Dynamic Memory

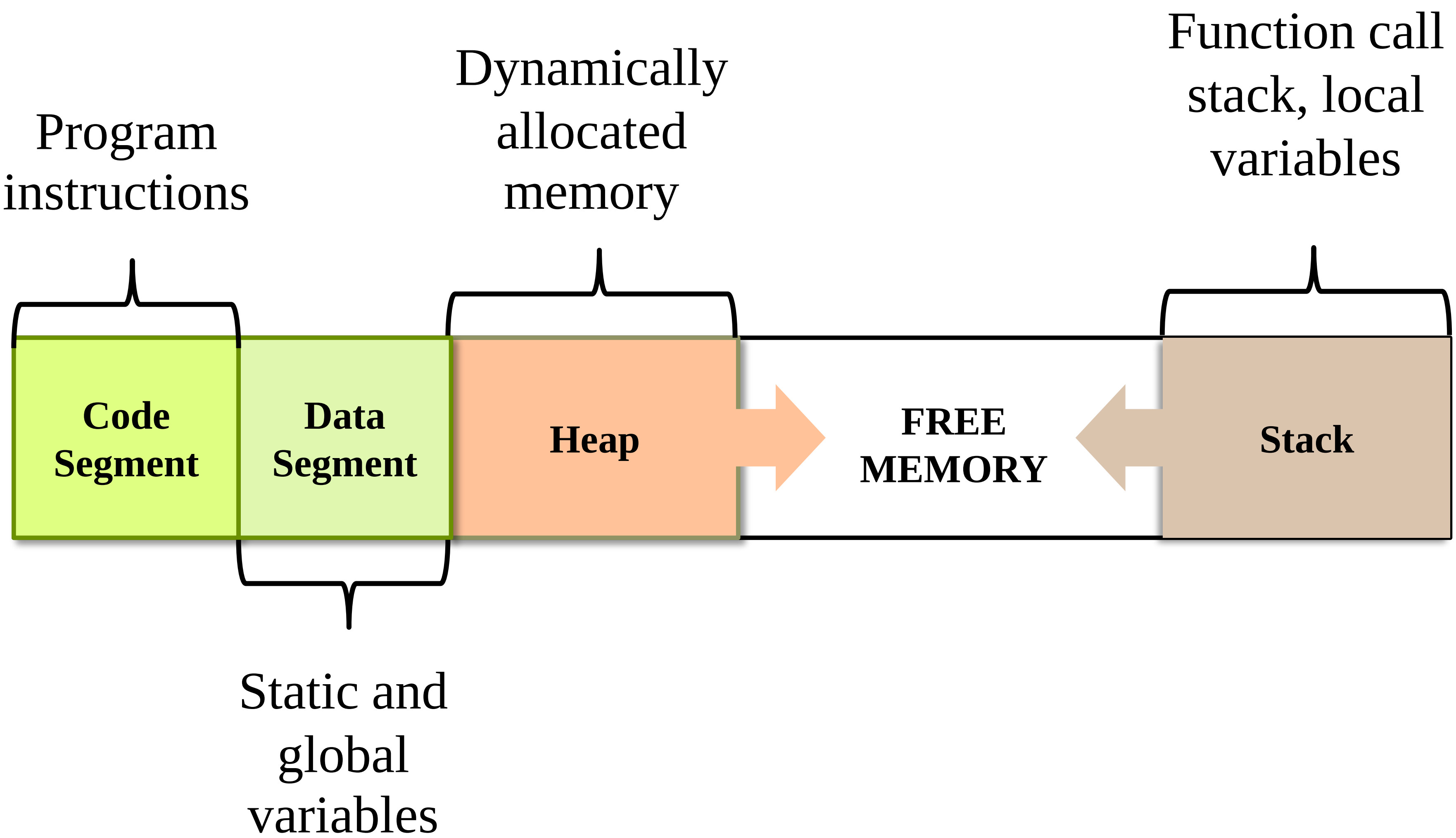
Stack and Heap

Pointers

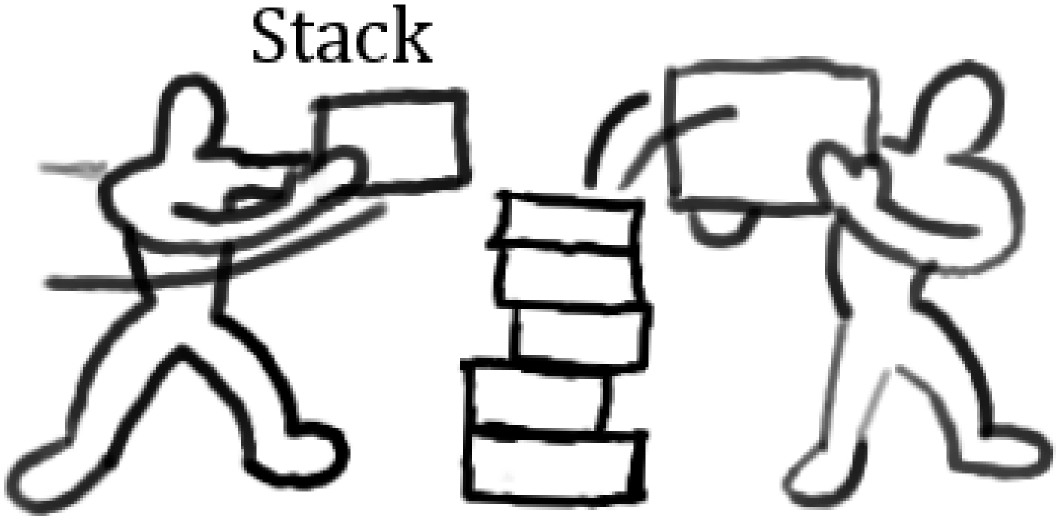
Allocating Memory on the Heap

Stack and Heap

C++ Program Memory Model

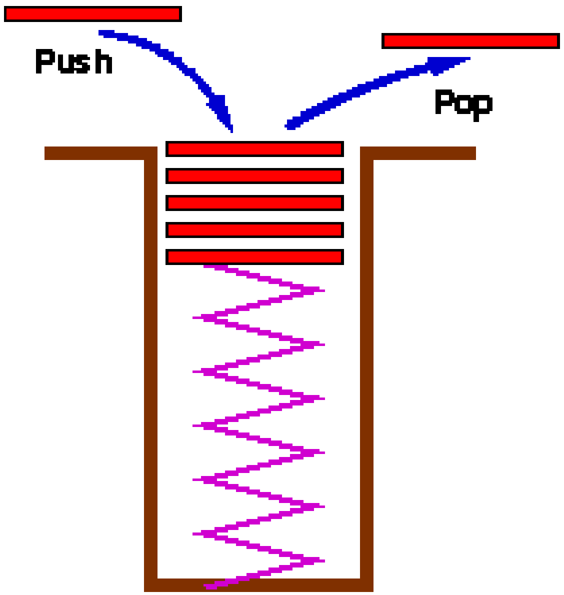


What is a stack?



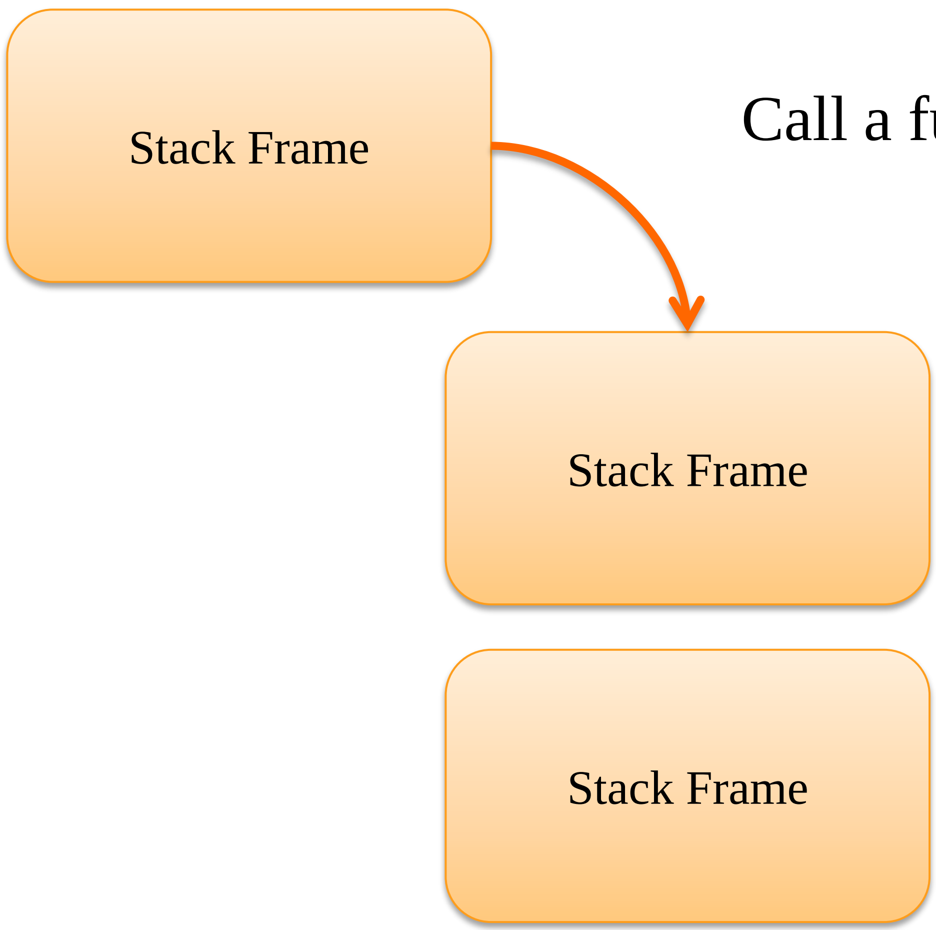
http://my.opera.com/malifarsi2/blog/show.dml/6777021

What is a stack?



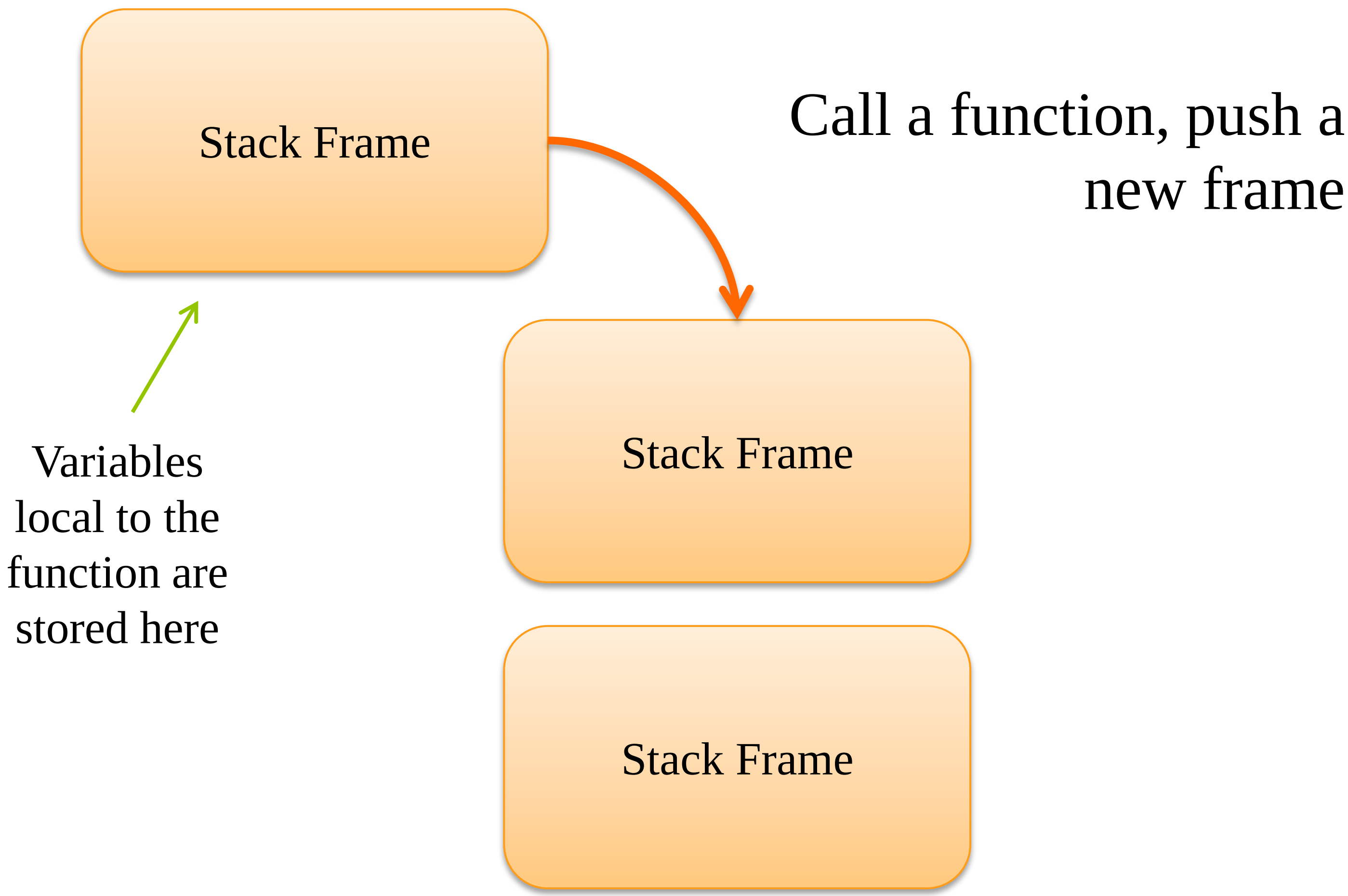
http://www.csee.umbc.edu/courses/201/spring11/lectures/adts.html#%288%29

What is the function call stack?



Call a function, push a new frame

What is the function call stack?



What is the function call stack?

```
void a()
{
}

void b()
{
  a();
}

int main()
{
  b();
}
```

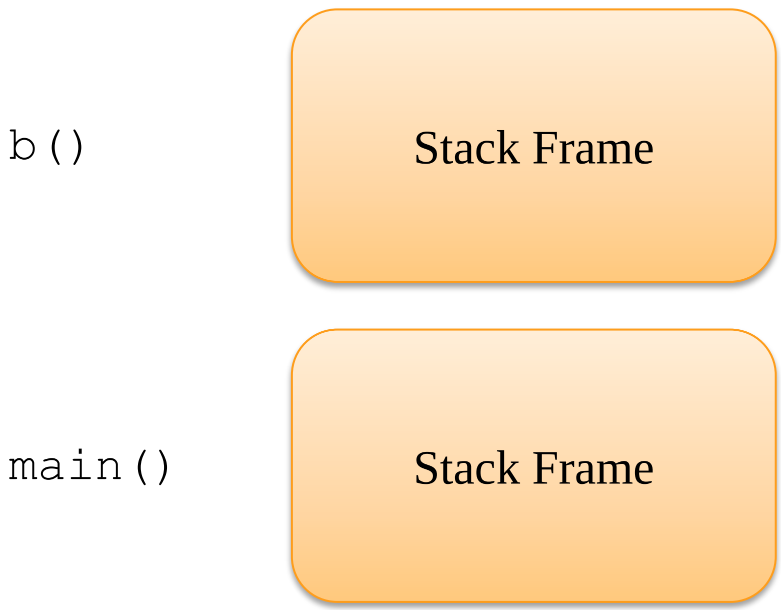
What is the function call stack?

main()

Stack Frame

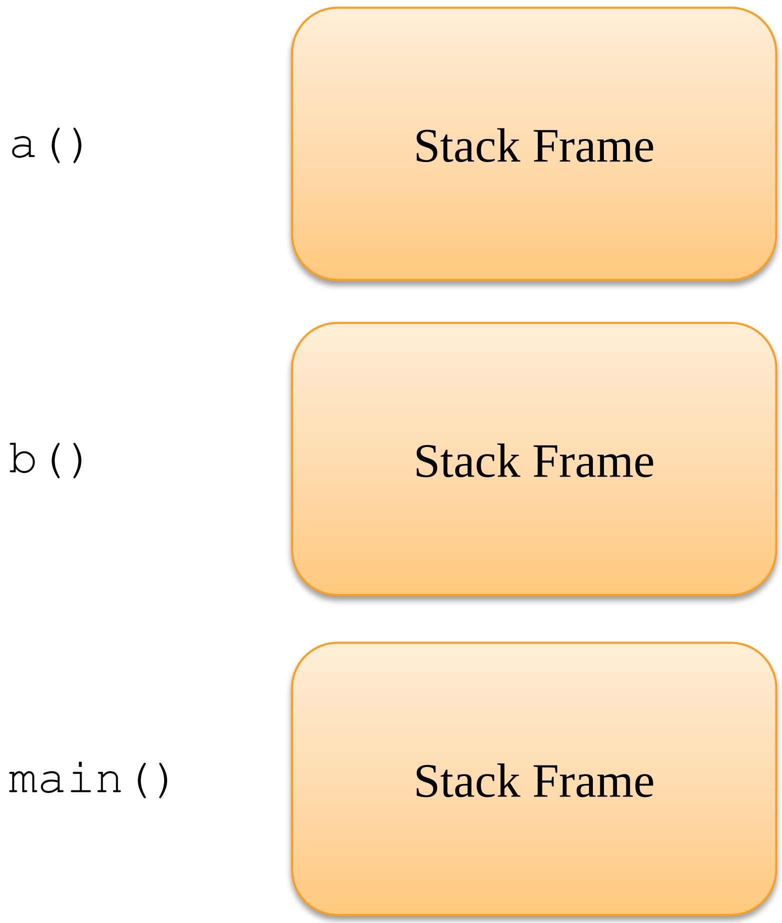
```
void a()
{
}

void b()
{
  a();
}

int main()
{
  b();
}
```

What is the function call stack?



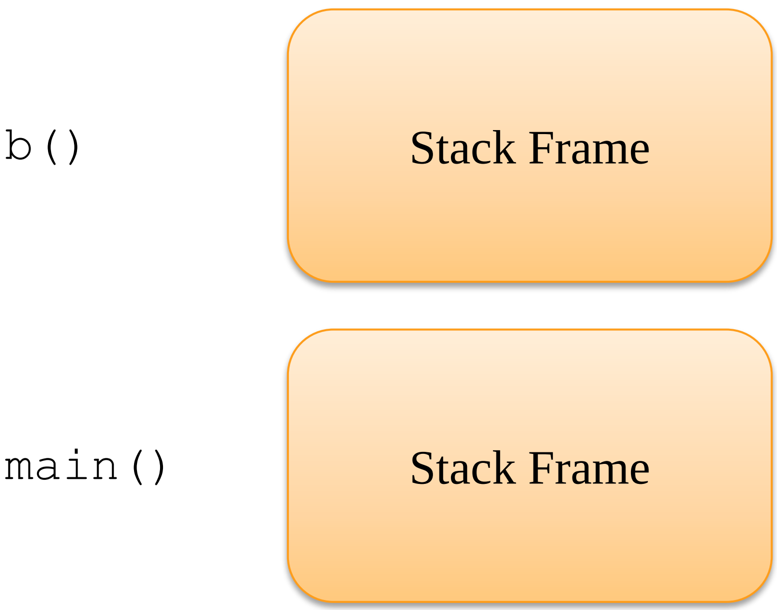

```
void a()
{
}

void b()
{
  a();
}

int main()
{
  b();
}
```

What is the function call stack?




```
void a()
{
}

void b()
{
  a();
}

int main()
{
  b();
}
```

What is the function call stack?




```
void a()
{
}

void b()
{
  a();
}

int main()
{
  b();
}
```

What is the function call stack?

```
void a()
{
}

void b()
{
   a();
}

int main()
{
   b();
}
```

main()

Stack Frame

What is the function call stack?

```
void a()
{
}

void b()
{
  a();
}

int main()
{
  b();
}
```

What is the heap?

Where dynamically allocated memory lives…

Static: allocated at compile-time
Dynamic: allocated at run-time

Static vs. Dynamic

Static: programmer declares variables, compiler reserves bytes

Once allocated, you can’t resize it!

You have to know how much memory is needed ahead of time.

Static vs. Dynamic

Dynamic: programmer reserves space on heap at run-time

Prone to memory leaks because the programmer is also responsible for freeing the space.

Pointers

Pointers and Walk-in Closets

http://computationaltales.blogspot.ca/2011/03/pointers-and-walk-in-closets.html

```
int *myPointer;
```

Declare a variable called `myPointer`

```
int *myPointer;
```

The * means it will be a pointer to something

```
int *myPointer;
```

This pointer will point to data of type `int`

```
int *myPointer;
```

```
int *myPointer;
int myInt = 5;
myPointer = &myInt;
```

This regular `int` variable is located at a particular location in memory

```
inter;
int myInt = 5;
myPointer = &myInt;
```

```
int *myPointer;
int myInt = 5;
myPointer = &myInt;
```

The & grabs the location of the integer to store in the pointer

```
int *myPointer;
int myInt = 5;
myPointer = &myInt;
```

Now `myPointer` is storing the location of `myInt`

```
int *myPointer;
int myInt = 5;
myPointer = &myInt;

cout << *myPointer << endl;
cout << myInt << endl;
```

```
int *myPointer;
int myInt = 5;
myPointer = &myInt;

cout << *myPointer << endl;
cout <<                  dl;
```

After a pointer has been declared, * lets you access its contents (if there are any!)

```
int *myPointer;
int myInt = 5;
myPointer = &myInt;

cout << *myPointer << endl;
cout << myInt << endl;
```

`myPointer` is pointing to `myInt`, so this will print the same thing as `*myPointer`

```
int *myPointer;
int myInt = 5;
myPointer = &myInt;

int *anotherPointer
    = myPointer;
```

```
int *myPointer;
int myInt = 5;
myPointer = &myInt;

int *anotherPointer
```

Another variable that will store the memory location of an `int`

```
int *myPointer;
int myInt = 5;
myPointer = &myInt;
```

```
int *anotherPointer
    = myPointer;
```

Now `anotherPointer` and `myPointer` both point to `myInt`'s memory location

```
int *myPointer;
int myInt = 5;
myPointer = &myInt;
```

```
int *anotherPointer
   = myPointer;
```

Same result:

```
anotherPointer = &myInt;
```

Binky's Fun With Pointers

http://www.youtube.com/watch?v=5VnDaHBi8dM

Functions With Pointer Parameters

```
void divide(int numerator, int denominator,
            int *dividend, int *remainder)
{
  if (denominator == 0)
  {
    cout << "Divide by zero\n";
  }

  *dividend = numerator / denominator;
  *remainder = numerator % denominator;
}
```

Pass by value

```
void divide(int numerator, int denominator,
            int *dividend, int *remainder)
{
  if (denominator == 0)
  {
    cout << "Divide by zero\n";
  }

  *dividend = numerator / denominator;
  *remainder = numerator % denominator;
}
```

```
void divide(int numerator, int denominator,
            int *dividend, int *remainder)
{
  if (denomina
  {
    cout << "Divide by zero\n";
  }

  *dividend = numerator / denominator;
  *remainder = numerator % denominator;
}
```

Pointer parameter types – memory locations instead of values

```
void divide(int numerator, int denominator,
            int *dividend, int *remainder)
{
  if (denominator == 0)
  {
    cout << "Divide by zero\n";
  }

  *dividend = numerator / denominator;
  *remainder = numerator % denominator;
}
```

Error condition

```
void divide(int numerator, int denominator,
            int *dividend, int *remainder)
{
  if (denominator == 0)
  {
    cout << "Divide by zero\n";
  }

  *dividend = numerator / denominator;
  *remainder = numerator % denominator;
}
```

Use * to save the result of the calculations into the memory location pointed to

```
int main()
{
  int x, y, d, r;

  x = 9;
  y = 2;

  divide(x, y, &d, &r);

  cout << d << " with " << r << " remainder.\n"
}
```

Static variables on the stack – these will exist until out of scope (i.e. until `main()` exits)

```
int main()
{
  int x, y, d, r;

  x = 9;
  y = 2;

  divide(x, y, &d, &r);

  cout << d << " with " << r << " remainder.\n"
}
```

```
int main()
{
  int x, y, d, r;

  x = 9;
  y = 2;

  divide(x, y, &d, &r);

  cout << d << " with " << r << " remainder.\n"
}
```

Pass by value

```
int main()
{
  int x, y, d, r;

  x = 9;
  y = 2;

  divide(x, y, &d, &r);

  cout << d << " with " << r << " remainder.\n"
}
```

Pass addresses instead of values

Poll Everywhere Question

What will the following code output?

```
int *x;
int *y = x;

int myInt = 10;
int mySecondInt = 25;

x = &myInt;

mySecondInt += *y;

cout << mySecondInt << endl;
```

Text 37607

134396: 35   **134398**: depends on address of `y`   **134406**: unknown/garbage

References are "Nice" Pointers

```
struct ball
{
  int x;
  int y;
};
```

We previously saw pass-by-reference like this…

```
void moveBall(ball &b)
{
   b.x += 5;
}

int main()
{
   ball b;
   b.x = 10;
   b.y = 20;

   moveBall(b);

   return 0;
}
```

References are "Nice" Pointers

```
struct ball
{
  int x;
  int y;
};
```

...but references are just nice ways of using pointers.

```
void moveBall(ball *b)
{
   (*b).x += 5;
}

int main()
{
   ball b;
   b.x = 10;
   b.y = 20;

   moveBall(&b);

   return 0;
}
```

References are "Nice" Pointers

```
struct ball
{
  int x;
  int y;
};
```

```
void moveBall(ball *b)
{
   b->x += 5;
}

int main()
{
   ball b;
   b.x = 10;
   b.y = 20;

   moveBall(&b);

   return 0;
}
```

Instead of dereferencing with `*`, then getting a member attribute with `.`, you can use `->`

Allocating Memory on the Heap

Creating Primitives on the Heap

```
int *intPointer = new int;
```

Creating Primitives on the Heap

```
int *intPointer = new int;
```

Using new indicates we want dynamic memory on the heap rather than static memory on the stack

Creating Primitives on the Heap

```
int *intPointer = new int;
```

This creates space on the heap, but nothing is initialized (just like with static variables)

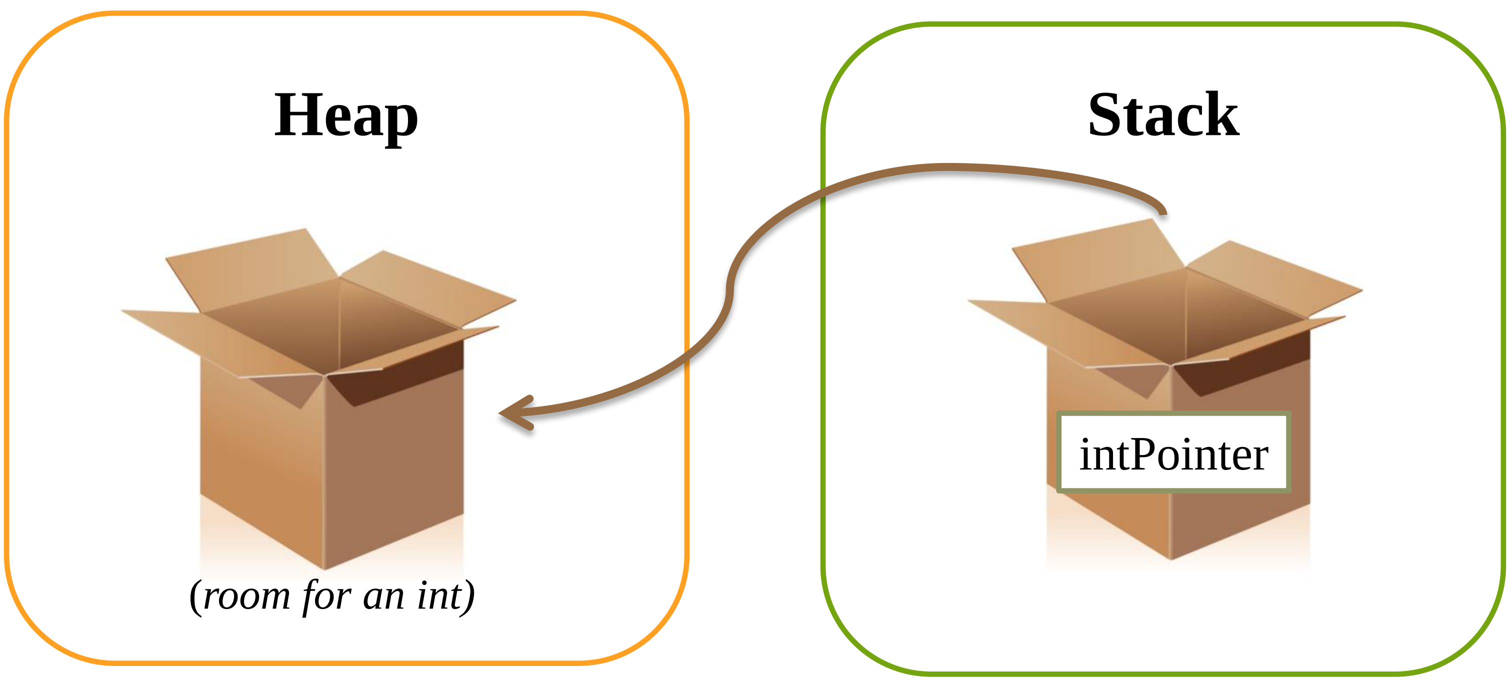
Creating Primitives on the Heap

```
int *intPointer = new int;
```

`new` returns a pointer to the location the space was created, which we can hold onto

Creating Primitives on the Heap

```
int *intPointer = new int;
```



Creating Arrays on the Heap

```
int *arrPointer = new int[2];
arrPointer[0] = 10;
arrPointer[1] = 20;
```

Creating Arrays on the Heap

```
int *arrPointer = new int[2];
arrPointer[0] = 10;
arrPointer[1] = 20;
```

Saves space for two contiguous integers in the heap

Creating Arrays on the Heap

```
int *arrPointer = new int[2];
a                = 10;
a                = 20;
```

Saves the pointer that `new` gives us so we can access our array

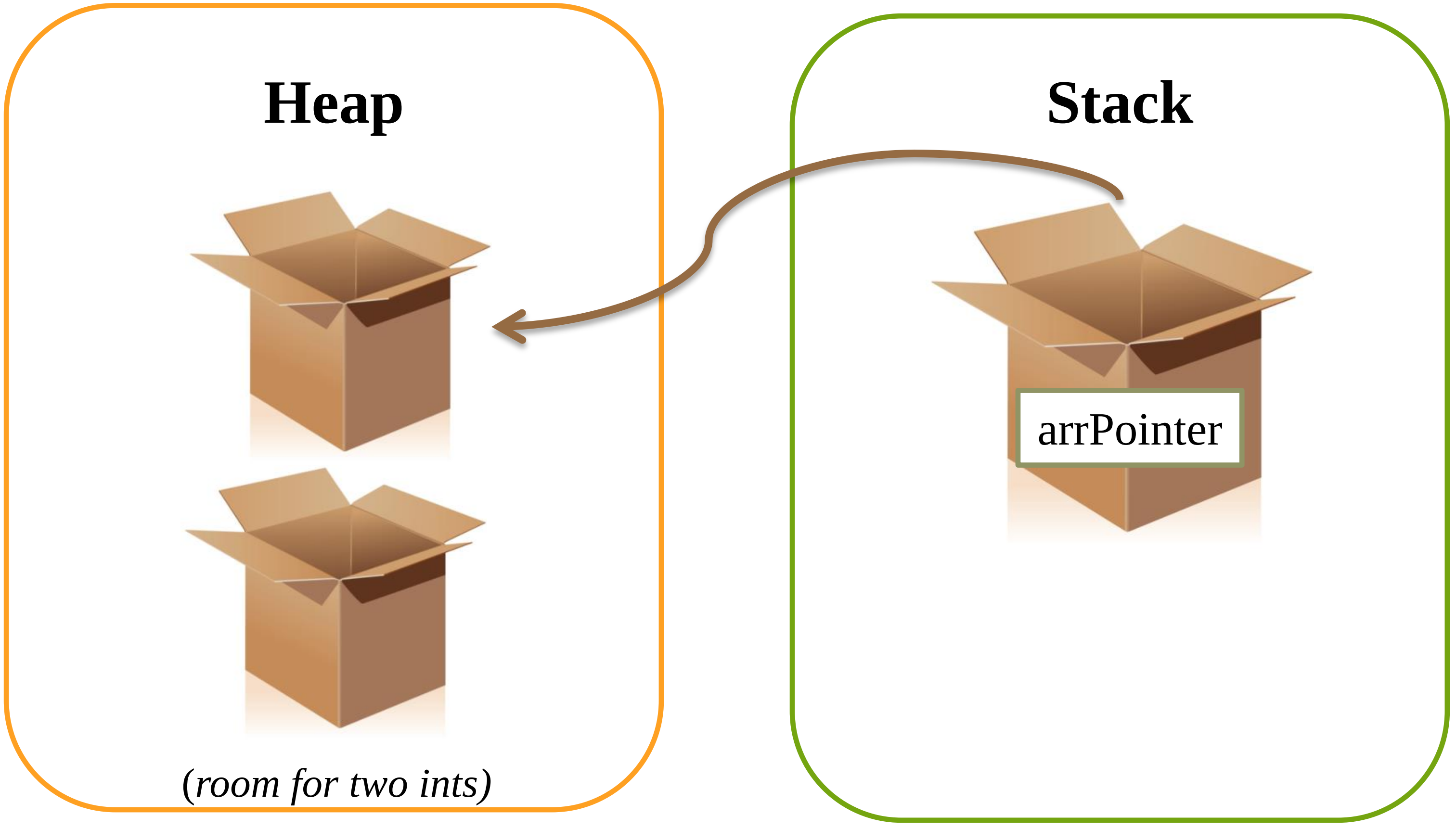
Creating Arrays on the Heap

```
int *arrPointer = new int[2];
arrPointer[0] = 10;
arrPointer[1] = 20;
```

Now we can index our array the same way as if it was on the stack

Creating Arrays on the Heap

```
int *arrPointer = new int[2];
```



Creating structs on the Heap

```
void moveBall(ball *b)
{
    (*b).x += 5;
}

int main()
{
    ball *b;
    b = new ball();
    b->x = 10;
    b->y = 20;

    moveBall(b);

    return 0;
}
```

Creating structs on the Heap

```
void moveBall(ball *b)
{
    (*b).x += 5;
}

int main()
{
    ball *b;
    b = new ball();
    b->x = 10;
    b->y = 20;

    moveBall(b);

    return 0;
}
```

We can separate pointer declaration and assignment onto two lines.

Creating structs on the Heap

```
void moveBall(ball *b)
{
    (*b).x += 5;
}

int main()
{
    ball *b;
    b = new ba
    b->x = 10;
    b->y = 20;

    moveBall(b);

    return 0;
}
```

This line only makes space for a pointer on the stack.

Creating structs on the Heap

```
void moveBall(ball *b)
{
    (*b).x += 5;
}

int main()
{
    ball *b;
    b = new ball();
    b->x = 10;
    b->y = 20;

    moveBall(b);

    return 0;
}
```

This creates space for the ball attributes on the heap and saves the pointer into `b`

Creating structs on the Heap

```
void moveBall(ball *b)
{
    (*b).x += 5;
}

int main()
{
    ball *b;
    b = new ball
    b->x = 10;
    b->y = 20;

    moveBall(b);

    return 0;
}
```

Since `b` is a pointer, we should use the arrow instead of the dot.

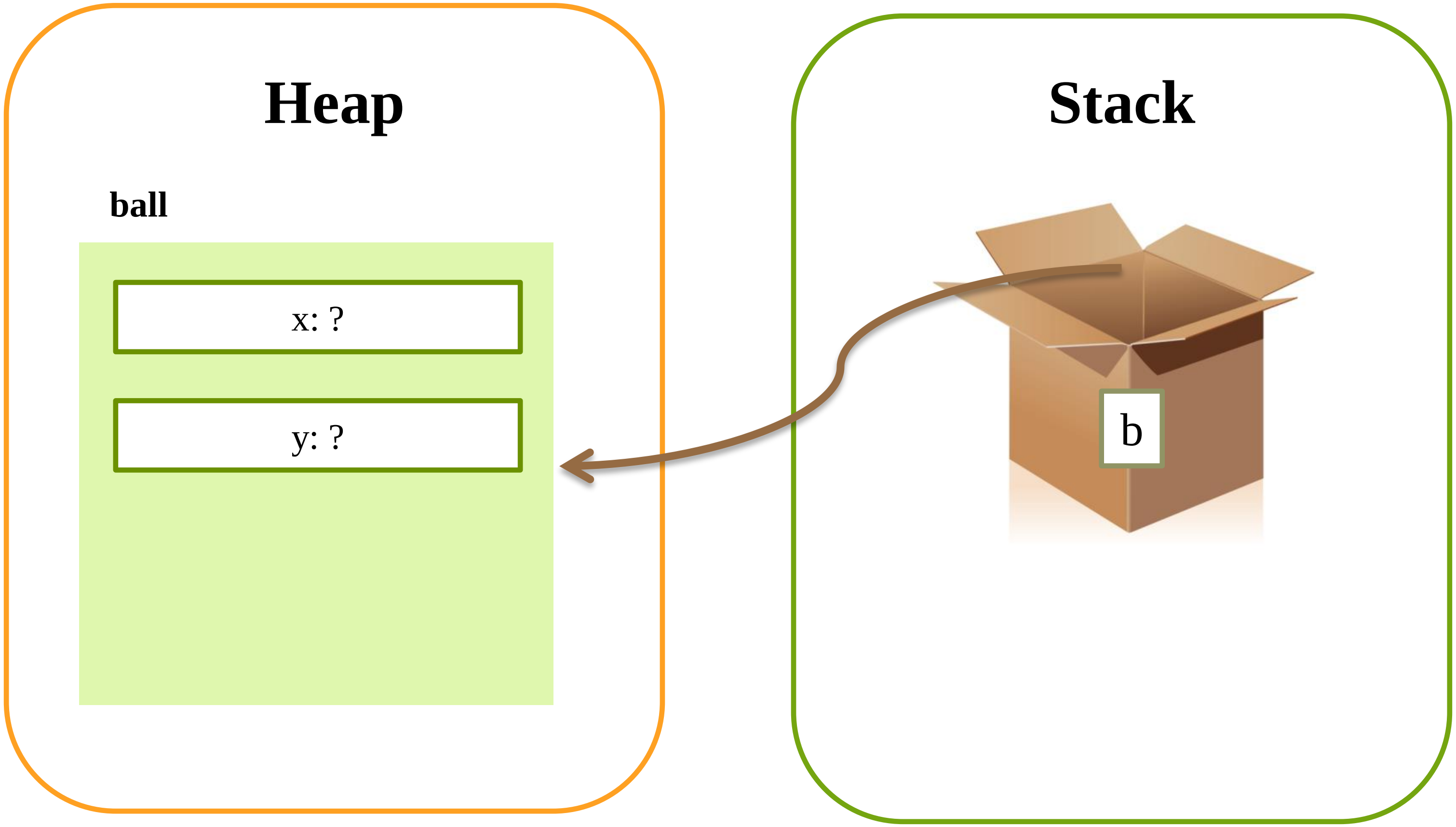
Creating structs on the Heap

```
void moveBall(ball *b)
{
    (*b).x += 5;
}

int main()
{
    ball *b;
    b = new ball();
    b->x = 10;
    b->y = 20;

    moveBall(b);

    return 0;
}
```

Since `b` is already a memory location, we don't need to use `&`.

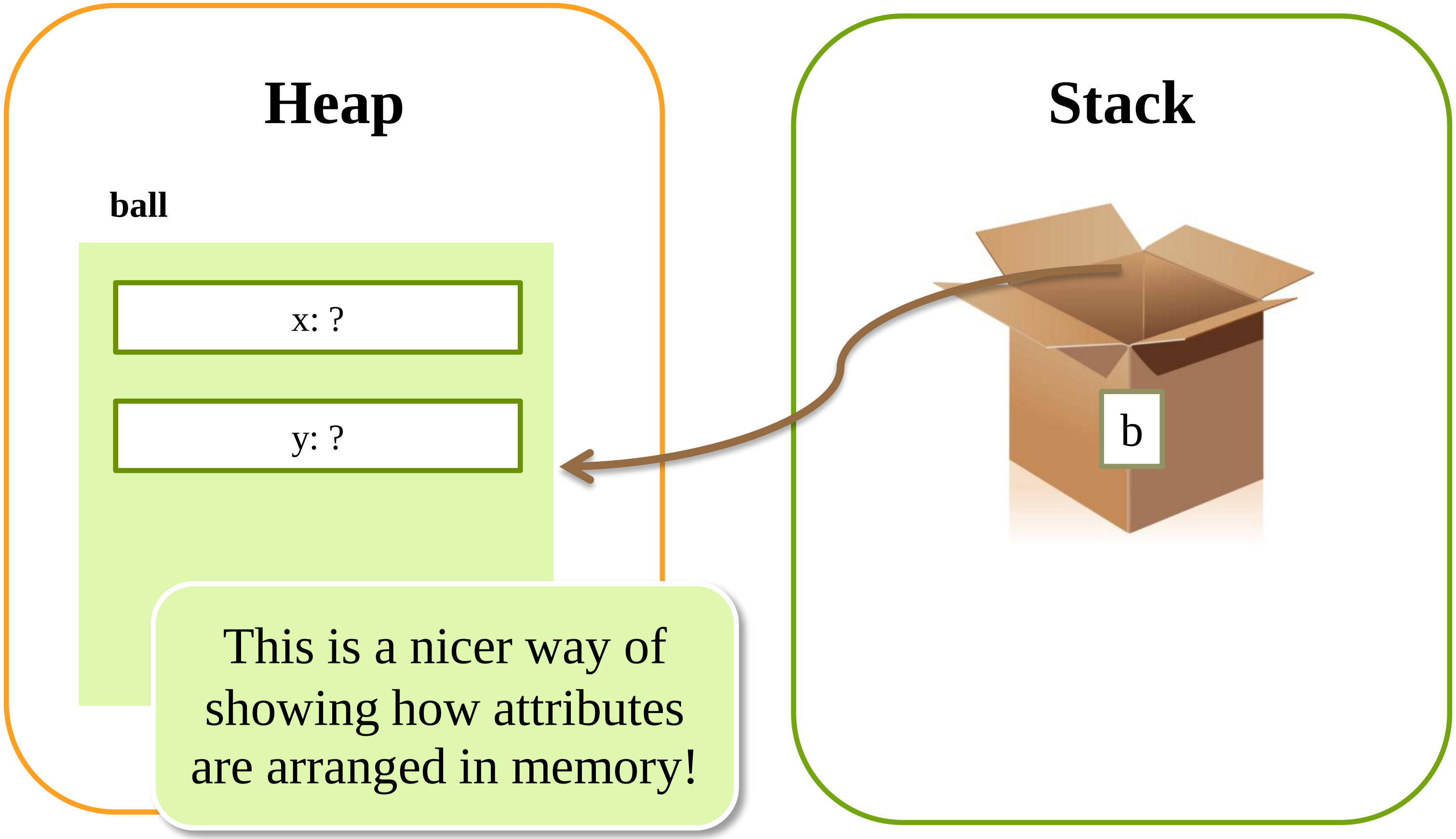
Creating structs on the Heap

```
ball *b = new ball();
```



Creating structs on the Heap

```
ball *b = new ball();
```



Deleting Dynamic Memory

We have to remove our memory from the heap when we're done with it!

```
delete intPointer;
delete [] arrPointer;
delete b;
```