CMPSC 311 - Introduction to Systems Programming

PennState

Systems and Internet Infrastructure Security Labratory

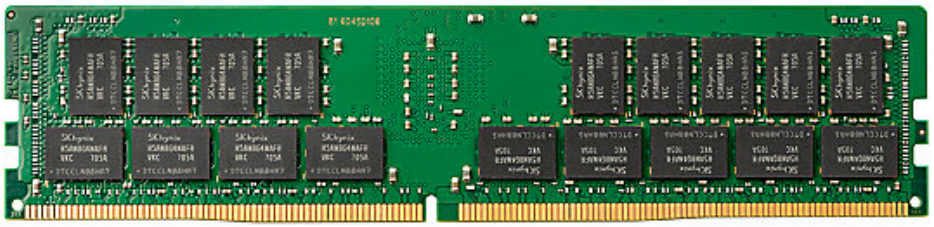
Module: Memory Management

Professor Patrick McDaniel

Memory Management

PennState

- Most of the data you will maintain on a system/program will not be held in variables statically defined or placed on the stack
  - main memory
  - secondary storage

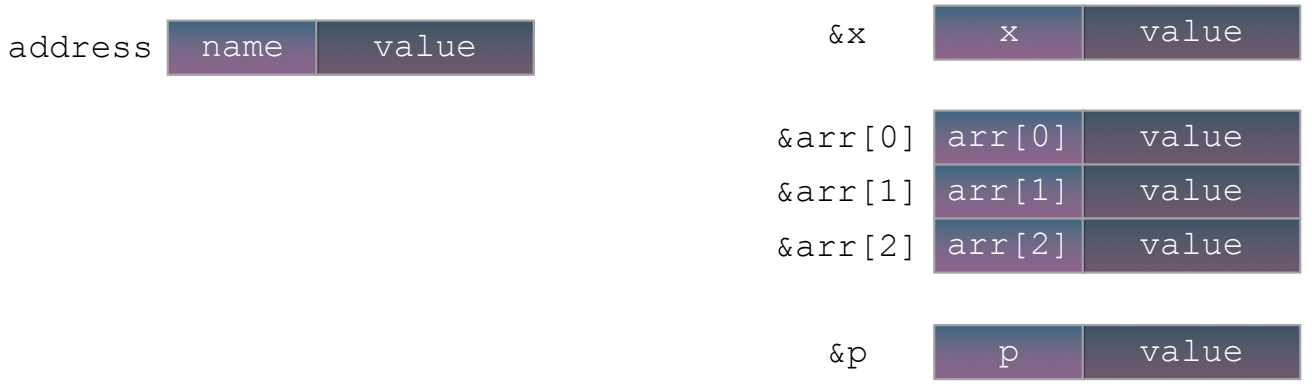


Learning to allocate, manage, and deallocate memory is key to become a good systems programmer.

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;
}
```

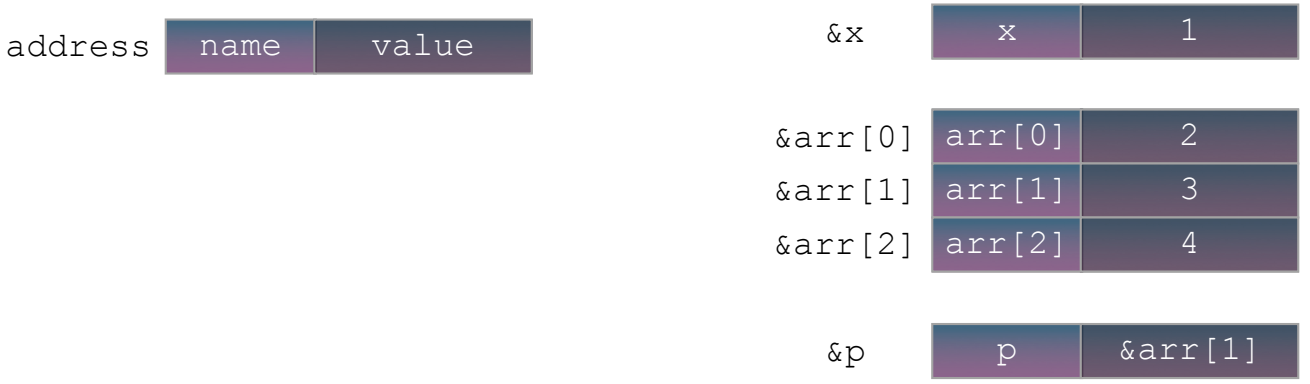


Box and arrow diagrams

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;
}
```

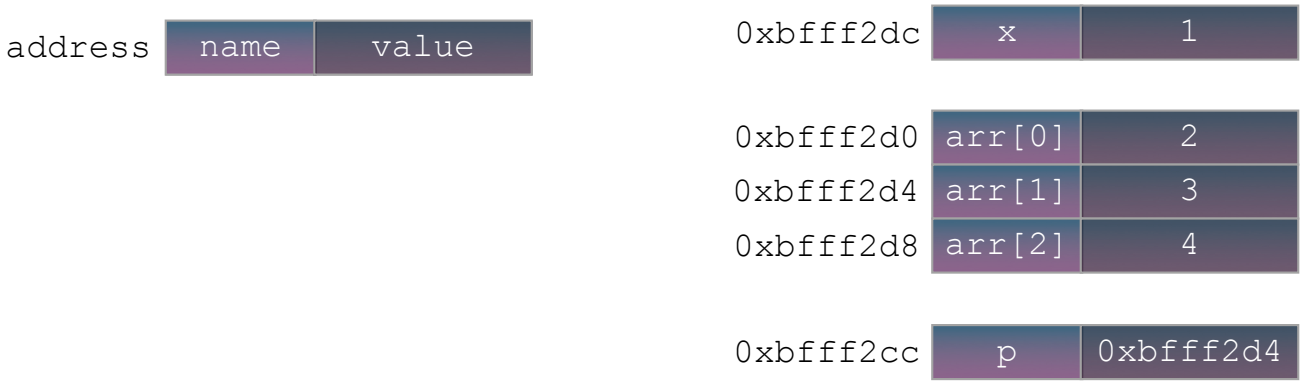


Box and arrow diagrams

PennState

boxarrow.c

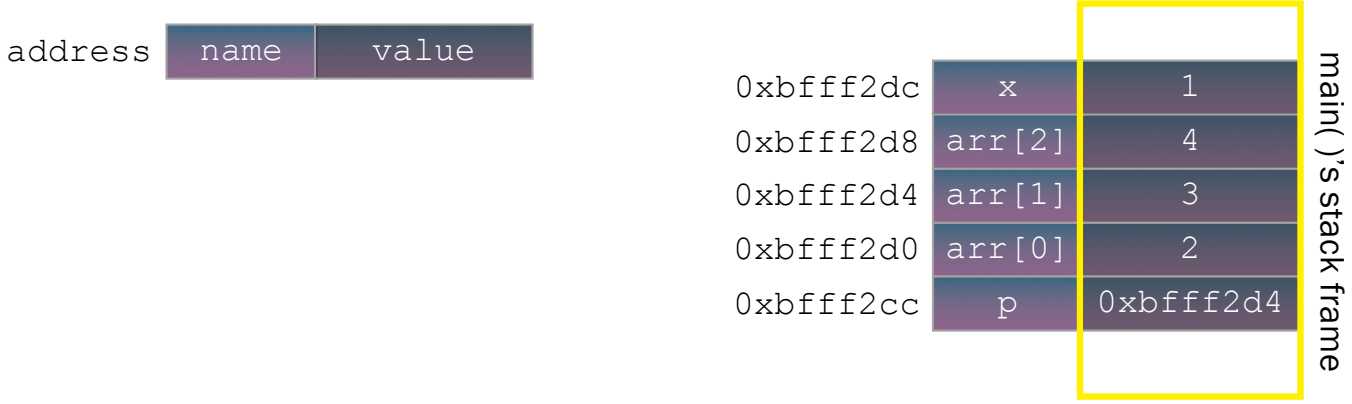
```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];

  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;
}
```



Box and arrow diagrams

boxarrow.c

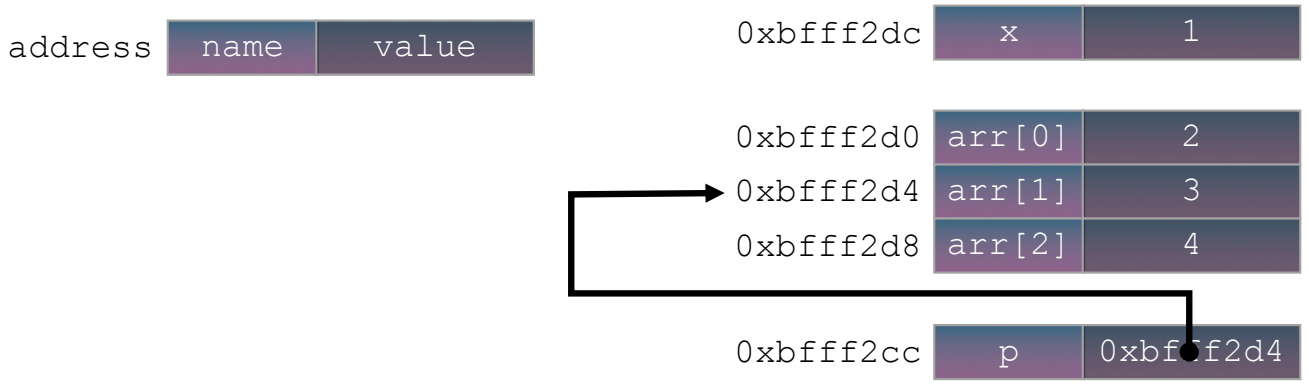
```
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];

  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;
}
```



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

PennState

- 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;
}
```

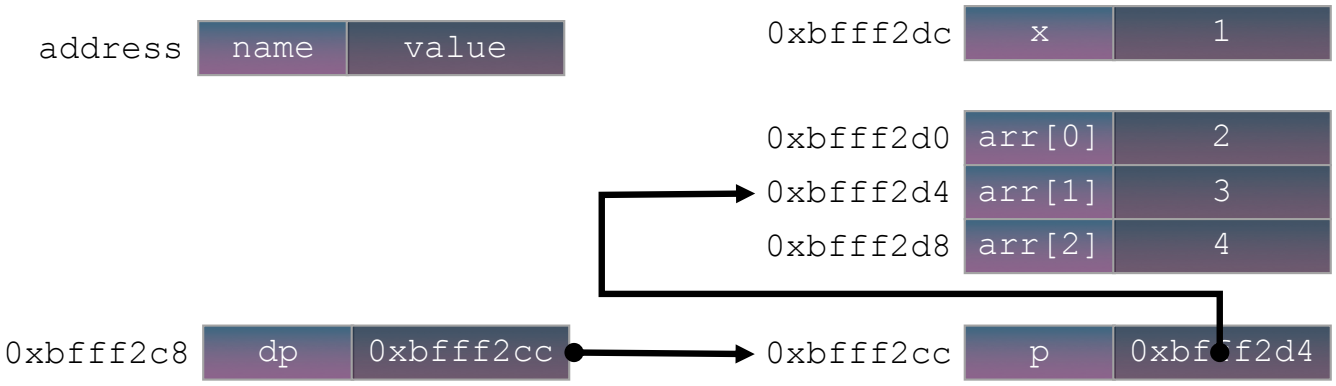
exercise0.c

Box and arrow diagrams

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;
}
```

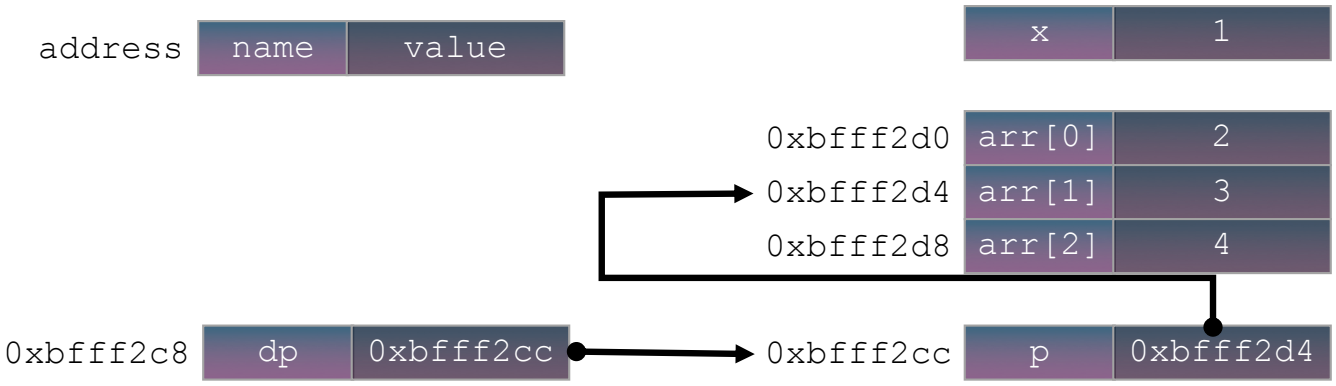


Box and arrow diagrams

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;
}
```

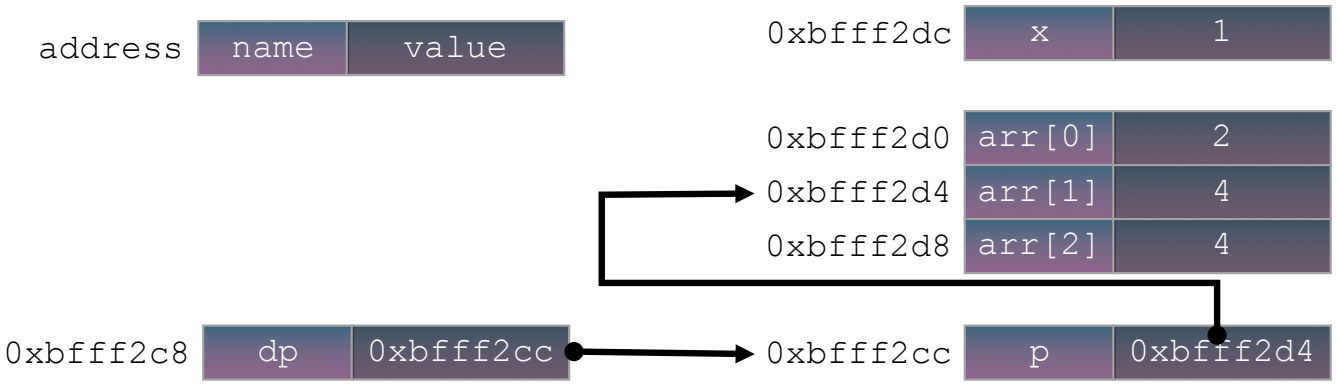


Box and arrow diagrams

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;
}
```

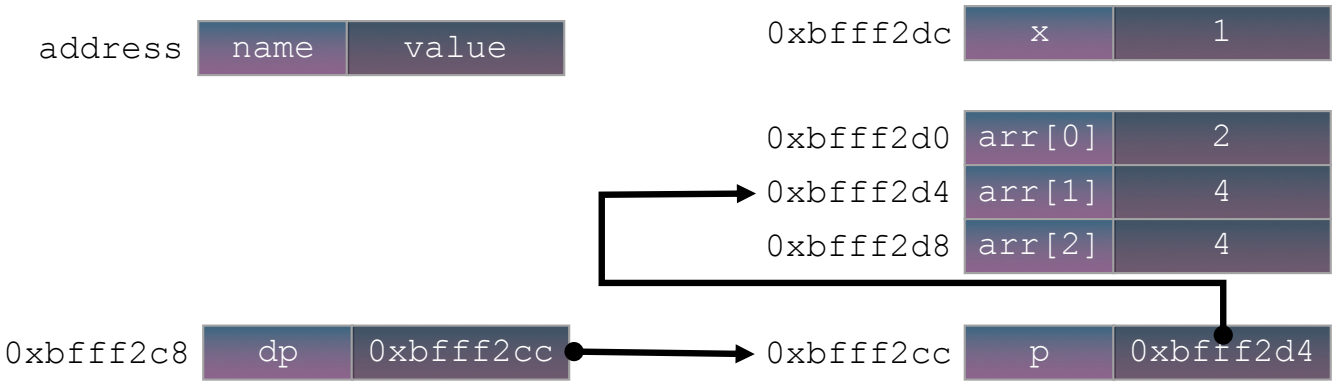


Box and arrow diagrams

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;
}
```

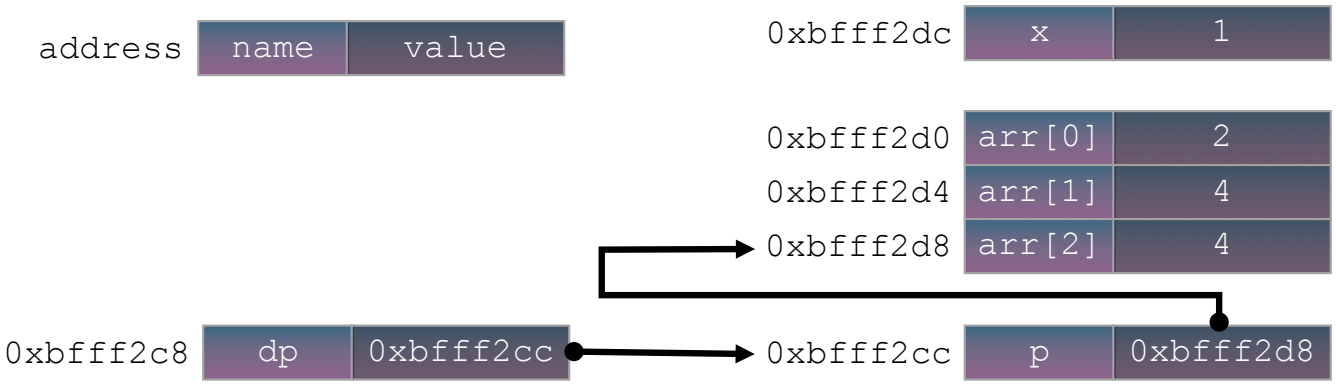


Box and arrow diagrams

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;
}
```

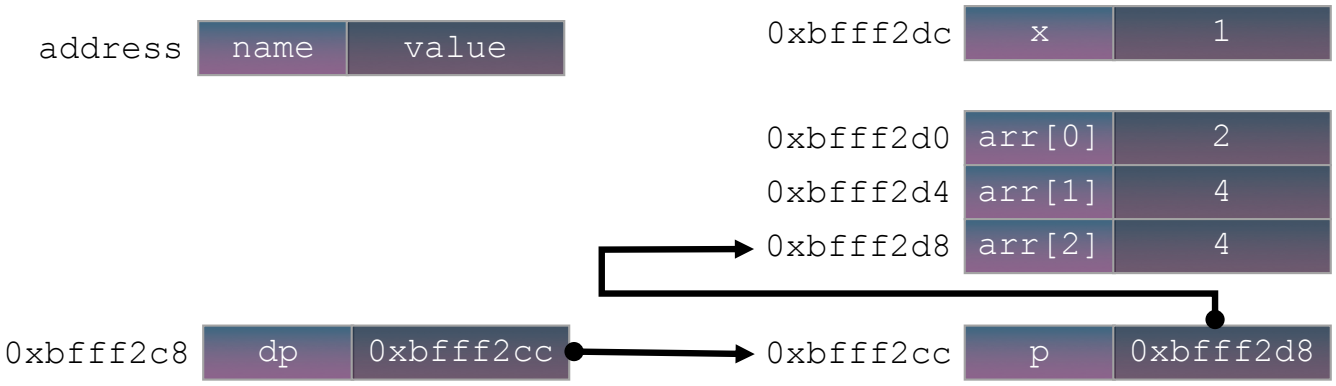


Box and arrow diagrams

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;
}
```

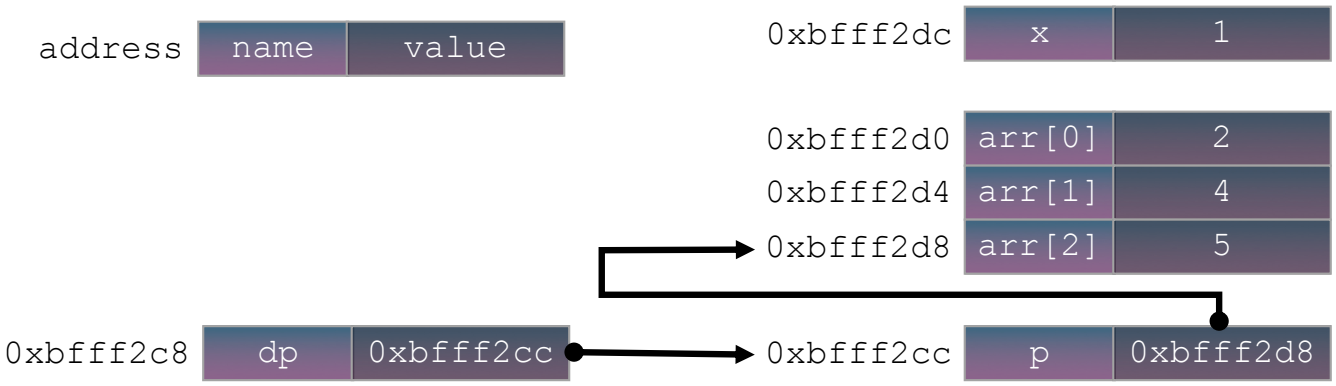


Box and arrow diagrams

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;
}
```



Double pointers

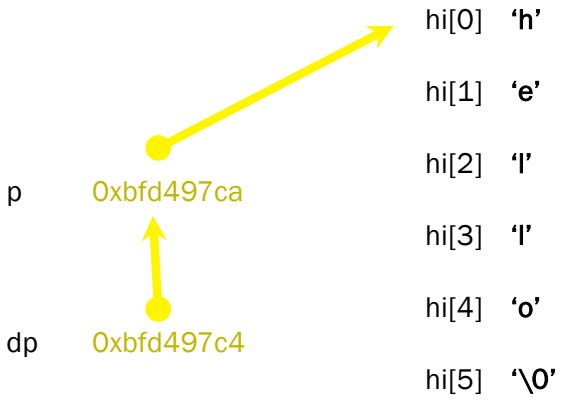
PennState

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

```c
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

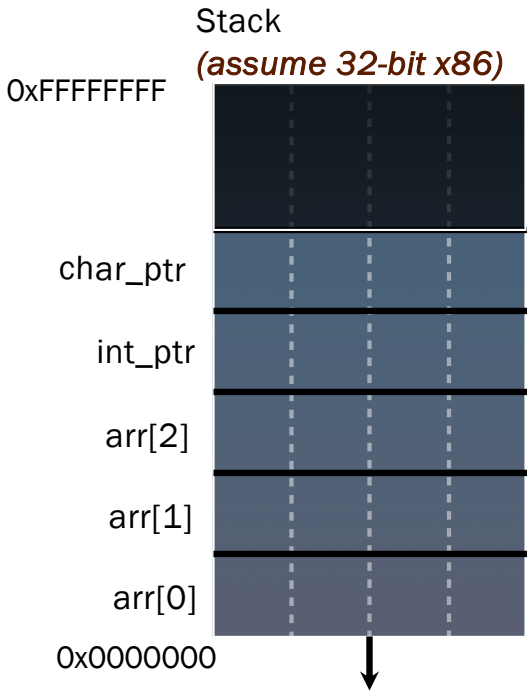
PennState

- 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

PennState

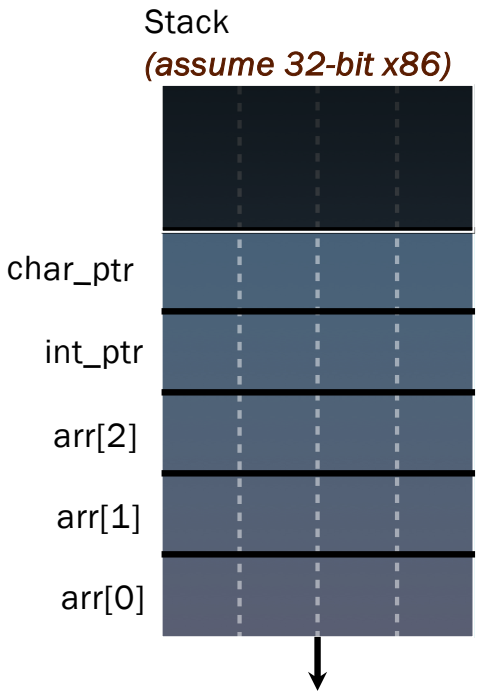
```c
#include <stdio.h>

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;
}
```



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;
}
```



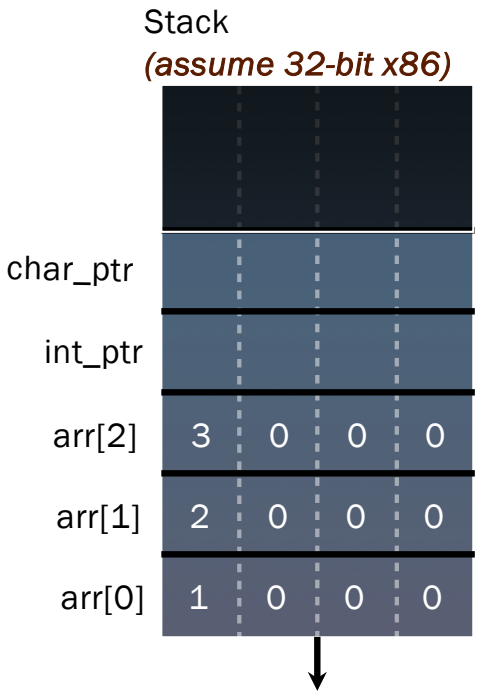
(x86 is little endian)

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;
}
```



pointerarithmetic.c

Sample Pointer Arithmetic

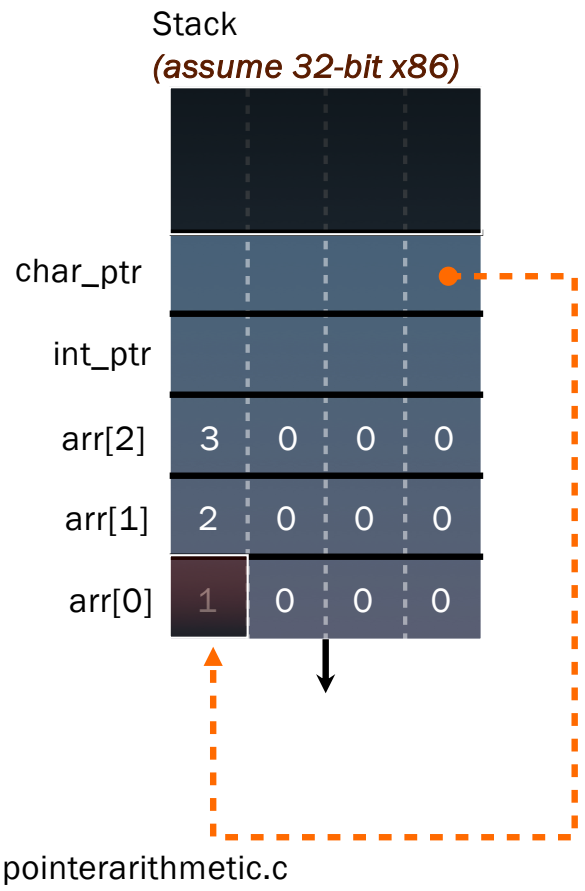
PennState

```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;
}
```

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;
}
```



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
```

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

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;
}
```

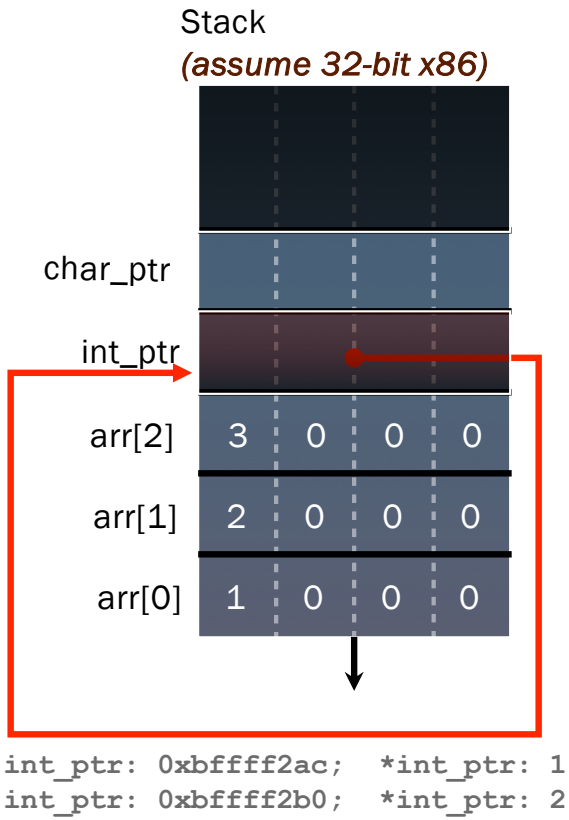

```
int_ptr: 0xbffff2ac;  *int_ptr: 1
int_ptr: 0xbffff2b0;  *int_ptr: 2
```

pointerarithmetic.c

Sample Pointer Arithmetic

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

PennState

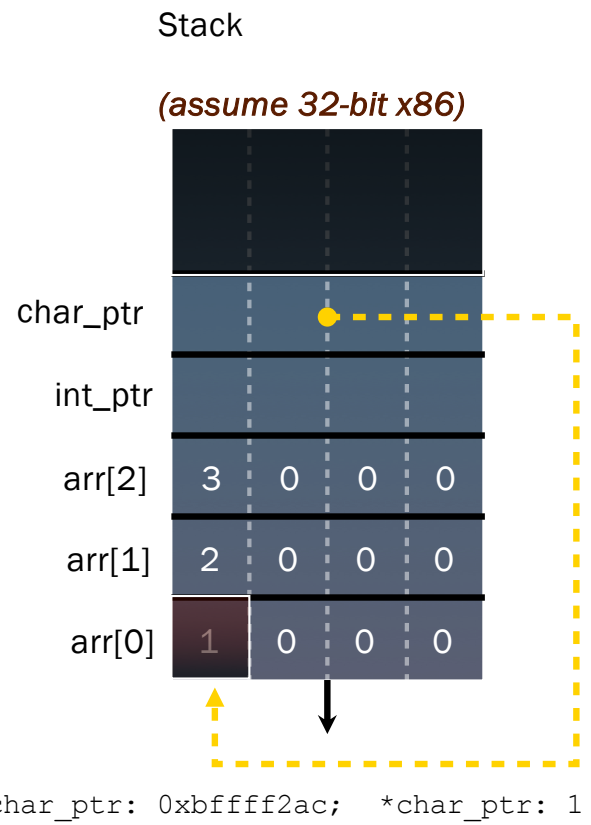
```c
#include <stdio.h>

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
int_ptr: 0xbffff2b8;  *int_ptr: -1073745224
```

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;
}
```

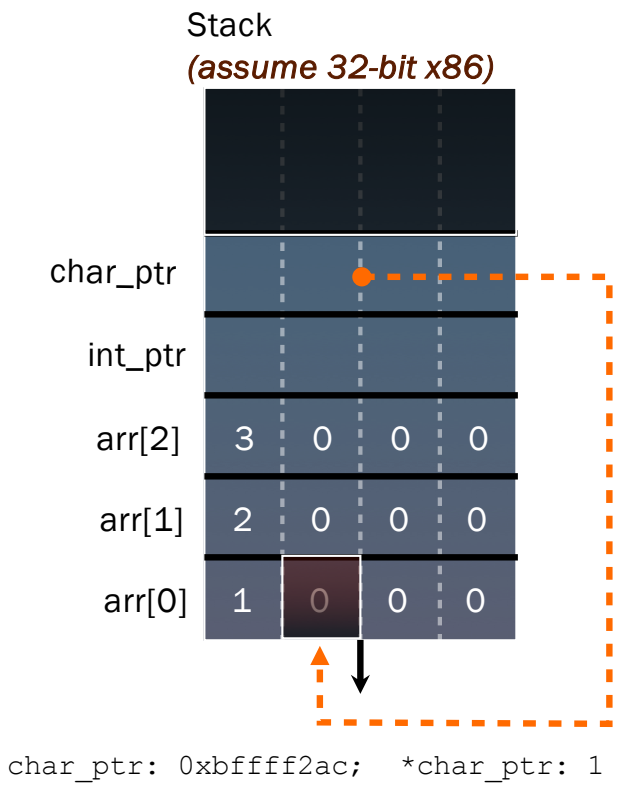


pointerarithmetic.c

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;
}
```



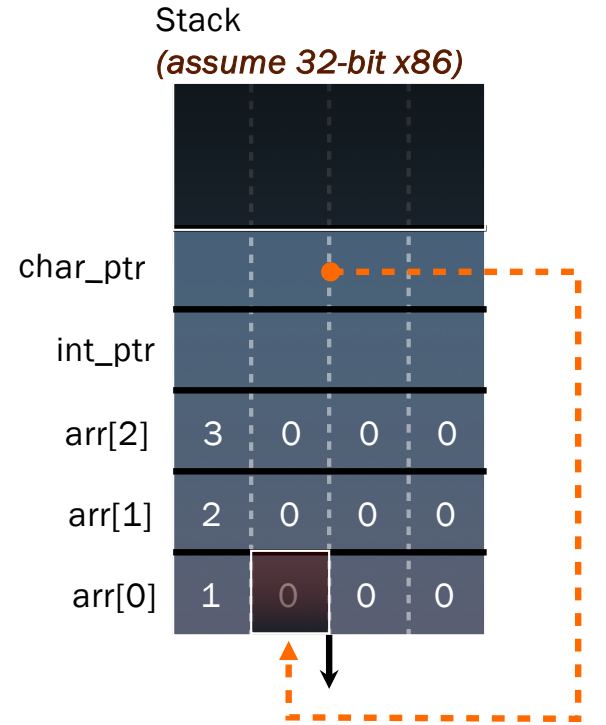

```
char_ptr: 0xbffff2ac;  *char_ptr: 1
```

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;
}
```



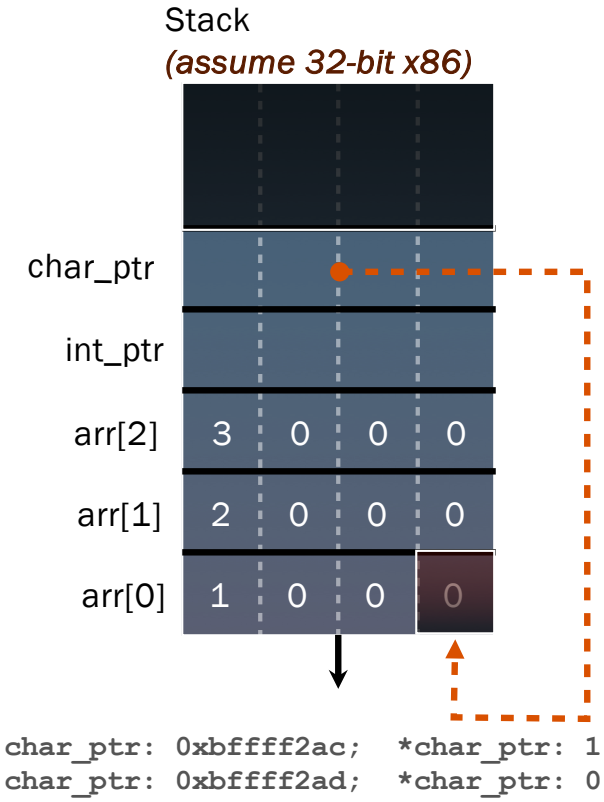

```
char_ptr: 0xbffff2ac;  *char_ptr: 1
char_ptr: 0xbffff2ad;  *char_ptr: 0
```

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;
}
```

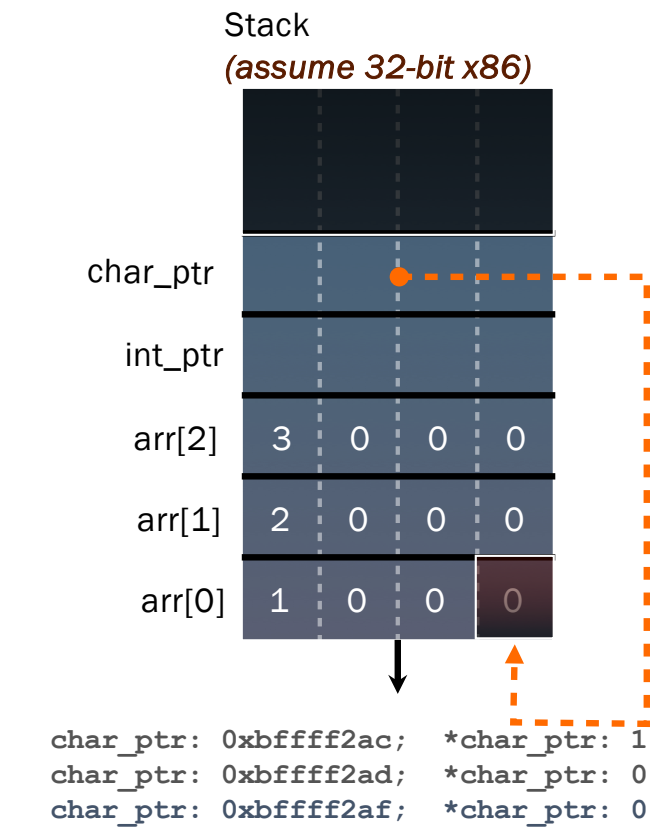


pointerarithmetic.c

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;
}
```



pointerarithmetic.c

Buffers

PennState

- 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?

PennState

- 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`

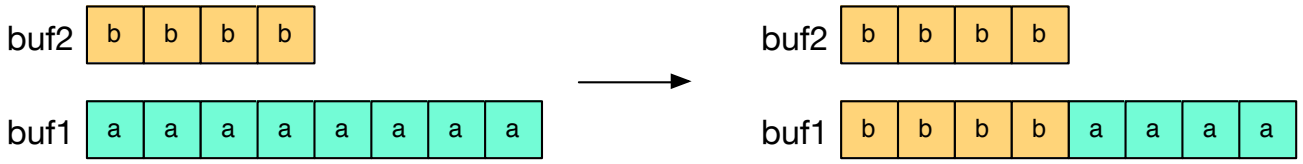
```
Before
buf1[i] = 0, buf2[i] = 0
buf1[i] = 1, buf2[i] = 0
buf1[i] = 2, buf2[i] = 0
buf1[i] = 3, buf2[i] = 0
After
buf1[i] = 0, buf2[i] = 0
buf1[i] = 1, buf2[i] = 1
buf1[i] = 2, buf2[i] = 2
buf1[i] = 3, buf2[i] = 3
```

Copying memory

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



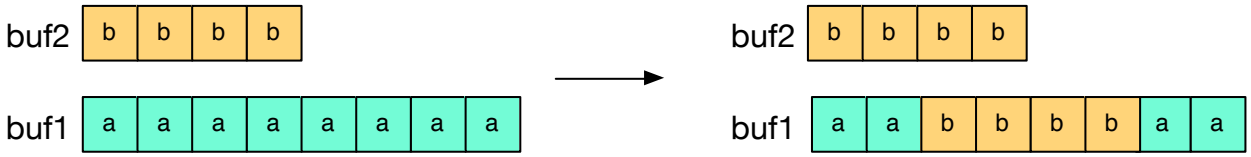
Copying memory

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

Buffer splicing!



Memory comparison ...

PennState

- 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

PennState

- 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

PennState

- 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]*
    - a garbage collector collects it *[automatic memory management]*
  - C requires you to manually manage memory
    - gives you more control, but causes headaches
    - *C has no garbage collection*

Note: this is user-defined scoping!

Garbage collection

PennState

- 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");
```

- Oddly, you don't control the deallocation of the object – it stays in memory until an invisible process behind the scenes frees.
  - This invisible process is called garbage collection, which looks through memory to see if any objects are no longer used by the program (i.e., the program no longer has any variable references that point to them), and frees them
  - This is typically done in the background by a background process
  - Pros: memory management is done for you
  - Cons: you don't have as much control over memory

Note: python also uses garbage collection.

C and malloc

PennState

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

  `malloc(size in bytes)`

  - returns a pointer to the first byte of that memory
    - `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

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

C and calloc

PennState

- Similar to malloc, but zeroes out allocated memory

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

Example malloc vs. calloc

PennState

```c
int do_calloc( void ) {

  int i;
  char *mbuf, *cbuf;
  mbuf = malloc( 5 );
  cbuf = calloc( 5, sizeof(char) );
  for (i=0; i<5; i++) {
    printf( "M[%d] = %x, C[%d] = %x\n", i,
            (unsigned char)mbuf[i], i,
            (unsigned char)cbuf[i] );
  }
  return( 0 );
}
```

```
M[0] = b8, C[0] = 0
M[1] = 9d, C[1] = 0
M[2] = 5e, C[2] = 0
M[3] = ab, C[3] = 0
M[4] = 4e, C[4] = 0
```

Deallocation

PennState

- Releases the memory pointed-to by the pointer

`free(pointer)`

  - 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

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

Dynamically allocated `structs`

PennState

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

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

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

Realloc (re-allocation)

PennState

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

  `realloc(ptr, size in bytes)`

  - 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

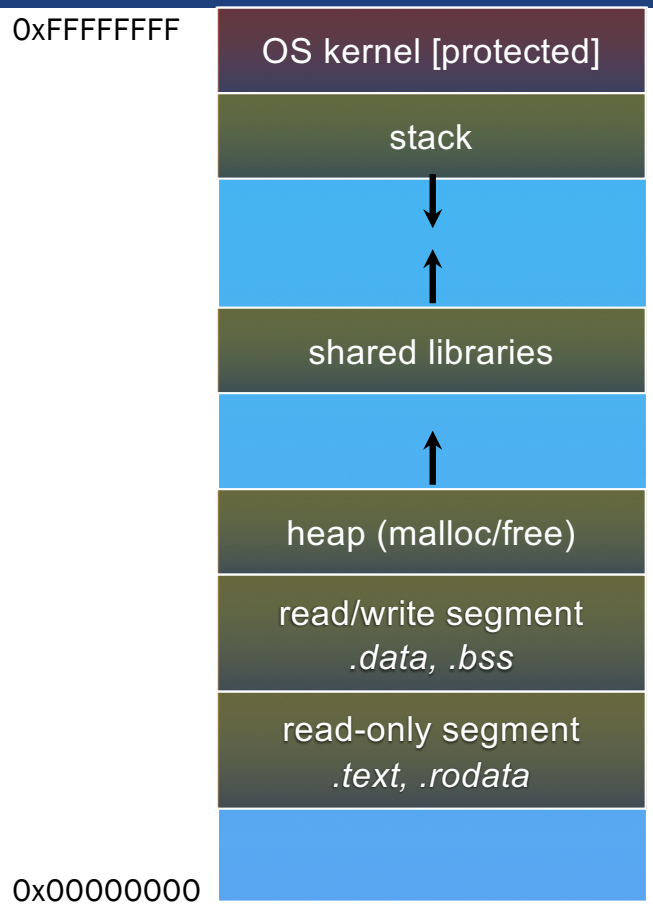
```
// allocate a 10-float array
char *buf, *rbuf;
buf = malloc(2); // allocate 2 byte array
memset(buf, 0xa, 2);
printf("buf = %x %x\n", buf[0], buf[1]);
rbuf = realloc(buf, 4); // resize to 4 bytes
printf("rbuf = %x %x %x %x\n", buf[0], buf[1],
         buf[2], buf[3]);
```

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

Heap

PennState

- 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

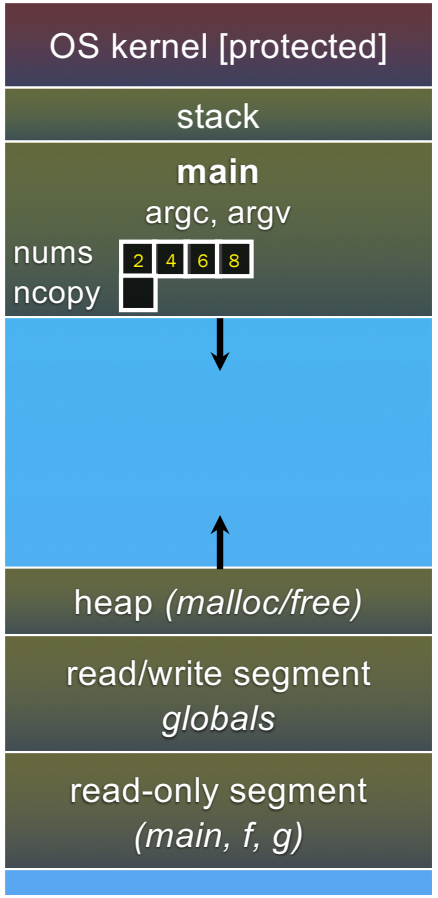


Heap + stack

PennState

```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

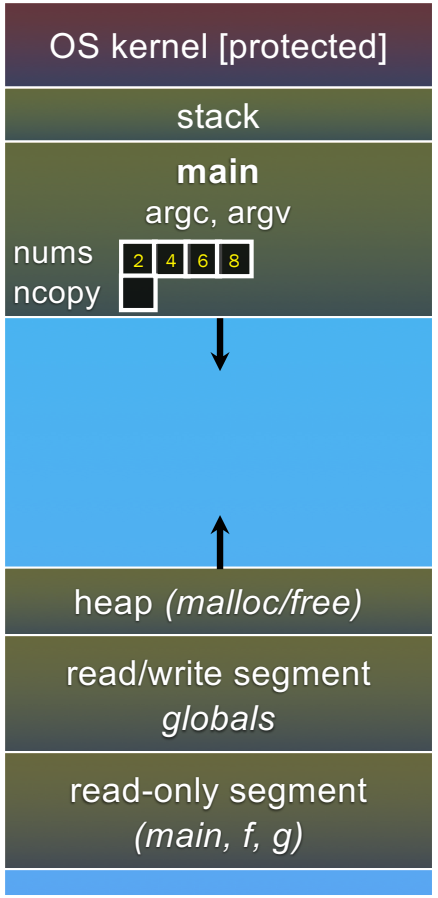


Heap + stack

PennState

```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

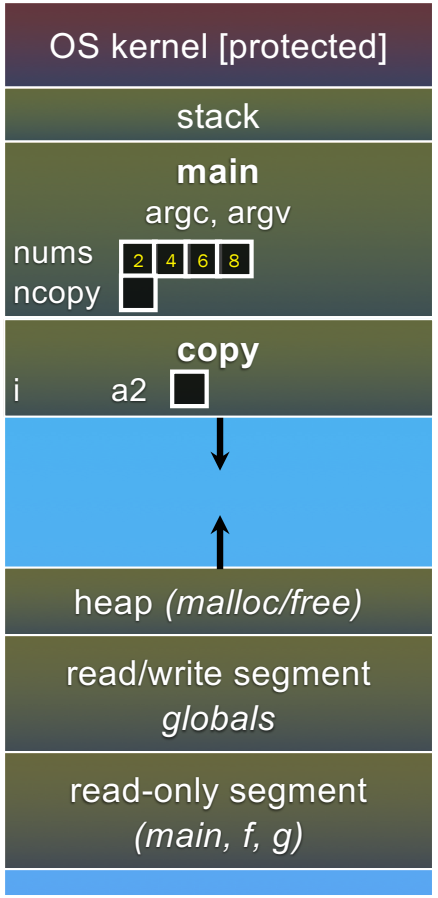


Heap + stack

PennState

```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

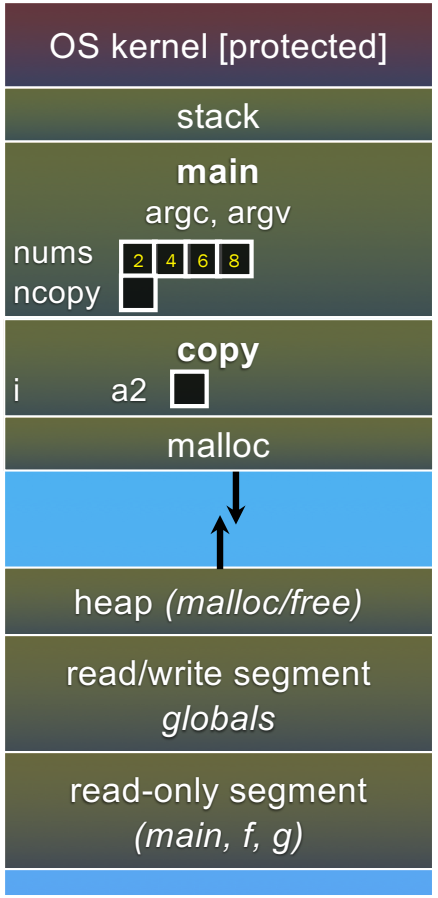


Heap + stack

PennState

```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;
}
```

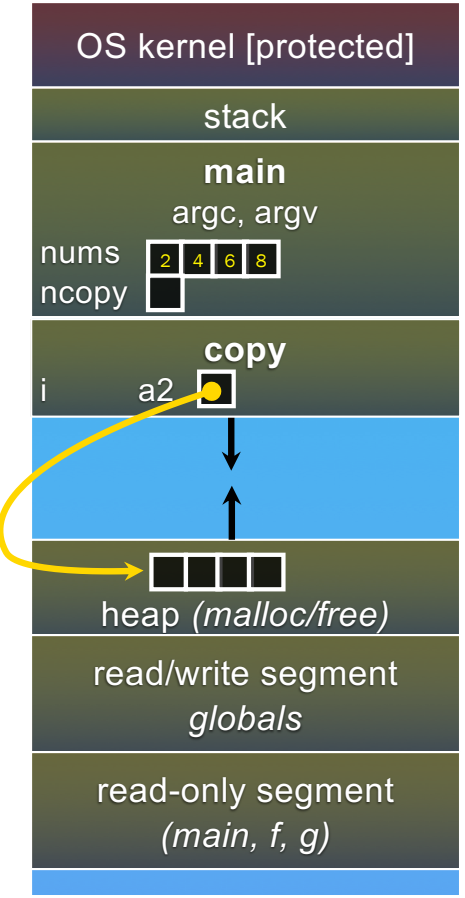
arraycopy.c



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;
}
```

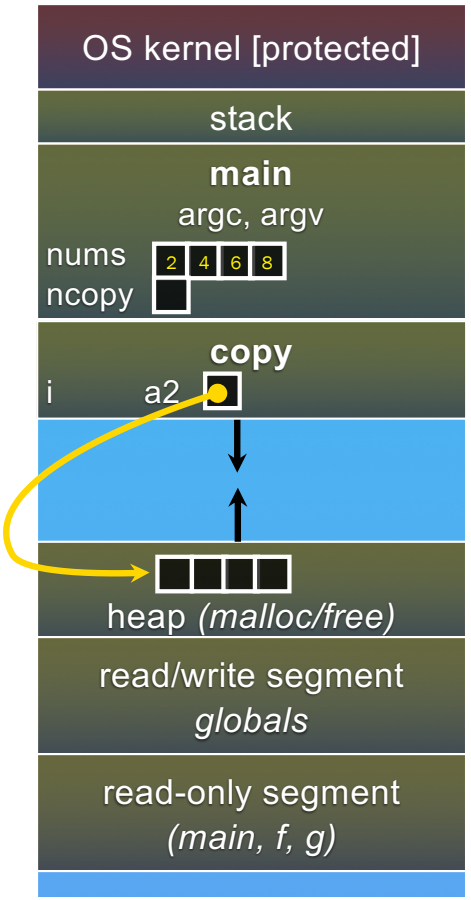
arraycopy.c



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

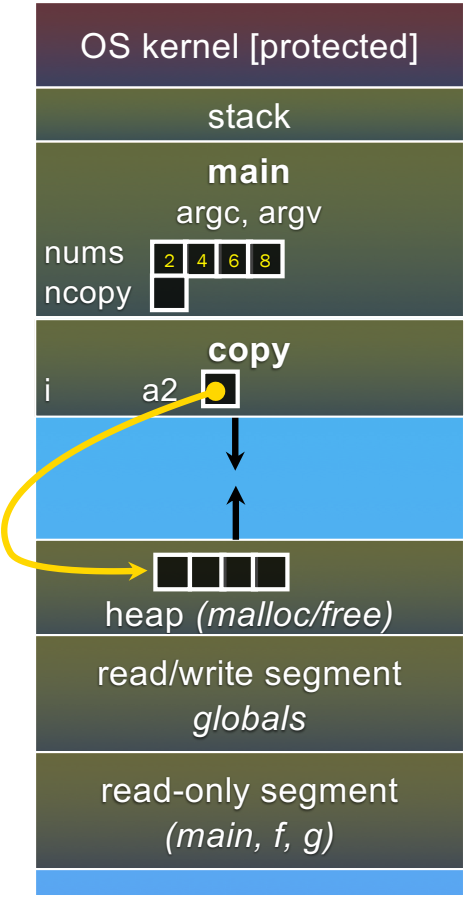


Heap + stack

PennState

```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

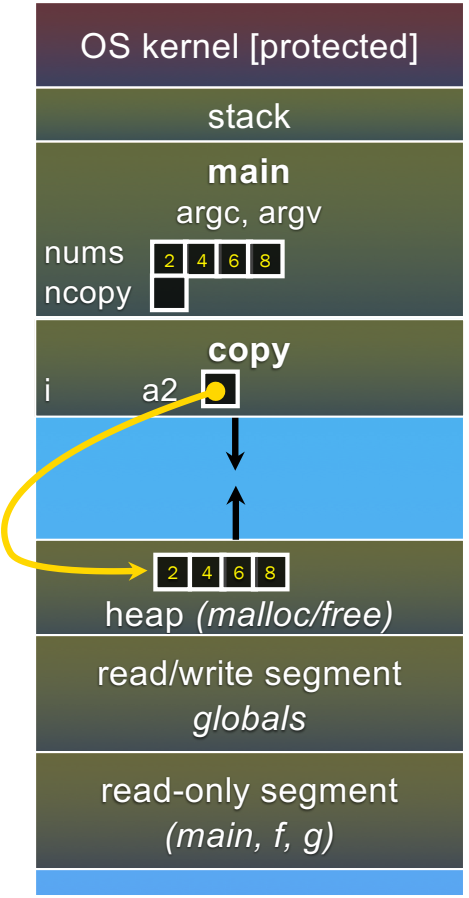


Heap + stack

PennState

```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;
}
```

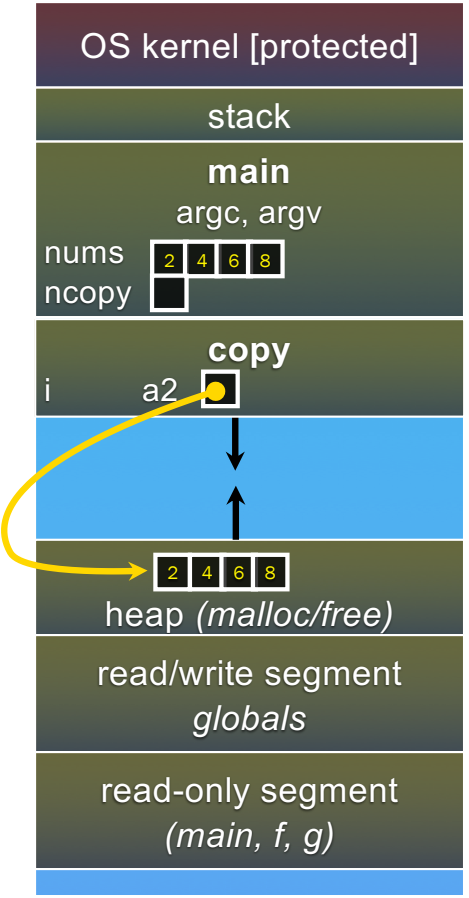
arraycopy.c



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

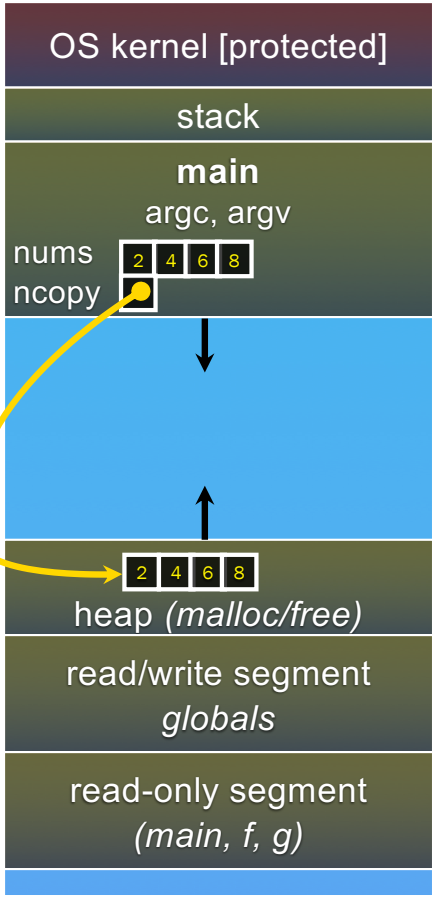


Heap + stack

PennState

```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

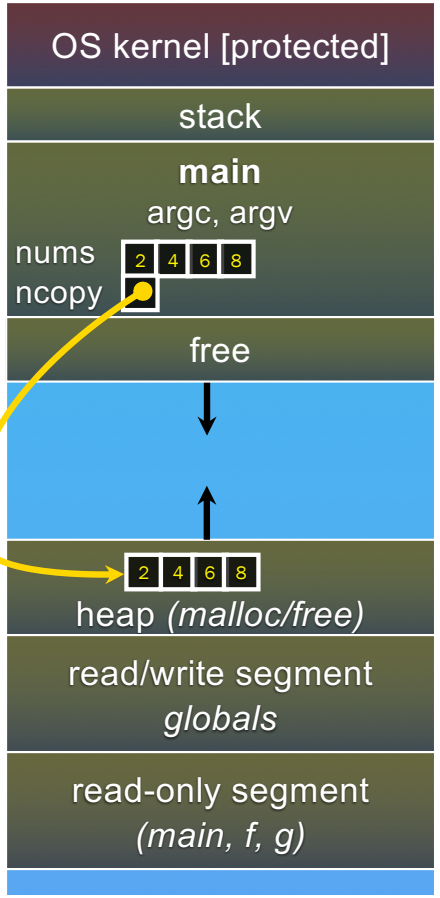


Heap + stack

PennState

```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

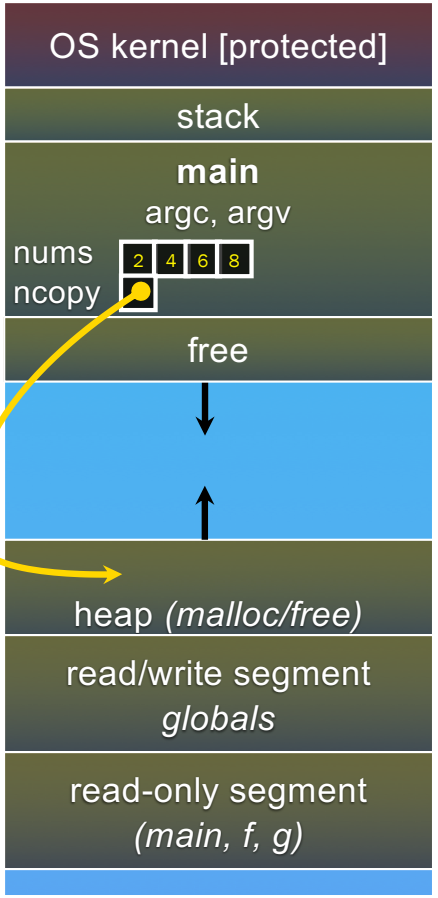


Heap + stack

PennState

```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

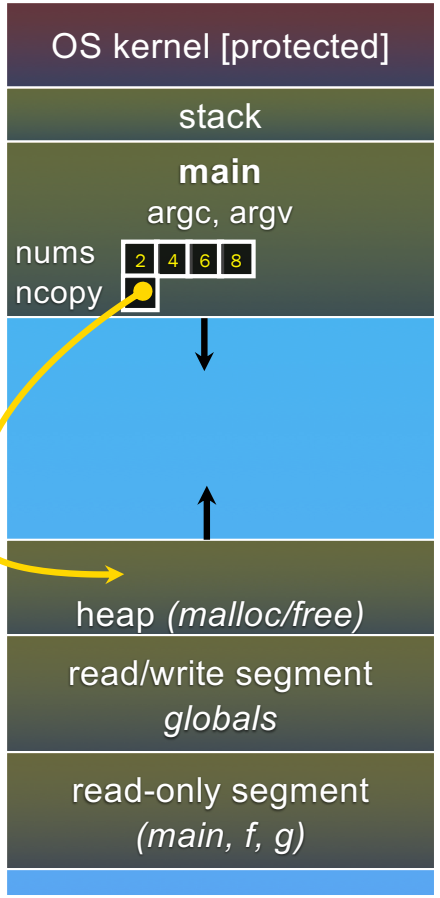


Heap + stack

PennState

```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



NULL

PennState

- 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

PennState

- 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)), *c;

  a[2] = 5;   // assign past the end of an array
  a[0] += 2;  // assume malloc zeroes out memory
  c = b+3;    // mess up your pointer arithmetic
  free(&(a[0]));  // 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

PennState

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

  ReadFileIntoBuffer(filename, filebuf);

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

Implications of a leak?

PennState

- 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()`

Look ahead: we will cover how to find leakage

Whence virtual memory?

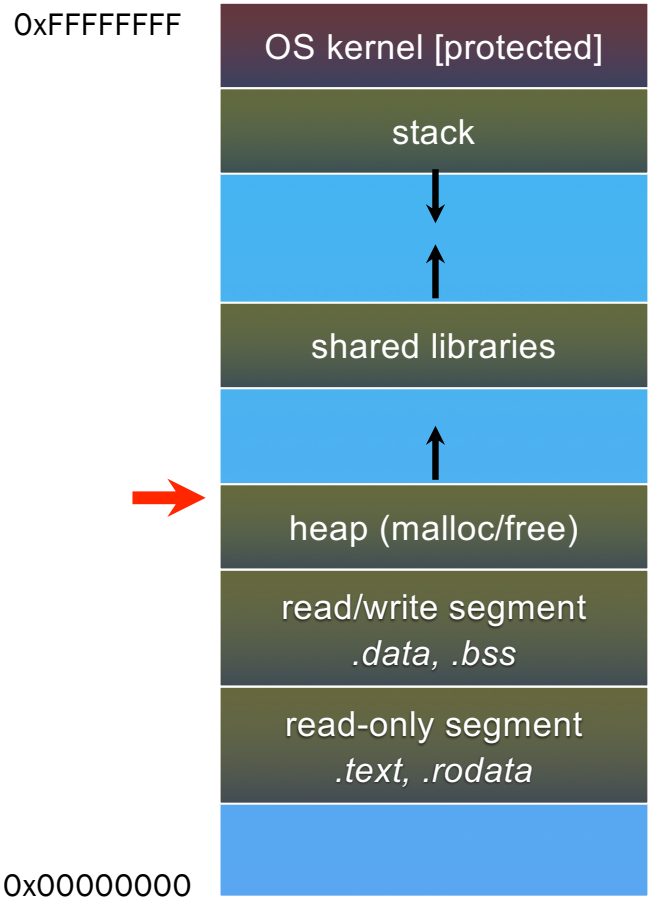
PennState

- 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

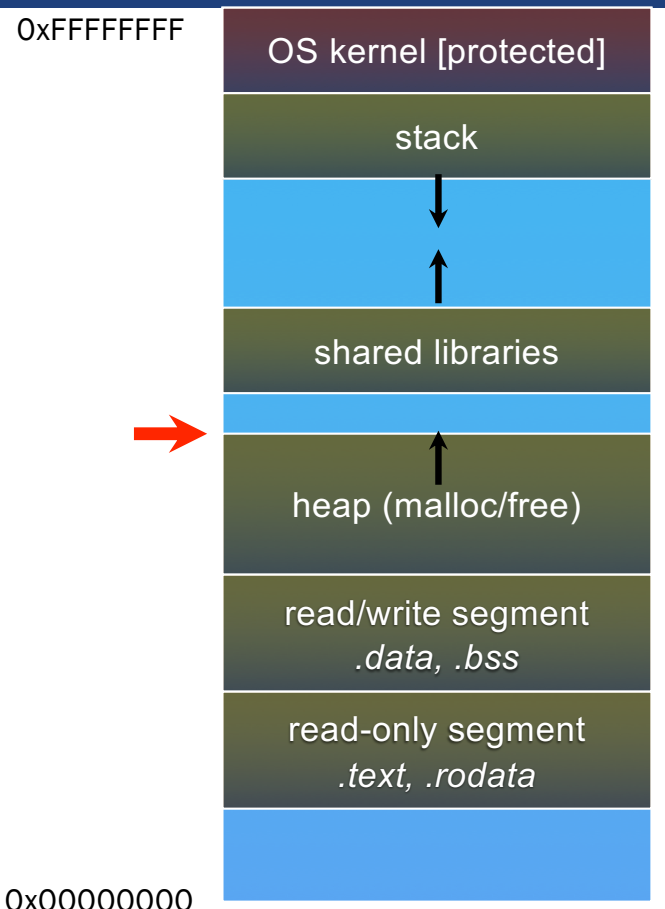
PennState

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



Program break

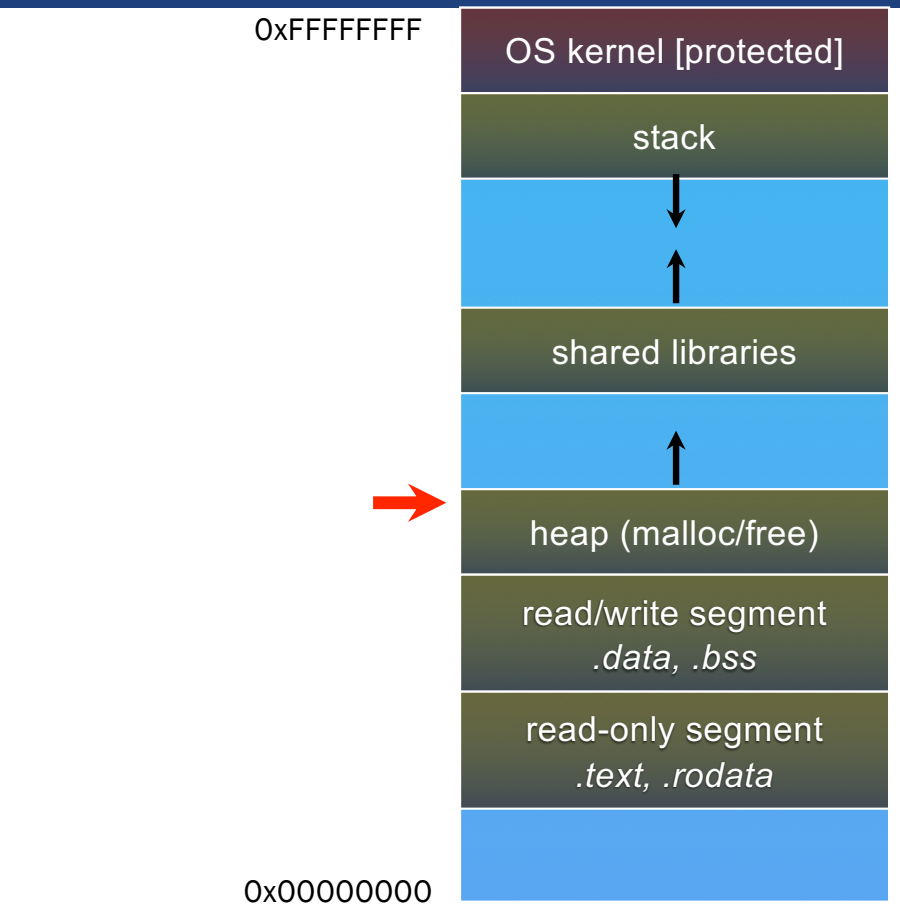
- The program break gets moved up and down as memory is allocated and deallocated from the heap.



Program break

PennState

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



Q: how?

brk() and sbrk()

PennState

- 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.

PennState

```
void *last = 0x0;

int check_memory( void ) {
        void *ptr = sbrk(0);
        printf( "The top of the heap is [%p] (%x)\n", ptr, (unsigned)(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] );
        }
        check_memory();
        return( 0 );
}
```

```
The top of the heap is [0x1415000] (0)
The top of the heap is [0x1437000] (22000)
The top of the heap is [0x1836000] (3ff000)
The top of the heap is [0x1436000] (-c00000)
```

Some observations

PennState

- The program (through `malloc`) kept getting more and more memory over time.
- It released some, but not all of memory at the end.
  - “slack” is left over

0x1415000	N/A
0x1437000	+0x0022000
0x1836000	+0x03ff000
0x1436000	-0x0400000

An alternate allocation …

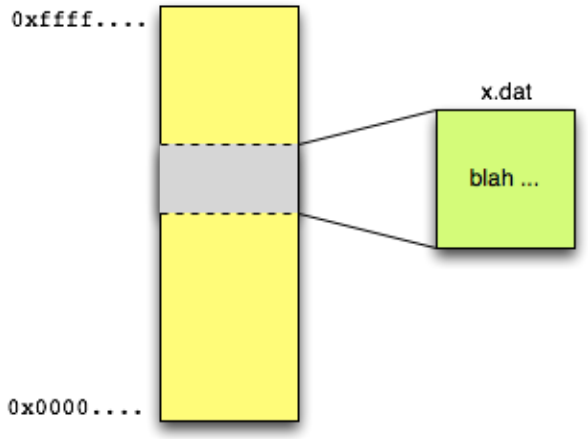
PennState

- Some implementations of malloc don't use the program break, but uses an alternate method that "*memory maps*" a region and size to allocate.
  - In essence, tells the OS that a range of memory should be available for use by the program.
  - This can be anywhere in the process (i.e., it ignores the program break)

Memory Mapping

PennState

- In its most general use, the memory map system call “maps” (overlays) a file onto physical memory.
  - A program treats the file as a random access memory.
  - Very, very fast because the OS “pages” the file content into memory in 4/8kb chunks.



Note: Memory mapping is often used to load the program code to execute.

mmap

PennState

- Where `void *mmap(addr, length, prot, flags, fd, offset);`
  - `void *addr` - address the object should be mapped to
    - If `NULL`, the OS will figure out a place to map it
  - `size_t length` - how much memory to map
  - `int prot` - protection bits that indicate what can be done to memory in that region
  - `int flags` - indicate how some features
  - `int fd` - a file descriptor (see I/O lectures)
  - `off_t` offset
- Returns pointer to region or "`(void *)-1`"

Protection bits ...

PennState

- The `prot` argument describes the desired memory protection of the mapping.
  - It is either `PROT_NONE` or the bit-wise OR of one or more of the following flags:
  - `PROT_EXEC` - Pages may be executed.
  - `PROT_READ` - Pages may be read.
  - `PROT_WRITE` - Pages may be written.
  - `PROT_NONE` - Pages may not be accessed.

These bits allow the program to police the use of mapped memory regions.

Flags

PennState

- Flags indicate what kind of mapping we are doing:
  - MAP_SHARED - Updates to the mapping are visible to other processes that map this file, and are carried through to the underlying file.
  - MAP_PRIVATE - Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file.
- And one more option
  - MAP_ANONYMOUS - The mapping is not backed by any file; its contents are initialized to zero.

munmap

PennState

```
void *munmap(addr, length);
```

- Where
  - void *addr - address the map that should be released as obtained from prior mmap
  - size_t length - how much memory to release

Note: modifications to mapped file will be written to disk

Mapping example

PennState

```
int mymap( char val ) {

        // Local variables
        int fh, i;
        char *ptr;

        // Open the file and "map" 20 bytes of it to memory
        if ( (fh = open( "mmap.dat", O_CREAT|O_RDWR )) == 0 ) {
                printf( "Error in open [%s], aborting\n", strerror(errno) );
                return( -1 );
        }
        if ( (ptr = mmap( NULL, 20, PROT_READ|PROT_WRITE, MAP_SHARED, fh, 0 )) == (void *)-1 ) {
                printf( "Error in map [%s], aborting\n", strerror(errno) );
                return( -1 );
        }

        // Add the values
        for (i=0; i<20; i++) {
                ptr[i] = val; // Replicate the value into the file
        }

        // "free" the mapped memory, close the file
        munmap( ptr, 20 );
        close( fh );
        return( 0 );
}
```

Mapping example

PennState

```
int mymap( char val ) {

        // Local variables
        int fh, i;
        char *ptr;

        // Open the file and "map" 20 bytes of it to memory
```

```
BEFORE:
(gdb) x /20xb ptr
0x7ffff7ff6000: 0x7f    0x45    0x4c    0x46    0x02    0x01    0x01    0x00
0x7ffff7ff6008: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffff7ff6010: 0x02    0x00    0x3e    0x00

AFTER:
(gdb) x /20xb ptr
0x7ffff7ff6000: 0x58    0x58    0x58    0x58    0x58    0x58    0x58    0x58
0x7ffff7ff6008: 0x58    0x58    0x58    0x58    0x58    0x58    0x58    0x58
0x7ffff7ff6010: 0x58    0x58    0x58    0x58
```

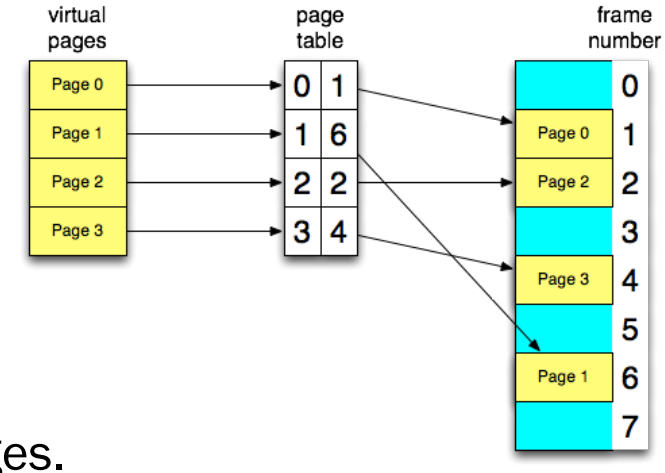
```
        // "free" the mapped memory, close the file
        munmap( ptr, 20 );
        close( fh );
        return( 0 );
}
```

An alternate allocation ...

PennState

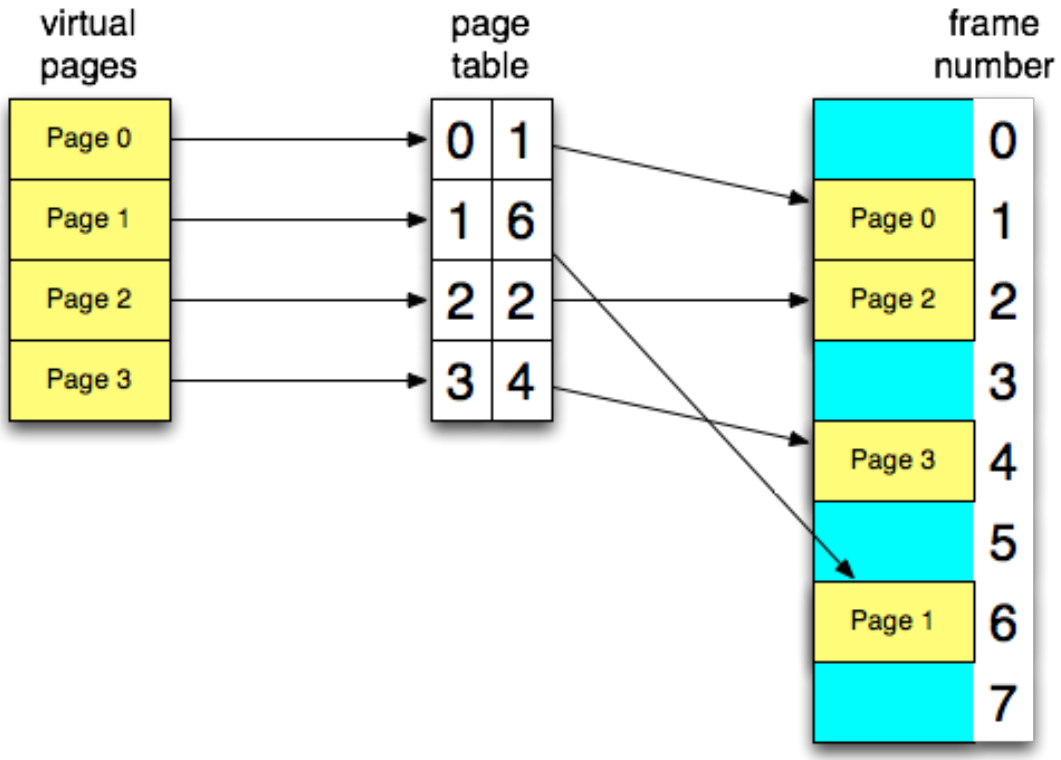
- Some implementations of malloc don't use the program break, but uses an alternate method that "*memory maps*" a region and size to allocate blocks of memory from.
  - In essence, memory maos tells the OS that a range of memory should be available for use by the program.
  - This can be anywhere in the process (i.e., it ignores the program break)
  - Implementation of malloc and free use memory mapping instead ...
    - `mmap` instead of `brk()`/`sbrk()`
    - `munmap` instead of `brk()`/`sbrk()`
  - It is a matter of design which variant we use

OS Paging



- Programs have a virtual address space (say 1 MB).
  - OS fetches data from either physical memory or disk.
  - Done by a mechanism called (demand) paging.
- Divide the virtual address space into units called “virtual pages” each of which is of a fixed size (usually 4K or 8K).
  - For example, 1M virtual address space has 256 4K pages.
- Physical addresses partitioned into “physical pages”/“frames”.
  - For example, 1M physical memory could have 256 4K-sized pages.
- OS tracks virtual pages in physical memory
  - Maintained in a data structure called “page-table”
  - “Page-tables” map Virtual-to-Physical addresses.

An example Page Table



And more processes ...