APS105
Winter 2012

Jonathan Deber
jdeber -at- cs -dot- toronto -dot- edu

Lecture 24
March 16, 2012

Today

- (Even More) Dynamic Memory Allocation

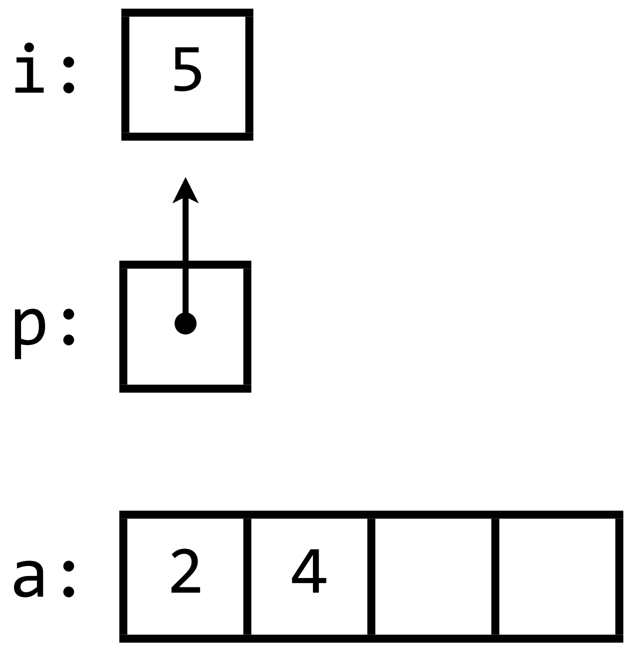
Dynamic Memory Allocation

`malloc()` and friends

Automatic Allocation

- We've been doing this since the first week
- Variables have a name and an address

```
int i;
i = 5;

int *p = &i;

int a[4];
a[0] = 2;
a[1] = 4;
```



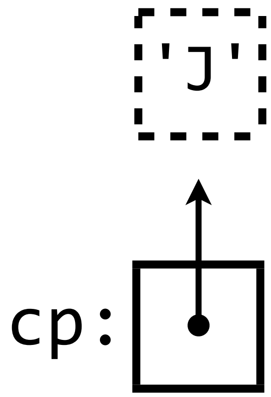
`malloc()`

```
void *malloc(~~size_t~~ size);
```

for now think of it as `int`

- "Memory Allocator"
- `#include <stdlib.h>`
- You ask `malloc()` for some memory, it finds some, and then gives it to you
- `void *` pointer ("generic" pointer)
- Different pool of memory (called the heap)

Dotted border means
“dynamically allocated”



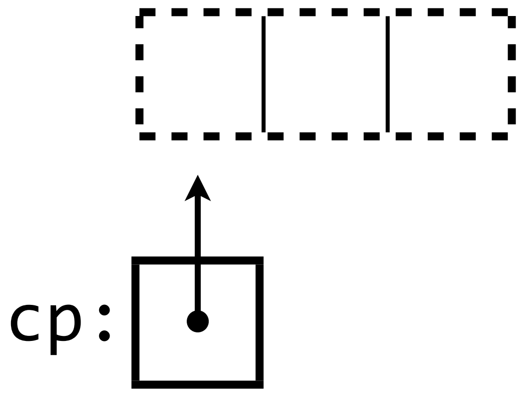
$$\overbrace{\texttt{malloc(1)}}^{\texttt{void *}}$$

```
char *cp = malloc(1);

  char c = malloc(1);
```

Wrong

```
     *cp = 'J';
```

```
int i = 5;
int j = &i;
```

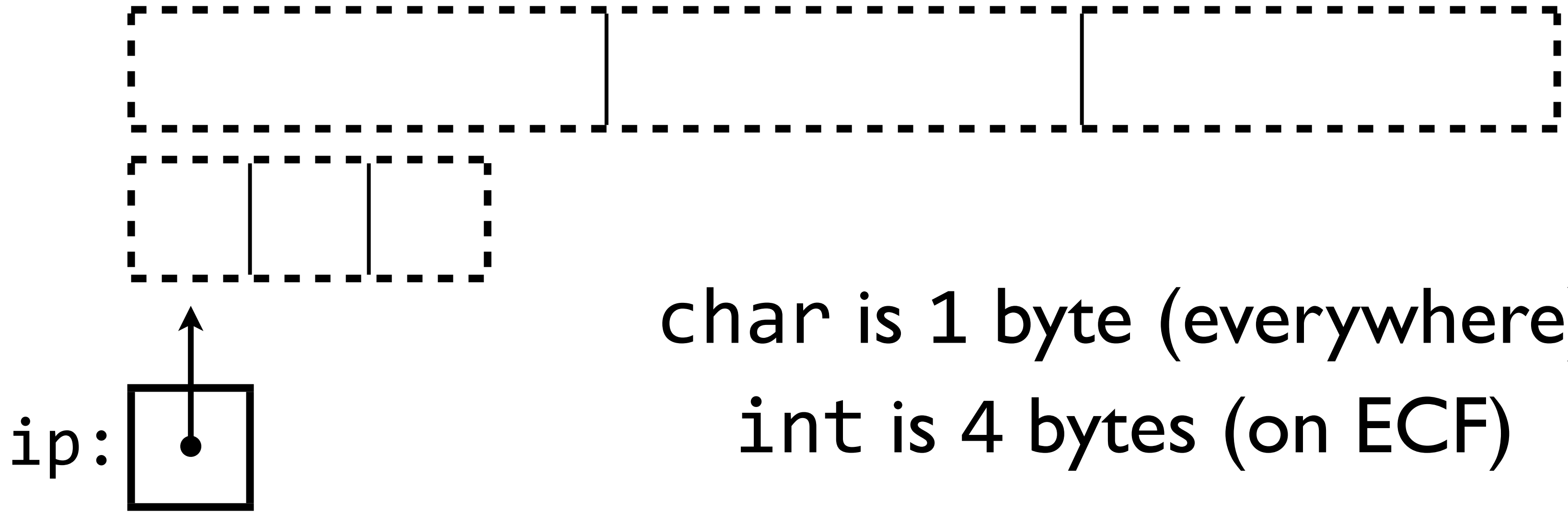


number of bytes

```
char *cp = malloc(3);
```

number of elements   size of each element

```
int *ip = malloc(3 * 4);
```



`char` is 1 byte (everywhere)

`int` is 4 bytes (on ECF)

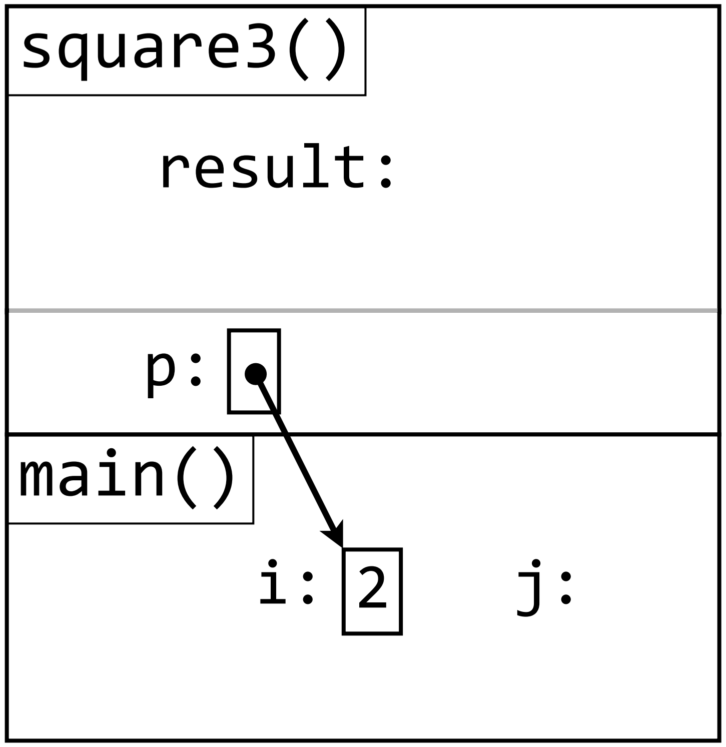
`sizeof()` and `malloc()`

```
int *ip = malloc(3 * 4);

int *ip = malloc(3 * sizeof(int));
```

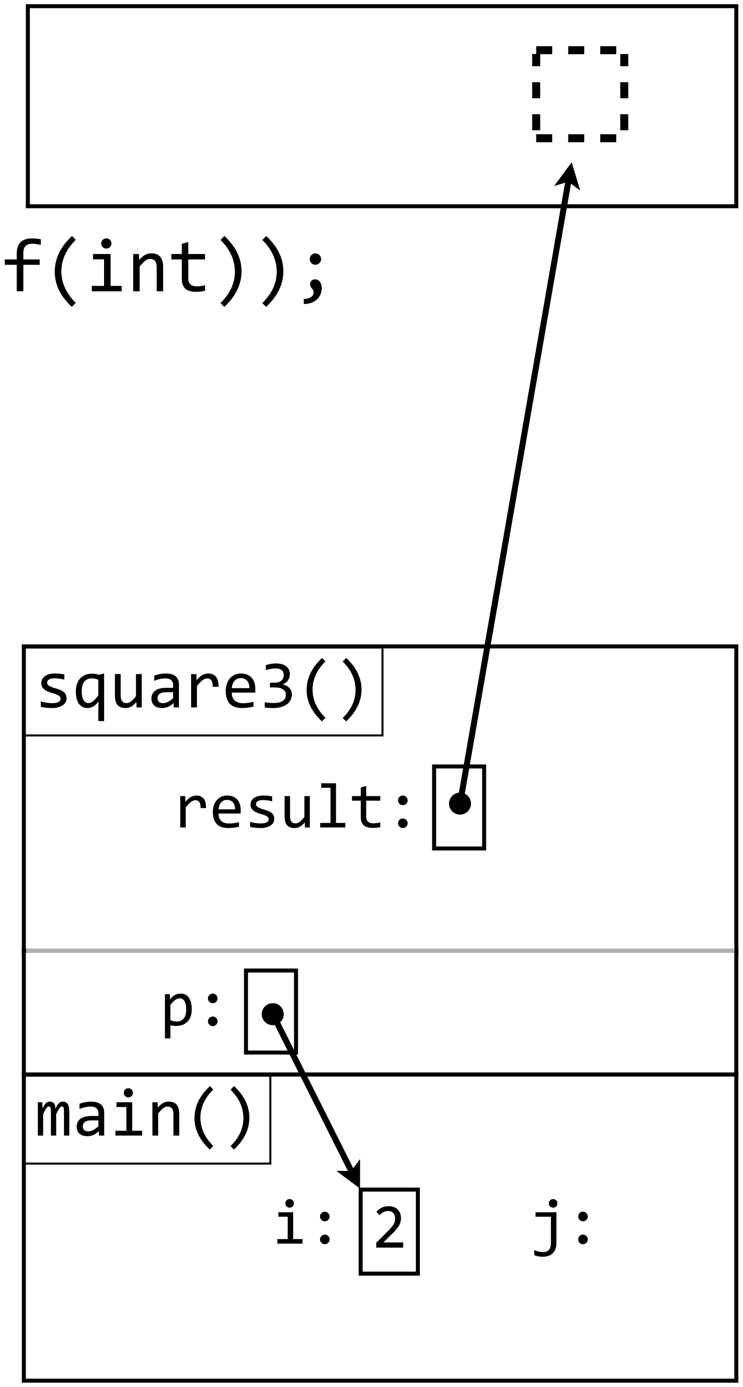
You should always use `sizeof()`

```
char *cp = malloc(3);

char *cp = malloc(3 * sizeof(char));
```

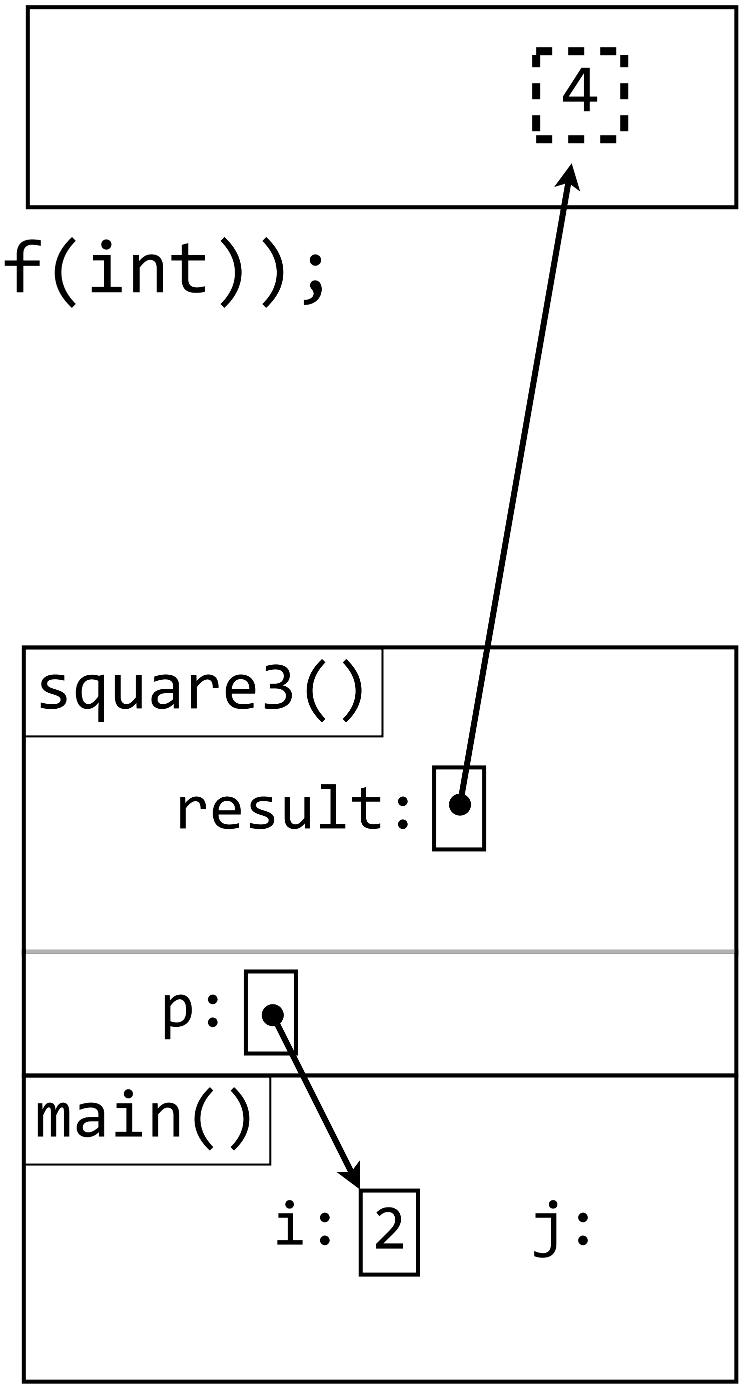
```
int *square3(int *p)
{
  int *result = malloc(sizeof(int));
  *result = *p * *p;
  return result;
}


int main (void)
{
  int i = 2;
  int *j = square3(&i);
  printf("%d", *j);

  return 0;
}
```



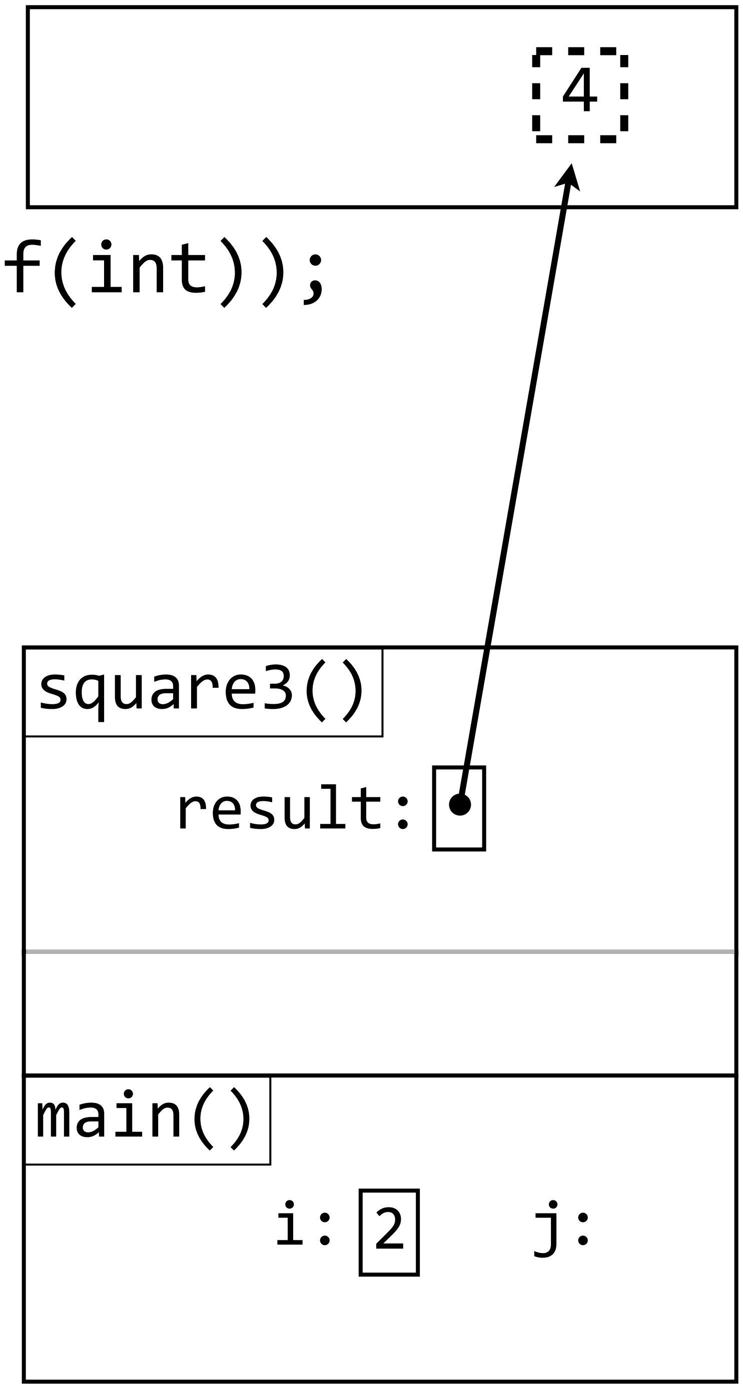
```
int *square3(int *p)
{
  int *result = malloc(sizeof(int));
  *result = *p * *p;
  return result;
}


int main (void)
{
  int i = 2;
  int *j = square3(&i);
  printf("%d", *j);

  return 0;
}
```



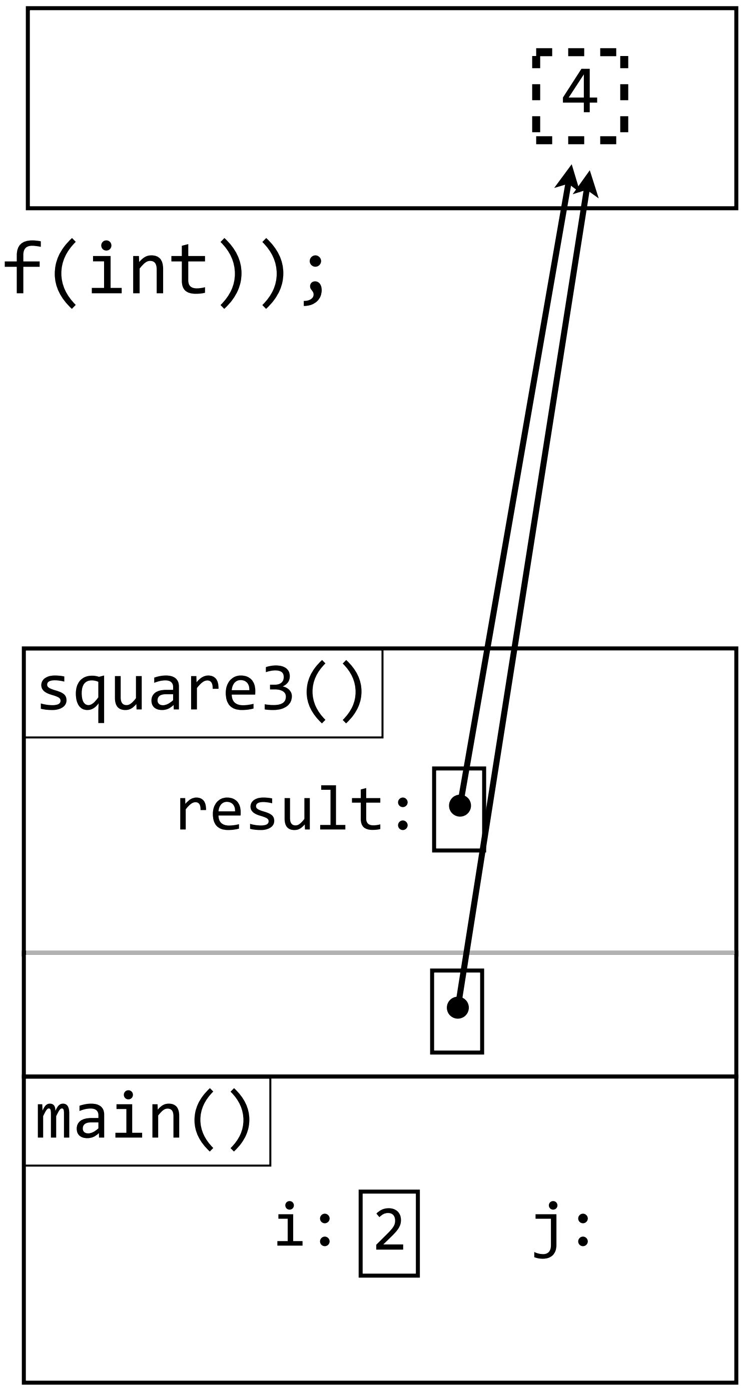
```
int *square3(int *p)
{
  int *result = malloc(sizeof(int));
  *result = *p * *p;
  return result;
}


int main (void)
{
  int i = 2;
  int *j = square3(&i);
  printf("%d", *j);

  return 0;
}
```



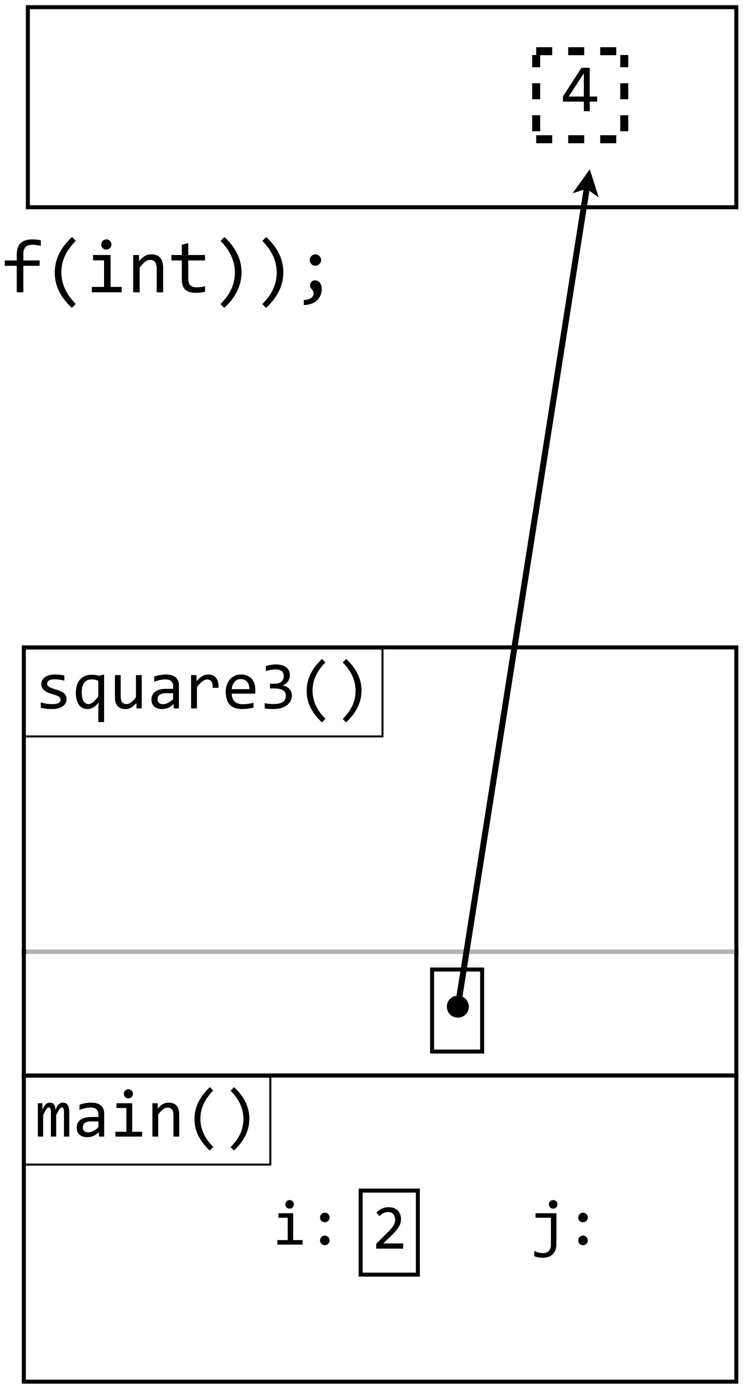
```
int *square3(int *p)
{
  int *result = malloc(sizeof(int));
  *result = *p * *p;
  return result;
}


int main (void)
{
  int i = 2;
  int *j = square3(&i);
  printf("%d", *j);

  return 0;
}
```



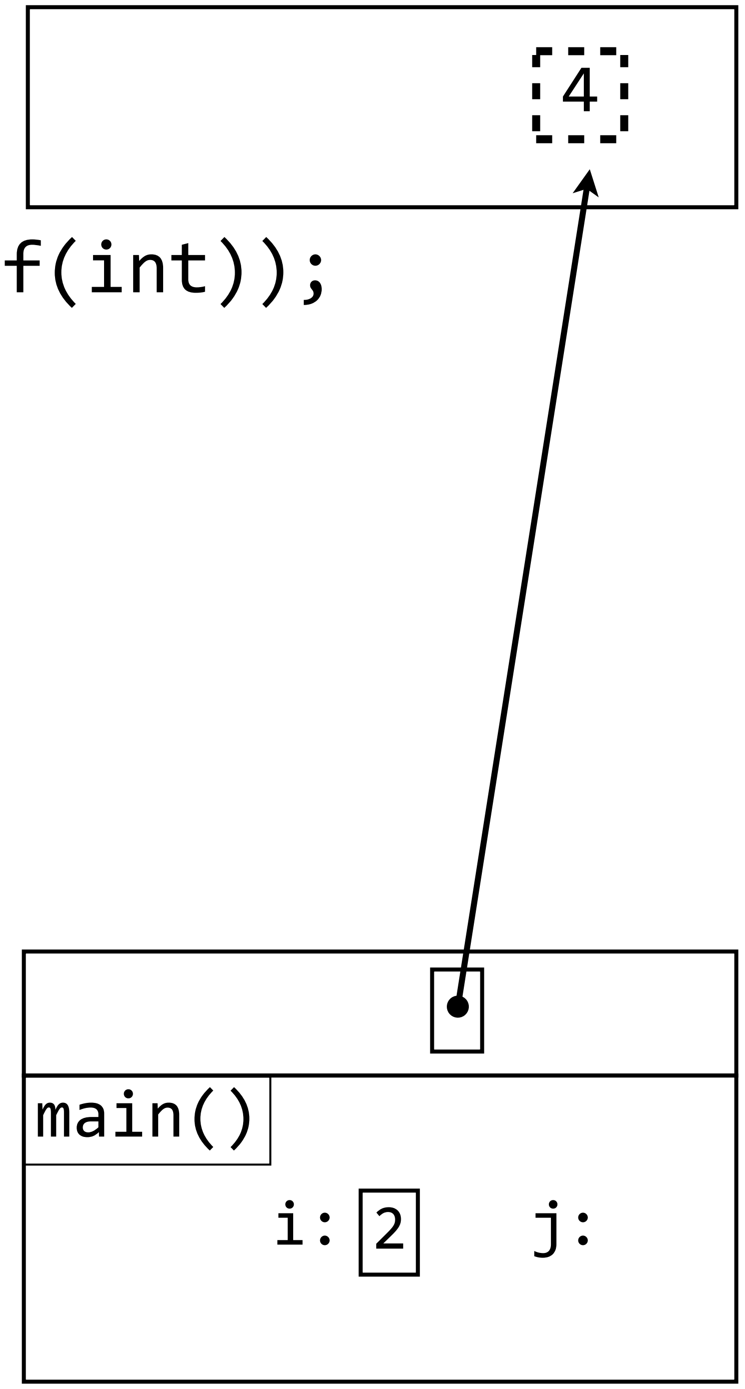
```c
int *square3(int *p)
{
  int *result = malloc(sizeof(int));
  *result = *p * *p;
  return result;
}


int main (void)
{
  int i = 2;
  int *j = square3(&i);
  printf("%d", *j);

  return 0;
}
```



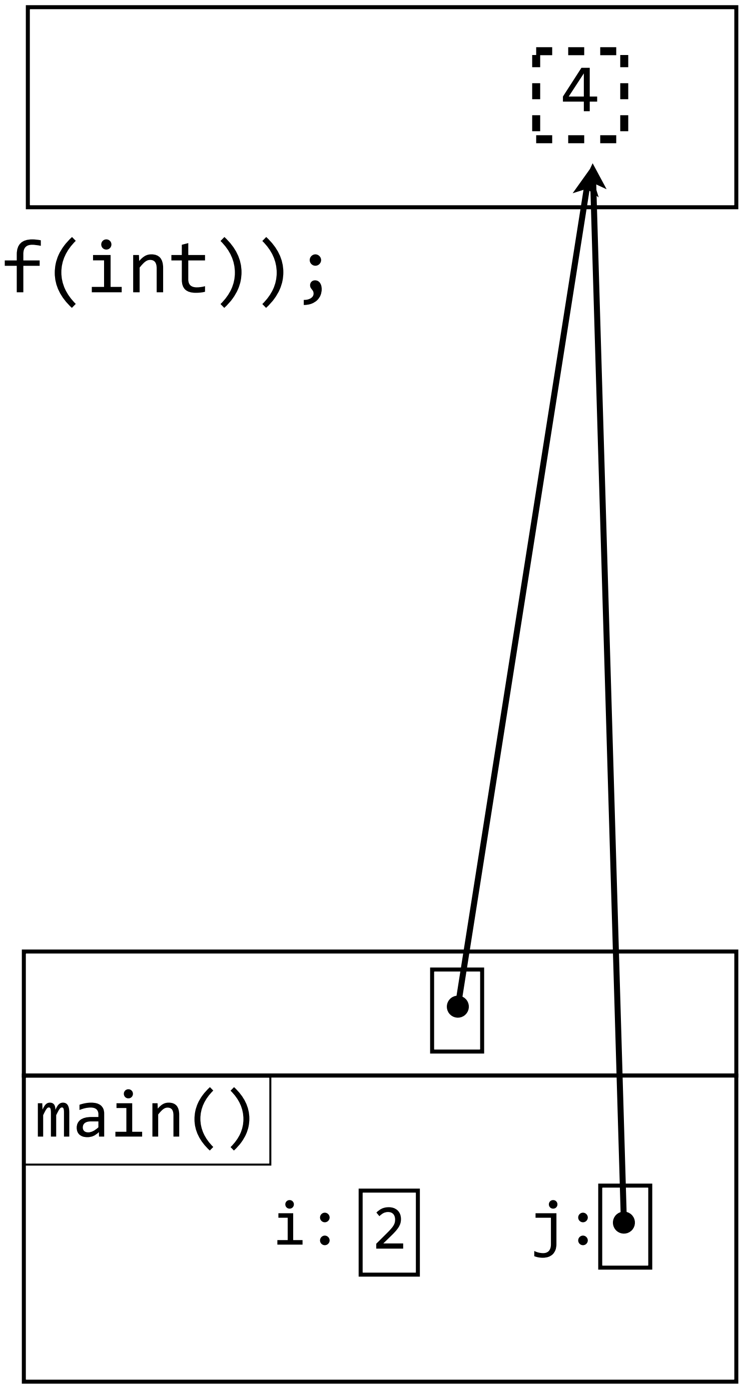
```c
int *square3(int *p)
{
  int *result = malloc(sizeof(int));
  *result = *p * *p;
  return result;
}


int main (void)
{
  int i = 2;
  int *j = square3(&i);
  printf("%d", *j);

  return 0;
}
```



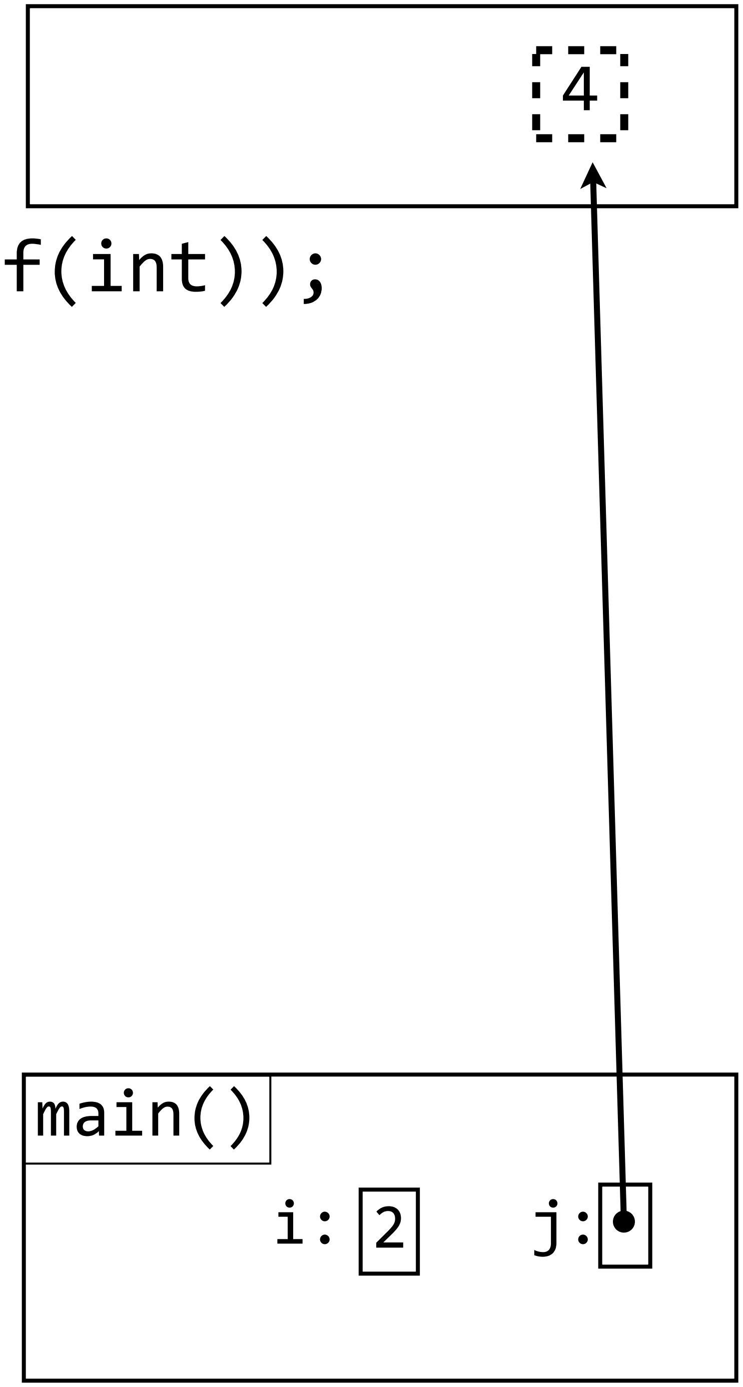
```
int *square3(int *p)
{
  int *result = malloc(sizeof(int));
  *result = *p * *p;
  return result;
}


int main (void)
{
  int i = 2;
  int *j = square3(&i);
  printf("%d", *j);

  return 0;
}
```



```
int *square3(int *p)
{
  int *result = malloc(sizeof(int));
  *result = *p * *p;
  return result;
}


int main (void)
{
  int i = 2;
  int *j = square3(&i);
  printf("%d", *j);

  return 0;
}
```

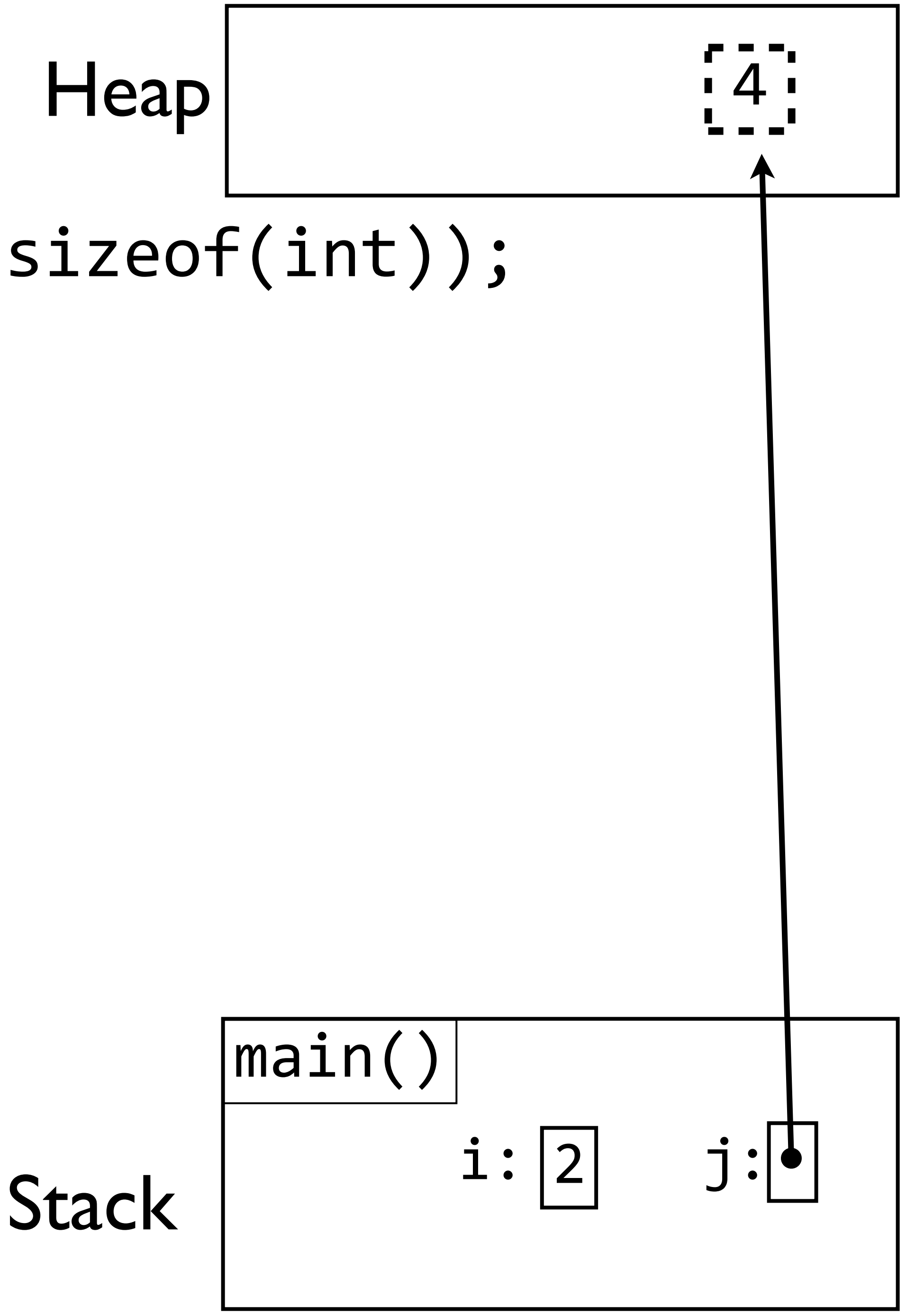


```
int *square3(int *p)
{
  int *result = malloc(sizeof(int));
  *result = *p * *p;
  return result;
}


int main (void)
{
  int i = 2;
  int *j = square3(&i);
  printf("%d", *j);

  return 0;
}
```



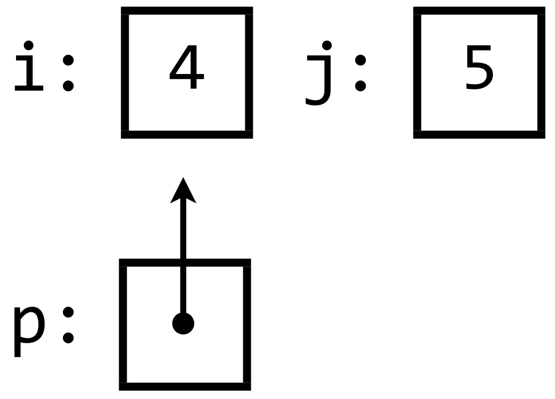
```
int *square3(int *p)
{
  int *result = malloc(sizeof(int));
  *result = *p * *p;
  return result;
}


int main (void)
{
  int i = 2;
  int *j = square3(&i);
  printf("%d", *j);

  return 0;
}
```

4



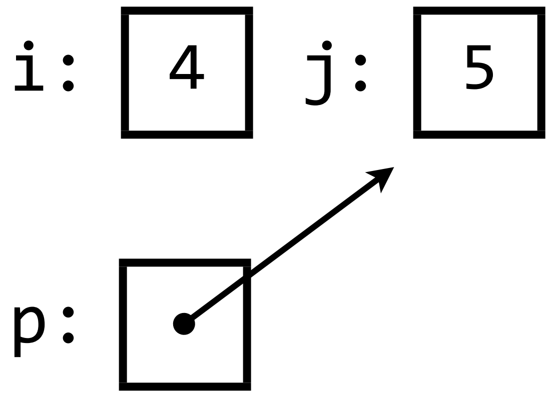
Cleaning Up

```
int i = 4;
int j = 5;
int *p = &i;
```



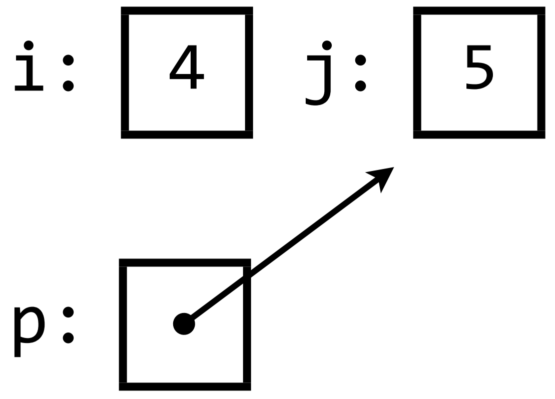
Cleaning Up

```
int i = 4;
int j = 5;
int *p = &i;

p = &j;
```

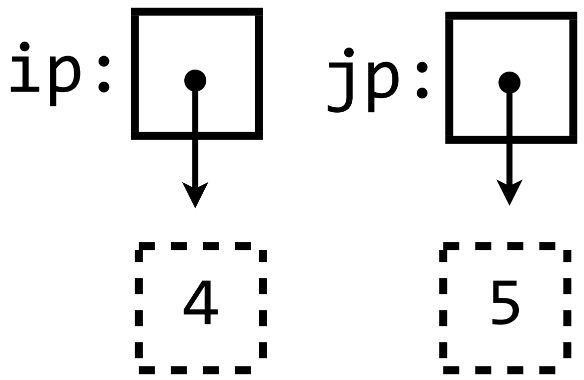


Cleaning Up

```
int i = 4;
int j = 5;
int *p = &i;

p = &j;
```

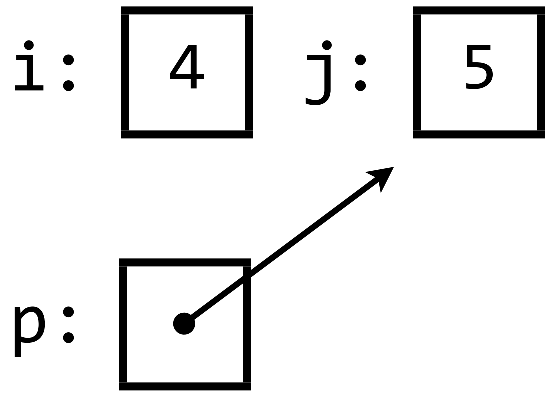


```
int *ip = malloc(sizeof(int));
int *jp = malloc(sizeof(int));
*ip = 4;
*jp = 5;
```

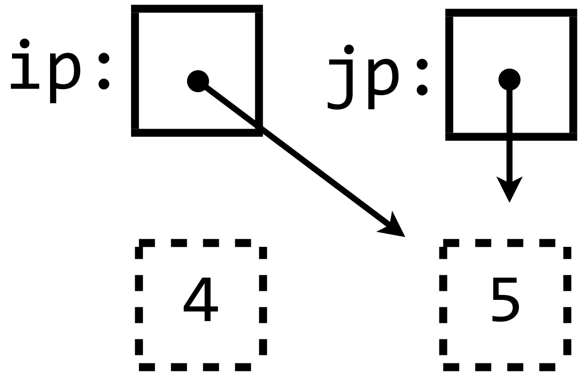


Cleaning Up

```
int i = 4;
int j = 5;
int *p = &i;

p = &j;
```

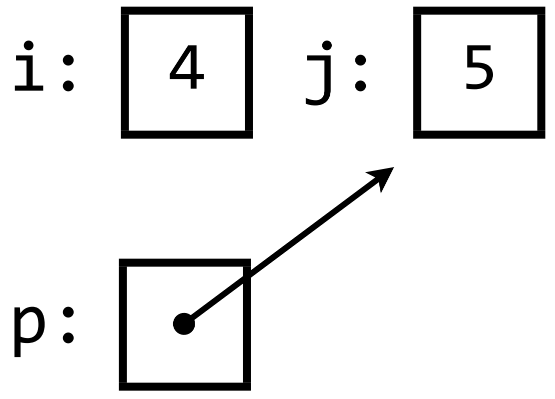


```
int *ip = malloc(sizeof(int));
int *jp = malloc(sizeof(int));
*ip = 4;
*jp = 5;

 ip = jp;
```



Cleaning Up

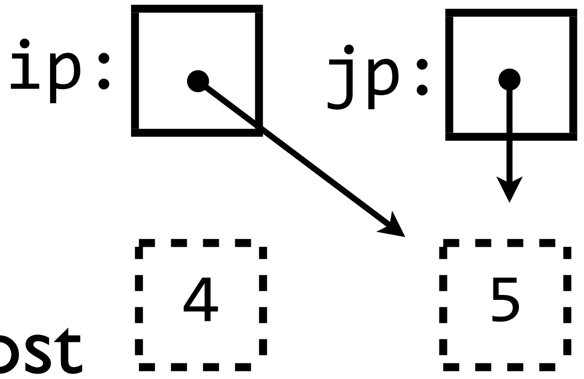
```
int i = 4;
int j = 5;
int *p = &i;
p = &j;
```

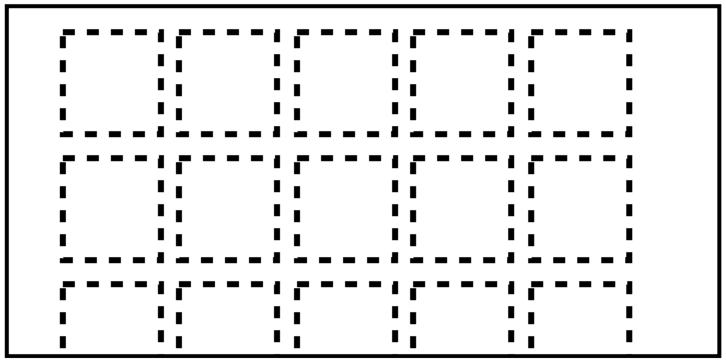


```
int *ip = malloc(sizeof(int));
int *jp = malloc(sizeof(int));
*ip = 4;
*jp = 5;
 ip = jp;
```

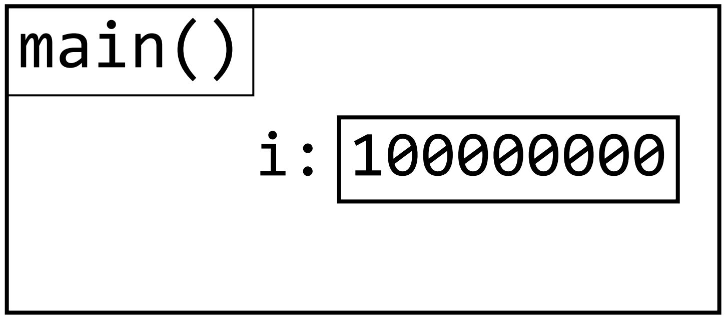
This value is lost

This bit of memory is lost





```
int main (void)
{
  int n = 1000000000;
  for (int i = 0; i < n; i++)
  {
    malloc(sizeof(int));
  }

  return 0;
}
```



Memory Leaks

- A memory leak is when you lose memory
- Possible because `malloc()`ed memory does not have a name, only an address

```
int *ip = malloc(sizeof(int));
int *jp = malloc(sizeof(int));
ip = jp;
```

This memory was leaked

- Causes your program to use more and more memory
- In C, it's your responsibility to prevent leaks
- Entirely manual process

`free()`

```
void free(void *p);
```

- Returns memory to the pool
  - It keeps track of the sizes of allocated blocks
- Any memory you get from `malloc()` *must* be cleaned up using `free()`

```
int *p = malloc(sizeof(int));
...
free(p);
```

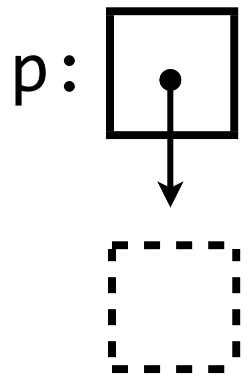
Three `free()` bugs

- "Use after `free()`"
- "Double `free()`"
- `free()`ing non-`malloc()`ed memory

“Use After Free”

- Once memory has been `free()`ed, you can’t use it again

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

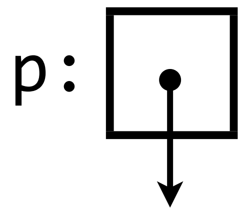


“Use After Free”

- Once memory has been `free()`ed, you can’t use it again

```
int *p = malloc(sizeof(int));
...
free(p);
```

This doesn’t change p itself

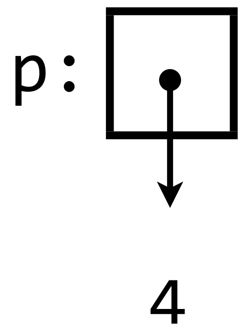


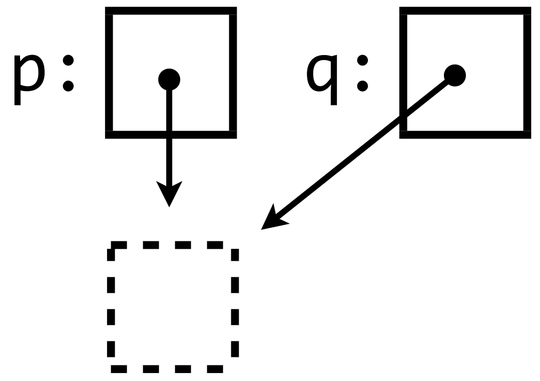
“Use After Free”

- Once memory has been `free()`ed, you can’t use it again

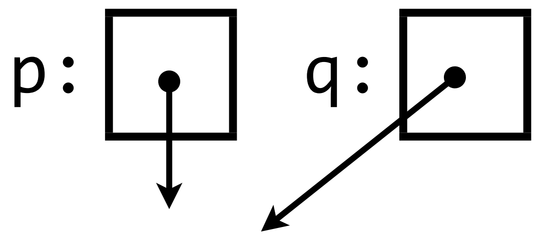
```
int *p = malloc(sizeof(int));
...
free(p);
```
This doesn’t change p itself

```
*p = 4;
```
Wrong

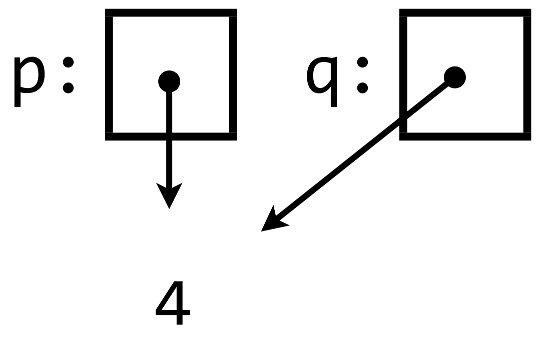




```
int *p = malloc(sizeof(int));
int *q = p;
```



```
int *p = malloc(sizeof(int));
int *q = p;
...
free(p);

// Make sure not to use p anymore!
```



```
int *p = malloc(sizeof(int));
int *q = p;
...
free(p);

// Make sure not to use p anymore!
*q = 4;
```

Wrong

“Double Free”

- You can only `free()` memory once

```
int *p = malloc(sizeof(int));
...
free(p);
free(p);
```

Wrong

"Double Free"

- You can only `free()` memory once

```
int *p = malloc(sizeof(int));
int *q = p;
...

free(p);
free(p);   Wrong
free(q);   Wrong
```

Non-`malloc()`ed Memory

- You can only `free()` memory obtained from `malloc()`

```
int a[] = {9, 8, 7, 6};
free(a);
```
Wrong

```
int i = 10;
int *p = &i;
free(p);
```
Wrong

`malloc()` Summary

- `malloc()` gives you memory from a separate pool of memory called the heap
- This memory exists outside of functions
- Returns a generic `void *` pointer, must be stored in a pointer variable
- Parameter is the number of bytes to allocate (`sizeof()` is your friend)

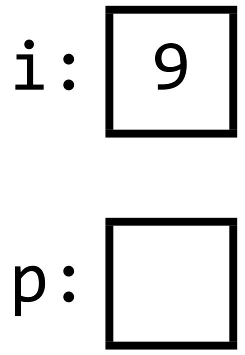
`free()` Summary

- All memory obtained from `malloc()` must eventually get passed to `free()`
- You can't use that memory after it's been `free()`ed
- You can't call `free()` more than once on the same chunk of memory
- You can't `free()` memory you didn't get from `malloc()`
- C being C, all of the above will compile, but are undefined behaviours, and may crash horribly

NULL

- A special pointer value that doesn't point at anything
- Not the same as being uninitialized!
- Need to `#include` one of the common libraries (e.g., `stdio.h` or `stdlib.h`)

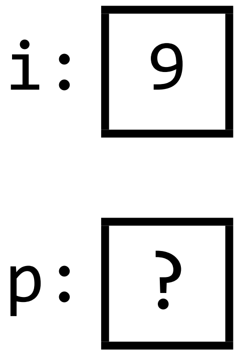
```
int i = 9;
int *p;
```
(p is uninitialized)



NULL

- A special pointer value that doesn't point at anything
- Not the same as being uninitialized!
- Need to `#include` one of the common libraries (e.g., `stdio.h` or `stdlib.h`)

```
int i = 9;
int *p;     (p is uninitialized)
```

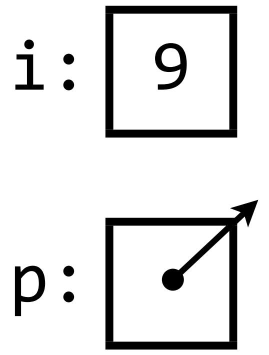


NULL

- A special pointer value that doesn't point at anything
- Not the same as being uninitialized!
- Need to `#include` one of the common libraries (e.g., `stdio.h` or `stdlib.h`)

```
int i = 9;
int *p;
```

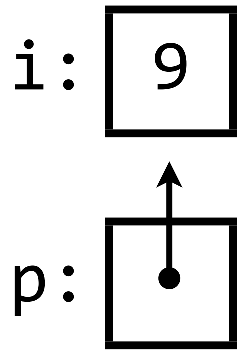
(p is uninitialized)



NULL

- A special pointer value that doesn't point at anything
- Not the same as being uninitialized!
- Need to `#include` one of the common libraries (e.g., `stdio.h` or `stdlib.h`)

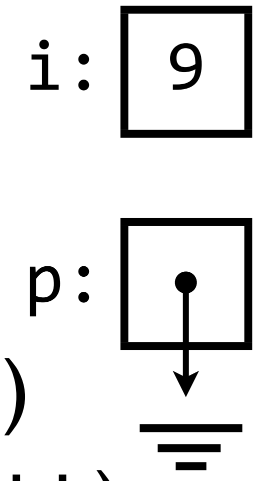
```
int i = 9;
int *p;      (p is uninitialized)

p = &i;      (p has the value &i)
```



NULL

- A special pointer value that doesn't point at anything
- Not the same as being uninitialized!
- Need to `#include` one of the common libraries (e.g., `stdio.h` or `stdlib.h`)

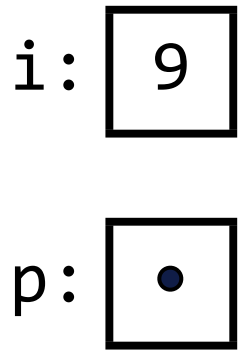
```
int i = 9;
int *p;     (p is uninitialized)

p = &i;     (p has the value &i)
p = NULL;   (p has the value NULL)
```



NULL

- A special pointer value that doesn't point at anything
- Not the same as being uninitialized!
- Need to `#include` one of the common libraries (e.g., `stdio.h` or `stdlib.h`)

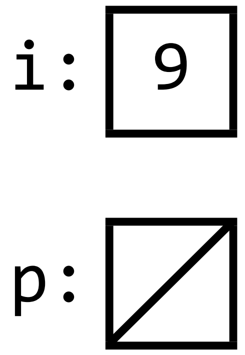
```
int i = 9;
int *p;     (p is uninitialized)

p = &i;     (p has the value &i)
p = NULL;   (p has the value NULL)
```



NULL

- A special pointer value that doesn’t point at anything
- Not the same as being uninitialized!
- Need to `#include` one of the common libraries (e.g., `stdio.h` or `stdlib.h`)

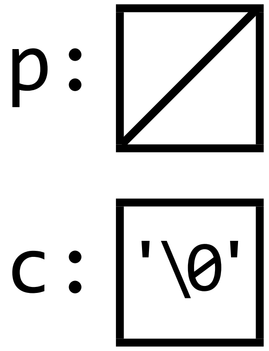
```
int i = 9;
int *p;     (p is uninitialized)

p = &i;     (p has the value &i)
p = NULL;   (p has the value NULL)
```



NULL vs. '\0'

- A pointer to NULL is called a null pointer
- The null pointer (NULL) is very different than the null character ('\0')
- One is a pointer that doesn't point to anything, the other is the character with ASCII character code 0

```
char *p = NULL;
char c = '\0';
```

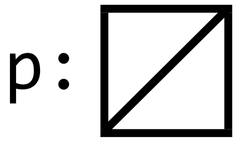


Following NULL

- You can't follow the null pointer

```
int *p = NULL;
*p = 8;
```

Wrong



- Well, it's C, so of course you *can*, but it's undefined behaviour

`malloc()` and `NULL`

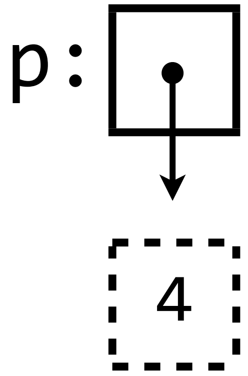
- If `malloc()` can't allocate memory, it returns the null pointer
- Technically, you should always check for this
- In this course, we won't

```
int *a = malloc(N * sizeof(int));
if (a == NULL)
{
  printf("Out of memory!");
  // handle error somehow
}
else
{
  // do something
}
```

free() and NULL

- free() doesn't change the pointer it frees

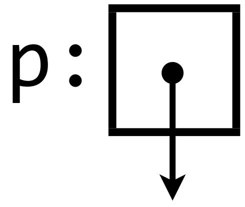
```
int *p = malloc(sizeof(int));
*p = 4;
```



free() and NULL

- free() doesn't change the pointer it frees

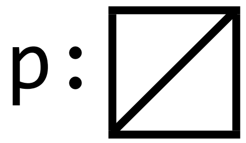
```
int *p = malloc(sizeof(int));
*p = 4;
...
free(p);
```



free() and NULL

- `free()` doesn't change the pointer it frees
- It's often a good idea to set `free()`ed pointers to `NULL`

```
int *p = malloc(sizeof(int));
*p = 4;
...
free(p);
p = NULL;
```



```
if (p has not been free()ed)
{
    do something
}
```

```
free(p);

if (???)
{
    do something
}
```

```
free(p);
p = NULL;

if (p != NULL)
{
    do something
}
```

free() and NULL

- free(NULL) is legal (and does nothing)

```
free(p);
...
free(p);
```

Wrong

```
free(p);
p = NULL;
...
free(p);
```

Not a Problem

```
q = p;
...
free(p);
p = NULL;
...
free(q);
```

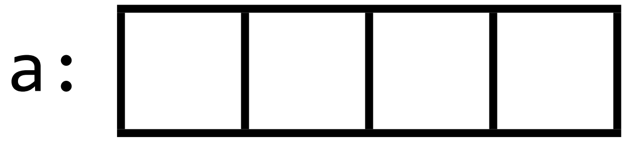
Wrong

Fixed Sizes

- Automatically allocated variables can’t change size
- Until C99, array sizes needed to be fixed at compile time

Pre-C99

```
int a[4];
```



C99

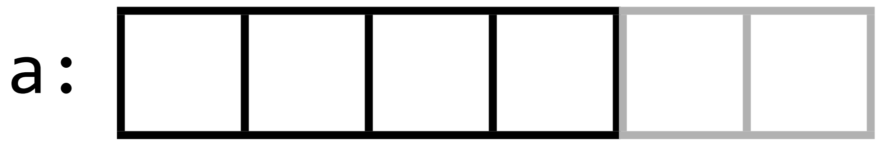
```
int n;
scanf("%d", &n);
int a[n];
```

Fixed Sizes

- Automatically allocated variables can’t change size
- Until C99, array sizes needed to be fixed at compile time

Pre-C99

```
int a[4];
```



C99

```
int n;
scanf("%d", &n);
int a[n];
```

realloc()

```
void *realloc(void *p, ~~size_t~~ size);
```

for now think of it as `int`

- Resizes a `malloc()`ed chunk of memory
- Can increase or decrease the size
- Returns a pointer to the resized memory
- `p = NULL` is equivalent to `malloc()`
- `size = 0` is equivalent to `free()`

```
int *a = malloc(4 * sizeof(int));
a = realloc(a, 5 * sizeof(int));
```