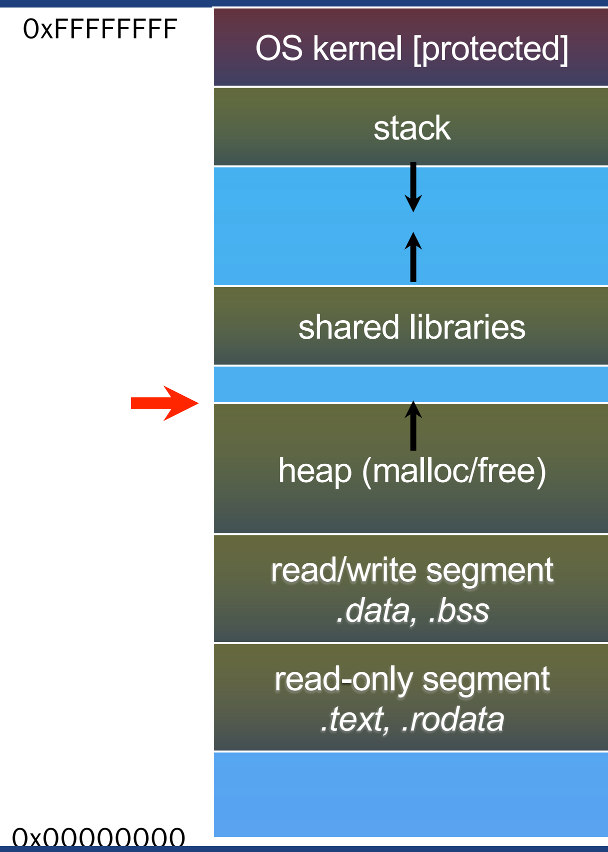
Program break

- The program break gets moved up and down as memory is allocated and deallocated from the heap.



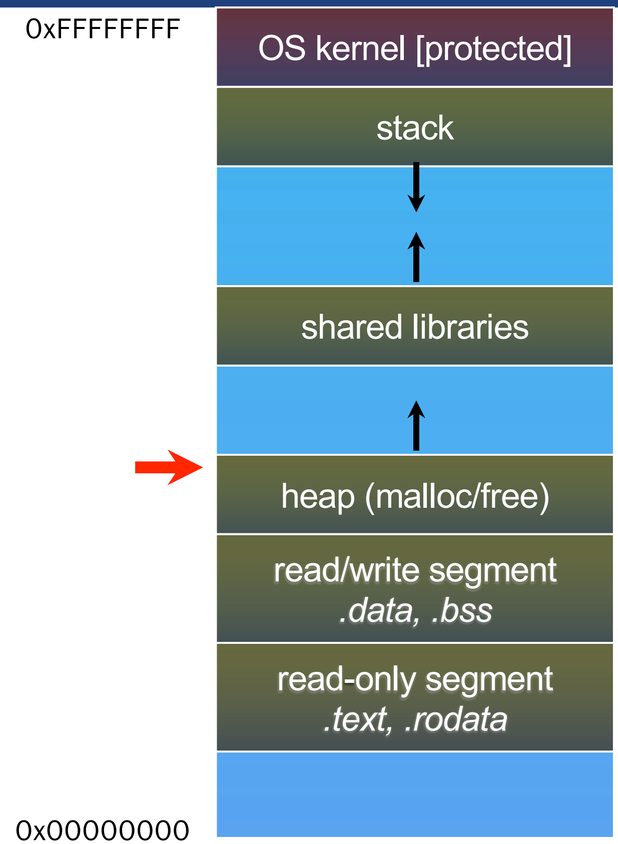
Program break

- The program break gets moved up and down as memory is allocated and deallocated from the heap.



Program break

- The program break gets moved up and down as memory is allocated and deallocated from the heap.



Q: how?

brk() and sbrk()

- The functions are used to manage the program break
  - `void * brk( void *ptr )` - changes the new program break to be the at the address of `ptr`?
    - This is an absolute pointer, so very dangerous
    - For example, by putting `ptr` in the stack
    - Calling `brk(NULL)` returns the current program break
  - `void * sbrk( int inc )` - moves the program break `inc` (increment) bytes upwards or downwards
    - A positive `inc` allocates new memory
    - A negative `inc` frees memory

These are really just wrappers for systems calls?

Lets try it.

```
void *last = 0x0;

int check_memory( void ) {
        void *ptr = sbrk(0);
        printf( "The top of the heap is %p %ld\n", ptr, ptr-last);
        last = ptr;
        return( 0 );
}

int main( void ) {
        void *xptr[2048];
        int i;

        last = sbrk(0);                      // Get initial state
        check_memory();
        xptr[0] = malloc( 0x1000 );          // Allocate buffer
        check_memory();
        for (i=1; i<1024; i++) {
                xptr[i] = malloc( 0x1000 );  // Allocate more buffers
        }
        check_memory();
        for (i=0; i<1024; i++) {
                free( xptr[i] );             // Deallocate the buffers
        }
        check_memory();
        return( 0 );
}
```

```
The top of the heap is 0x562b885d3000 0
The top of the heap is 0x562b885f4000 135168
The top of the heap is 0x562b889f3000 4190208
The top of the heap is 0x562b885f4000 -4190208
```