

# **CS 106B, Lecture 14**

# **Pointers and Memory Management**

# Plan for Today

- How does the computer store memory? The stack and the heap
- Memory management and dynamic allocation – powerful tools that allows us to create **linked data structures** (next two weeks of the course)
  - Structs – an easy way to group variables together
  - Pointers and memory addresses – another way to refer to variables
  - Arrays
- Points are tricky! I highly encourage reading chapter 11.

# Plan for Today

- How does the computer store memory? The stack and the heap
- Memory management and dynamic allocation – powerful tools that allows us to create **linked data structures** (next two weeks of the course)
  - Structs – an easy way to group variables together
  - Pointers and memory addresses – another way to refer to variables
  - Arrays
- Points are tricky! I highly encourage reading chapter 11.

# Structs

- Like a class, but simpler
  - Collection of variables together
  - Easy way to create more complex types

```
struct Album {  
    string title;  
    int year;  
    string artist_name;  
    int artist_age;  
    int artist_num_kids;  
    string artist_spouse;  
};
```

- You can declare a variable of this type and use "." to access fields

```
Album lifeChanges;  
lifeChanges.year = 2017;  
lifeChanges.title = "Life Changes";  
cout << lifeChanges.year << endl;
```

# Struct Design

- What's wrong with this struct design?

```
struct Album {  
    string title;  
    int year;  
  
    string artist_name;  
    int artist_age;  
    int artist_num_kids;  
    string artist_spouse;  
};
```

- Style: awkward naming
- How many times do we construct the artist info?

# Struct Design

```
Album lifeChanges = {  
    "Life Changes",  
    2017,  
    "Thomas Rhett",  
    28,  
    2,  
    "Lauren"  
};
```

```
Album tangledUp = {  
    "Tangled Up",  
    2015,  
    "Thomas Rhett",  
    28,  
    2,  
    "Lauren"  
};
```

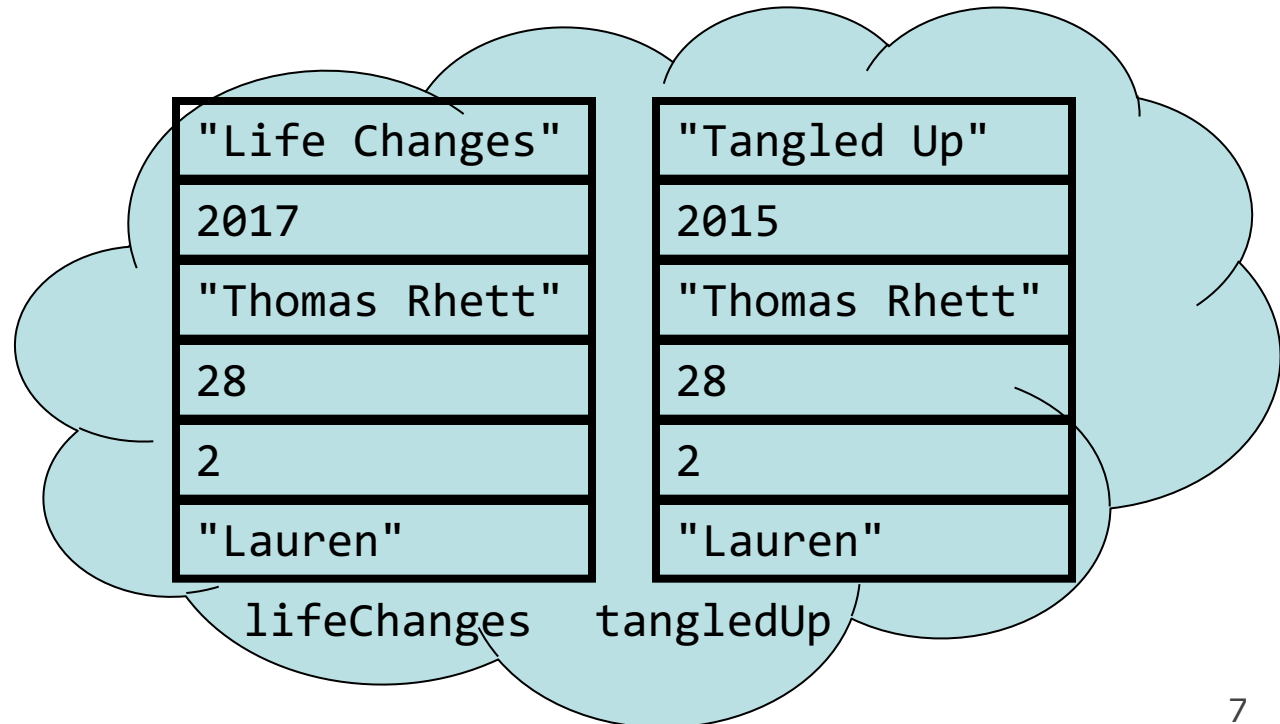
- Redundant code to declare and initialize these albums

# Struct Design

```
Album lifeChanges = {  
    "Life Changes",  
    2017,  
    "Thomas Rhett",  
    28,  
    2,  
    "Lauren"  
};
```

```
Album tangledUp = {  
    "Tangled Up",  
    2015,  
    "Thomas Rhett",  
    28,  
    2,  
    "Lauren"  
};
```

- Redundant code to declare and initialize these albums
- Redundant to store too
  - Imagine if the artist info took up a lot of space



# Fixing Redundancy

```
struct Album {  
    string title;  
    int year;  
  
    string artist_name;  
    int artist_age;  
    int artist_num_kids;  
    string artist_spouse;  
};
```



Should probably be  
another struct?



# The Artist Struct

```
struct Album {  
    string title;  
    int year;  
  
    Artist artist;  
};
```

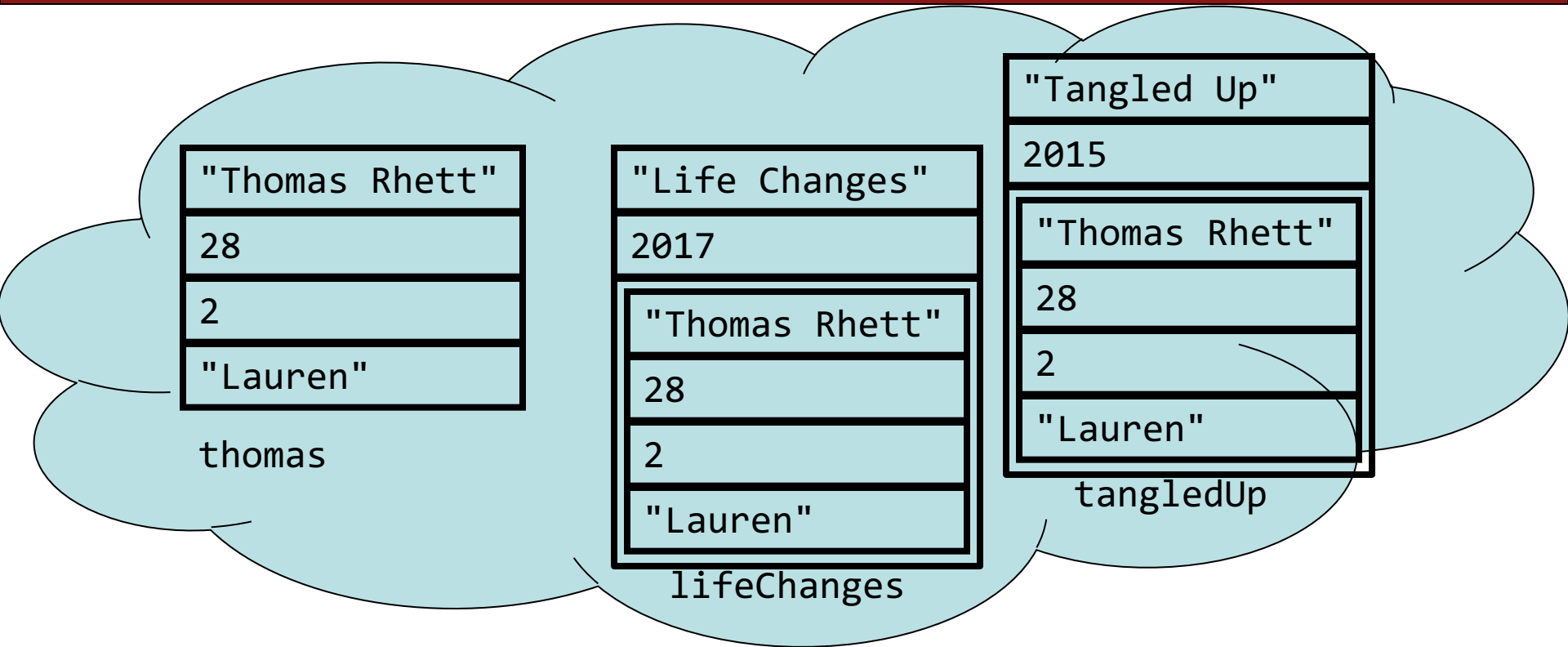
```
struct Artist {  
    string name;  
    int age;  
    int num_kids;  
    string spouse;  
};
```

```
Artist thomas = {"Thomas Rhett", 28, 2, "Lauren"};
```

```
Album lifeChanges = {"Life Changes", 2017, thomas};
```

```
Album tangledUp = {"Tangled Up", 2015, thomas};
```

# Artist In Memory

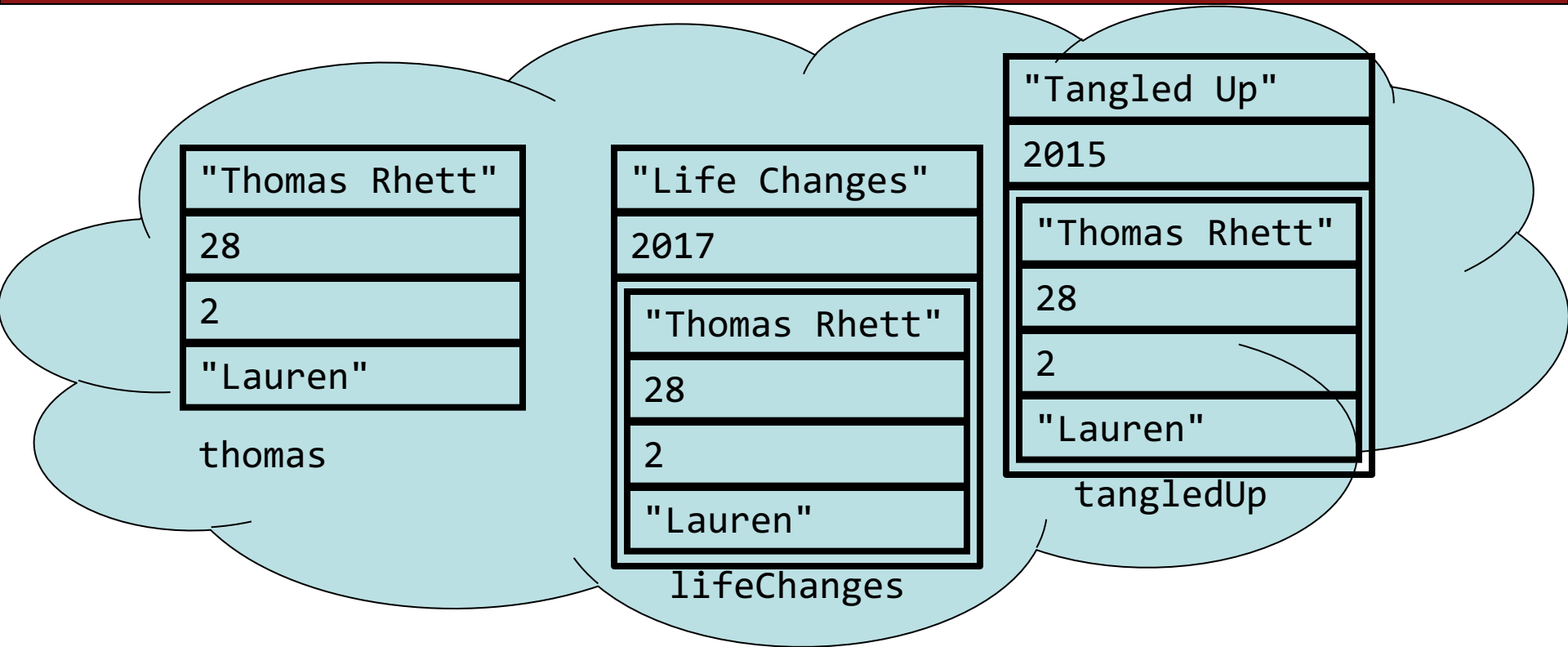


```
Artist thomas = {"Thomas Rhett", 28, 2, "Lauren"};
```

```
Album lifeChanges = {"Life Changes", 2017, thomas};
```

```
Album tangledUp = {"Tangled Up", 2015, thomas};
```

# Artful Redundancy



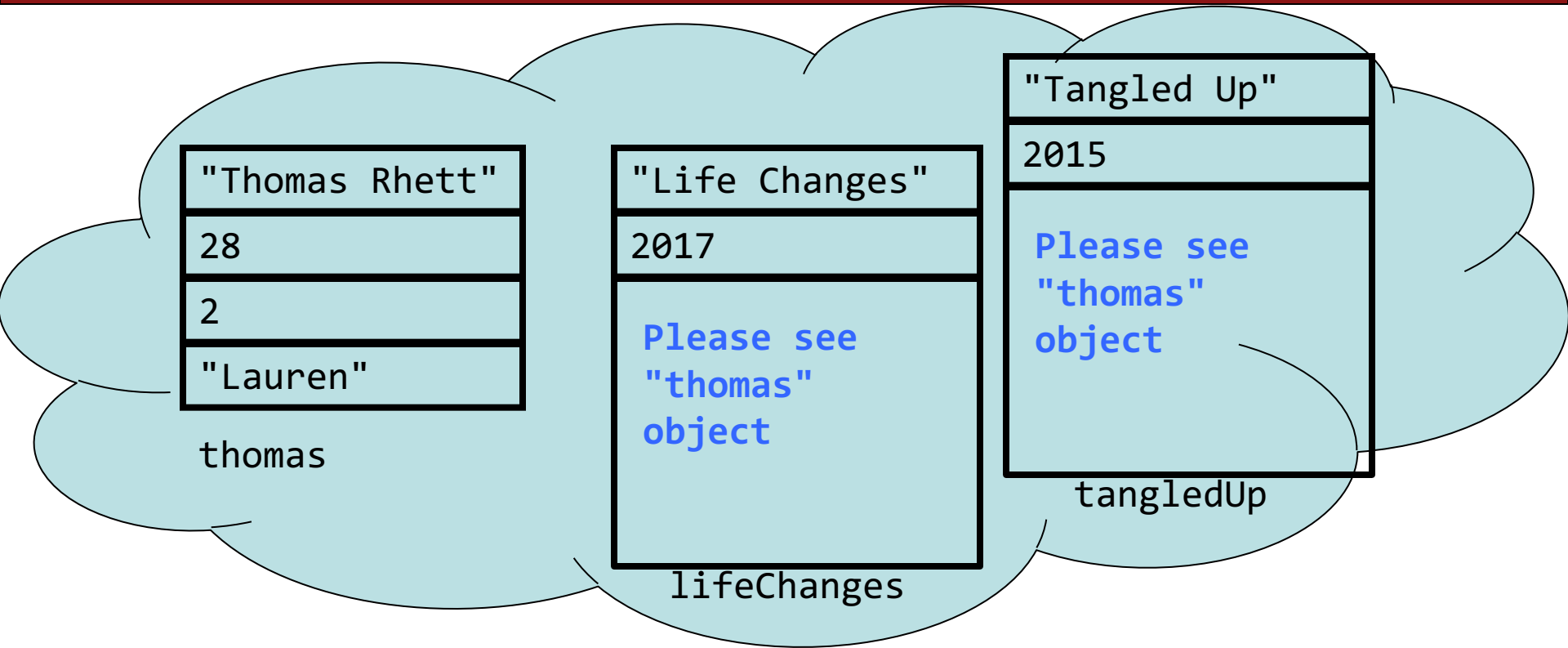
```
Artist thomas = {"Thomas Rhett", 28, 2, "Lauren"};
```

```
Album lifeChanges = {"Life Changes", 2017, thomas};
```

```
Album tangledUp = {"Tangled Up", 2015, thomas};
```

```
thomas.num_kids++; // what happens?
```

# What we want



- The artist field should **point to** or **refer to** the "thomas" data structure instead of storing it
  - if only we could just tell the computer **where in memory** to look for the thomas structure....
- In C++ - **pointers!**

# Plan for Today

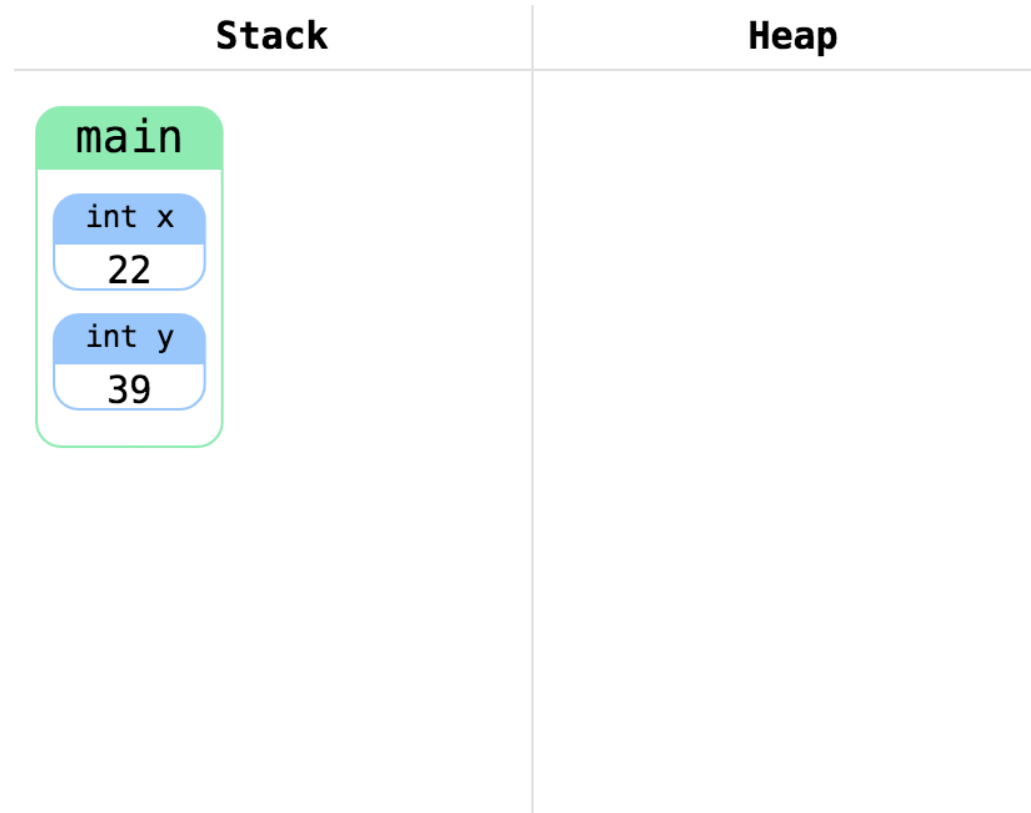
- How does the computer store memory? The stack and the heap
- Memory management and dynamic allocation – powerful tools that allows us to create **linked data structures** (next two weeks of the course)
  - Structs – an easy way to group variables together
  - Pointers and memory addresses – another way to refer to variables
  - Arrays
- Points are tricky! I highly encourage reading chapter 11.

# Computer Memory

- Creating a variable **allocates** memory (spot for the variable in the computer)
  - We number the spots in memory (just like houses) with a **memory address**
    - Can think of a computer's memory as a giant **array**, spread between stack and heap
- Stack
  - stores all the local variables, parameters, etc.
  - manages memory automatically
- Heap
  - memory that **you** manage
  - Advantage: you get to decide when the memory is freed (instead of it always disappearing at the end of a function)
  - Disadvantage: you need to manage the memory yourself

# Code Trace

```
int x = 22;  
int y = 39;
```

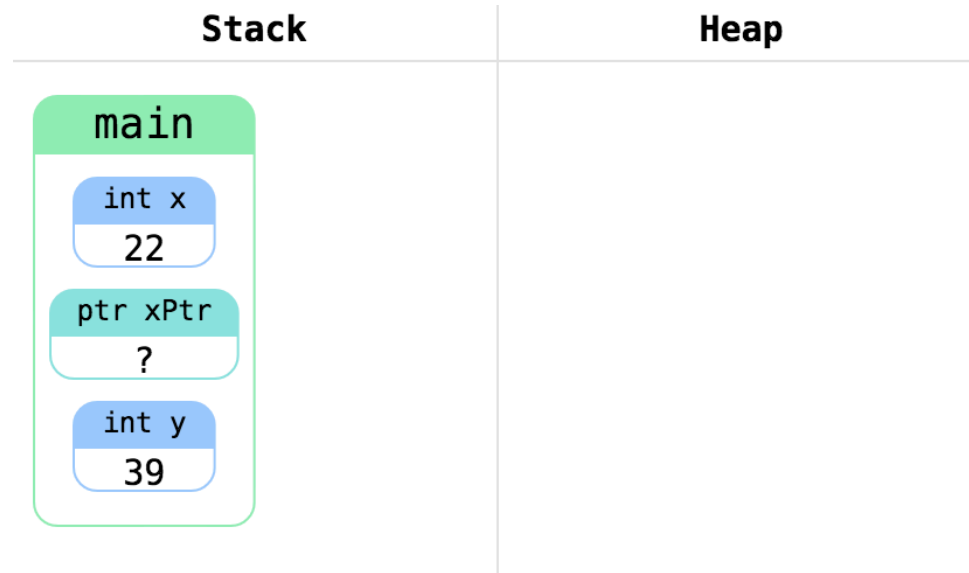


**Creating variables on the stack:**

These lines declare and initialize two variables on the stack

# Code Trace

```
int x = 22;  
int y = 39;  
int *xPtr;
```



## Creating a pointer:

xPtr will store a reference to an int

We say that a pointer "points to" a place in memory, because it stores a memory address

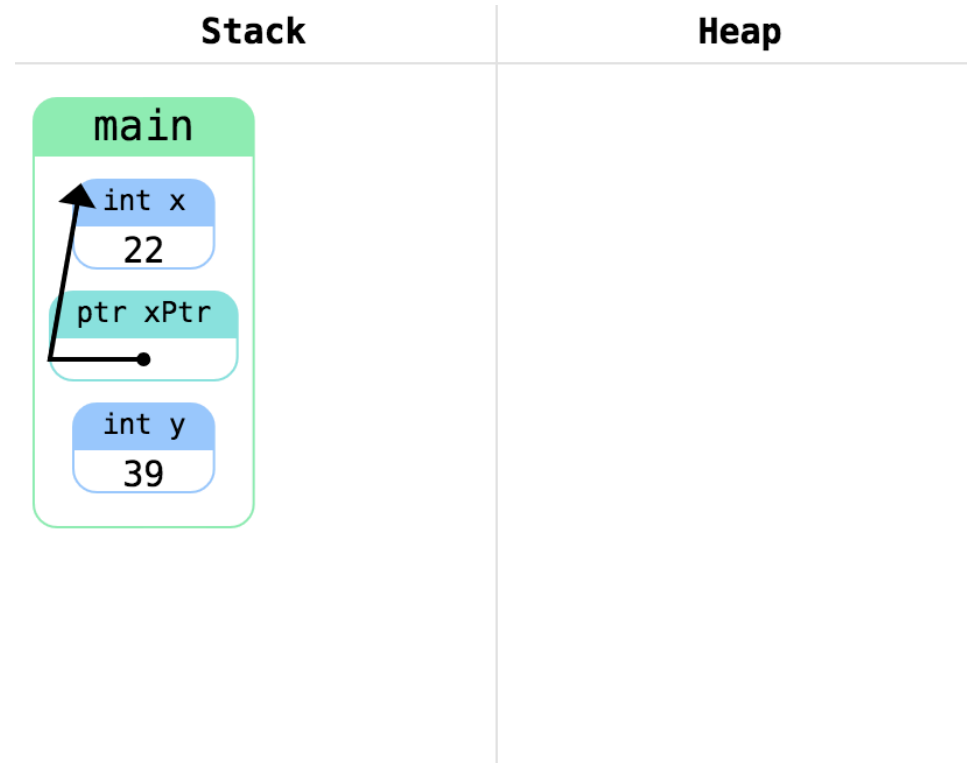
Like all local variables, xPtr is on the stack

The type before the asterisk is the type the pointer points to



# Code Trace

```
int x = 22;  
int y = 39;  
int *xPtr;  
xPtr = &x;
```

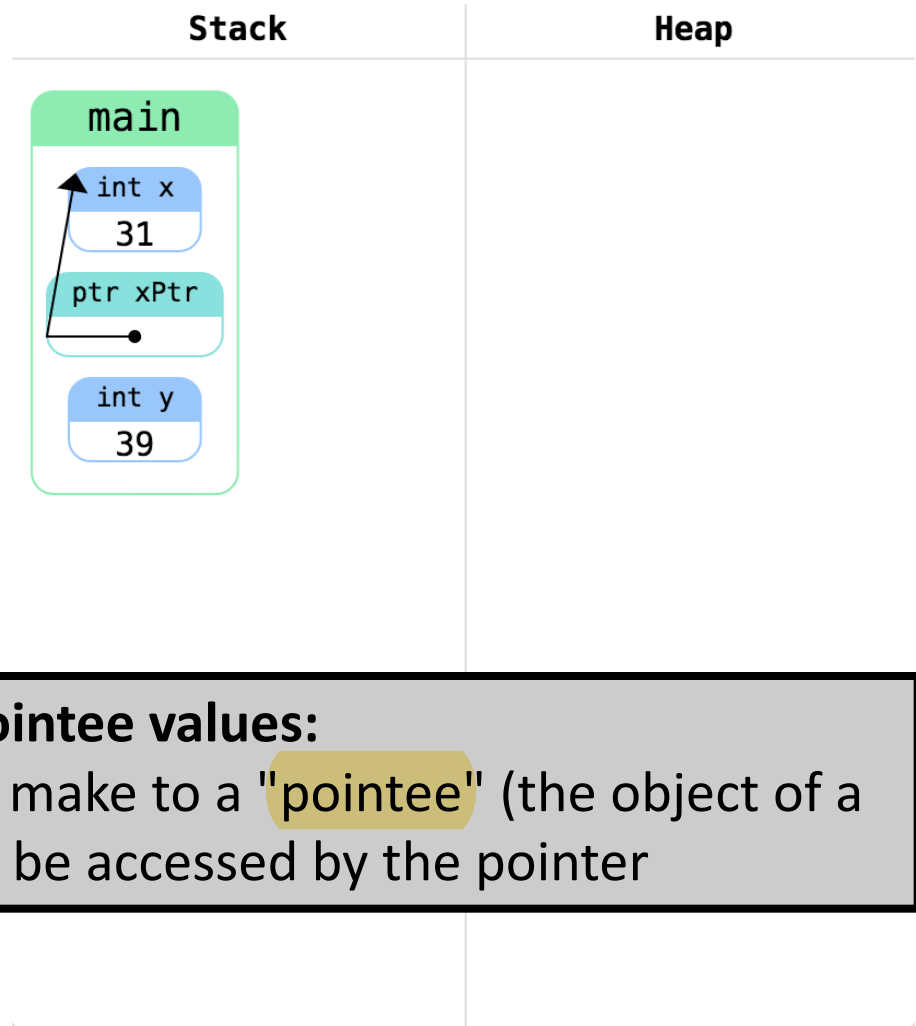


## Initializing a pointer:

xPtr now points to the variable x (the pointee)  
The & operator gets the memory address of a variable, which is now stored in xPtr

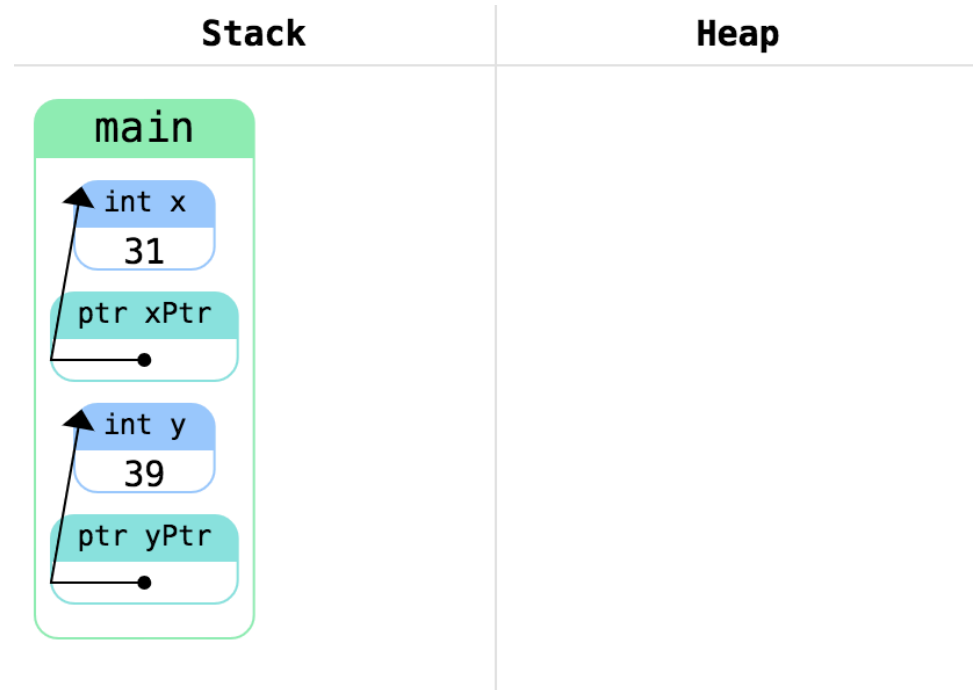
# Code Trace

```
int x = 22;  
int y = 39;  
int *xPtr;  
xPtr = &x;  
x += 9;
```



# Code Trace

```
int x = 22;  
int y = 39;  
int *xPtr;  
xPtr = &x;  
x += 9;  
int *yPtr = &y;
```

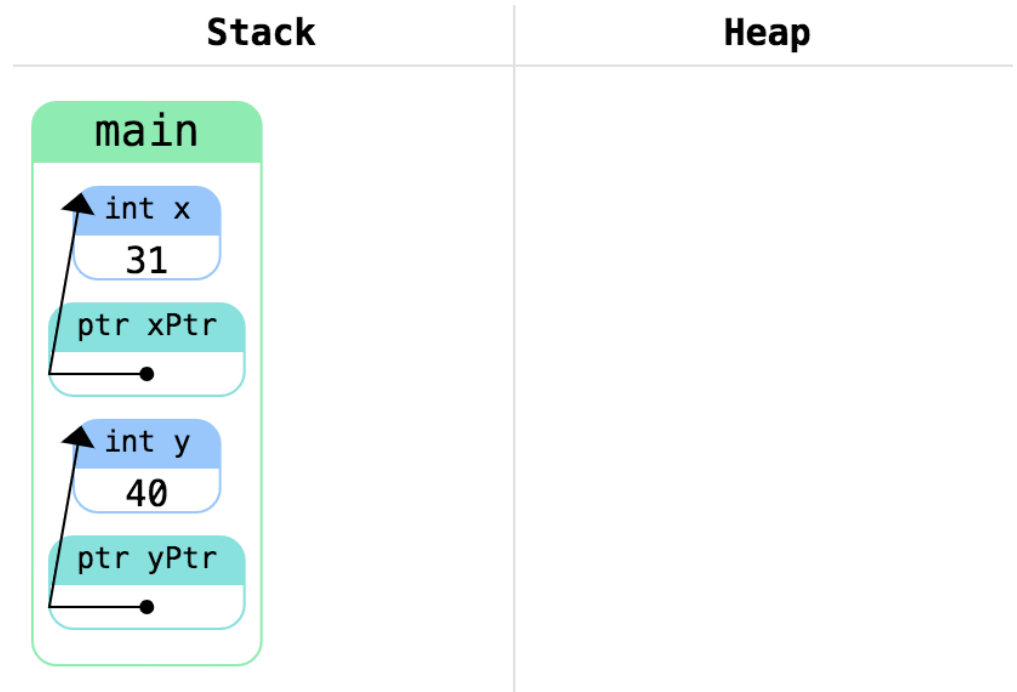


## Creating a pointer:

Here we create another pointer, this time pointing to the variable y

# Code Trace

```
int x = 22;  
int y = 39;  
int *xPtr;  
xPtr = &x;  
x += 9;  
int *yPtr = &y;  
(*yPtr)++;
```



## Accessing Pointees:

We can **dereference** a pointer using the \* operator

In this example, we add 1 to the value that yPtr points to

# The Stack

- A **pointer** is a special type that stores the address for a variable

```
int *pointer; // stores the memory address for an int
string *strPointer; // stores memory address for a string
```
- To create a variable on the stack, we just declare it (all variables you've created in this class so far have been on the stack)

```
Album lifeChanges;
```

  - We can get the memory address using an & (address operator)

```
Album *pointer = &lifeChanges;
```

# Pointer Syntax Recap

- Declaring a pointer

***type\* name;***

- Dereferencing a pointer

- Gets the variable from the address (the variable the pointer points to)
- Also uses the \*

***type variable = \*pointer;***

- To access a field in a pointer to a struct:

**`int year = (*album).year;`**

- Alternative syntax uses -> instead:

**`int year = album->year;`**

# Pointer mystery

- As parameters, pointers work similarly to references.

```
void mystery(int a, int& b, int* c) {  
    a++;  
    (*c)--;  
    b += *c;  
    cout << a << " " << b << " " << *c << " " << endl;  
}  
  
int main() {  
    int a = 4;  
    int b = 8;  
    int c = -3;  
  
    cout << a << " " << b << " " << c << " " << endl;  
    mystery(c, a, &b);  
    cout << a << " " << b << " " << c << " " << endl;  
    return 0;  
}
```

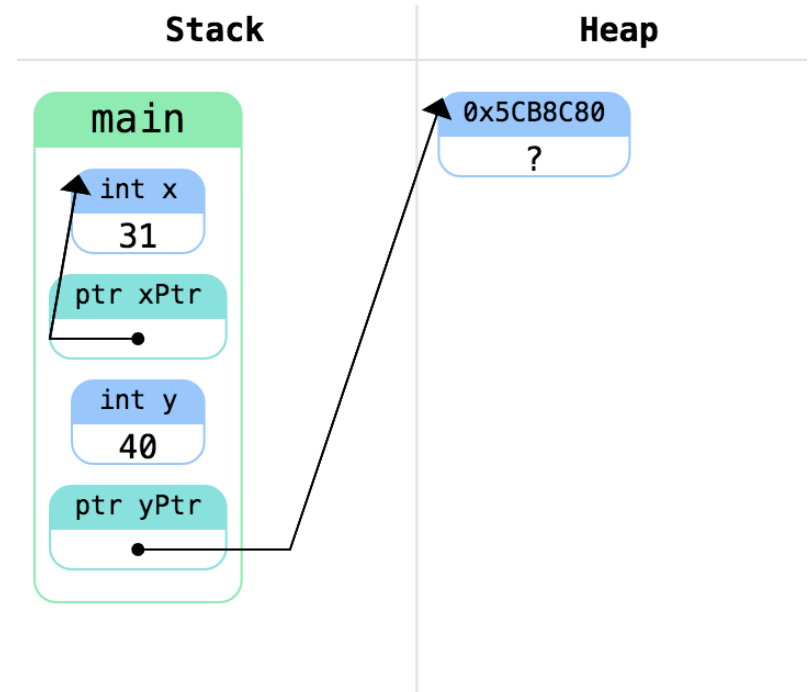
# Announcements

- Exam logistics
  - Midterm info online:  
<https://web.stanford.edu/class/cs106b/exams/midterm.html>
  - **We don't grade on style, but global variables are still not allowed**
  - General tips: use CodeStepByStep, section handouts, and redoing problems from lecture for further practice
  - **Highly Recommended:** Complete assignment 4 before the midterm – backtracking will be tested. Assignment 4 will not be due until July 25<sup>th</sup> though
  - Lectures 14 and 15 are NOT included on the midterm.
    - Though we may use a struct in a problem.



# Code Trace Continued

```
int x = 22;  
int y = 39;  
int *xPtr;  
xPtr = &x;  
x += 9;  
int *yPtr = &y;  
(*yPtr)++;  
yPtr = new int;
```



## Creating memory on the heap:

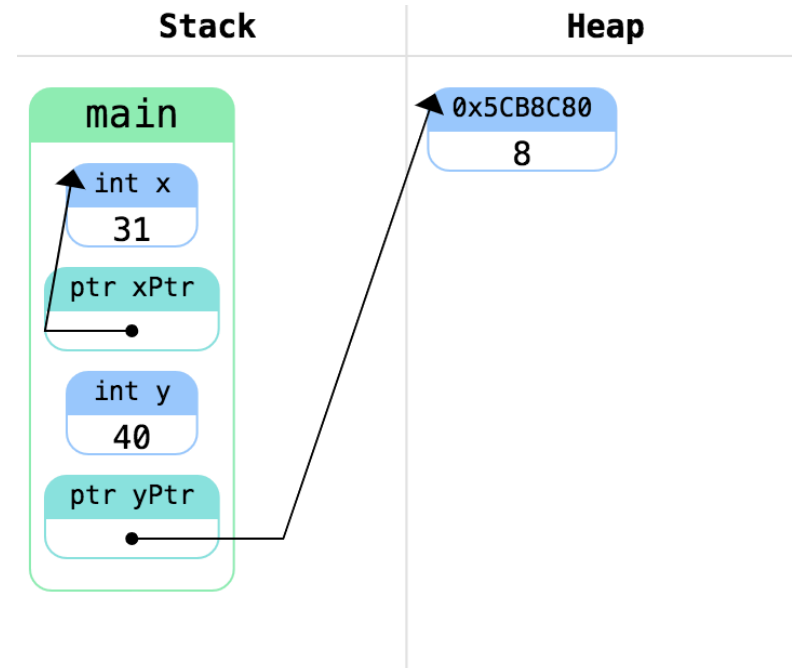
Only way to create memory on the heap is with **new**

Asks the computer for more memory

You're responsible for unallocating (freeing) the memory

# Code Trace Continued

```
int x = 22;  
int y = 39;  
int *xPtr;  
xPtr = &x;  
x += 9;  
int *yPtr = &y;  
(*yPtr)++;  
yPtr = new int;  
*yPtr = 8;
```

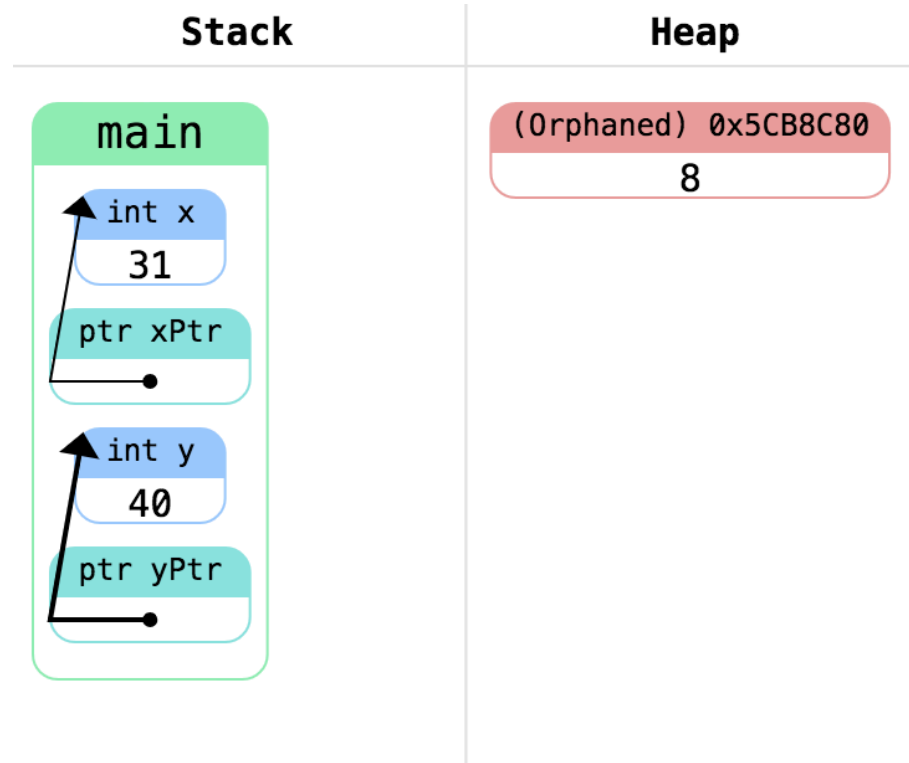


## Accessing Heap Memory:

Same as with pointers to memory on the stack  
Use the **\*** to dereference

# Code Trace Continued

```
int x = 22;  
int y = 39;  
int *xPtr;  
xPtr = &x;  
x += 9;  
int *yPtr = &y;  
(*yPtr)++;  
yPtr = new int;  
*yPtr = 8;  
yPtr = &y;
```



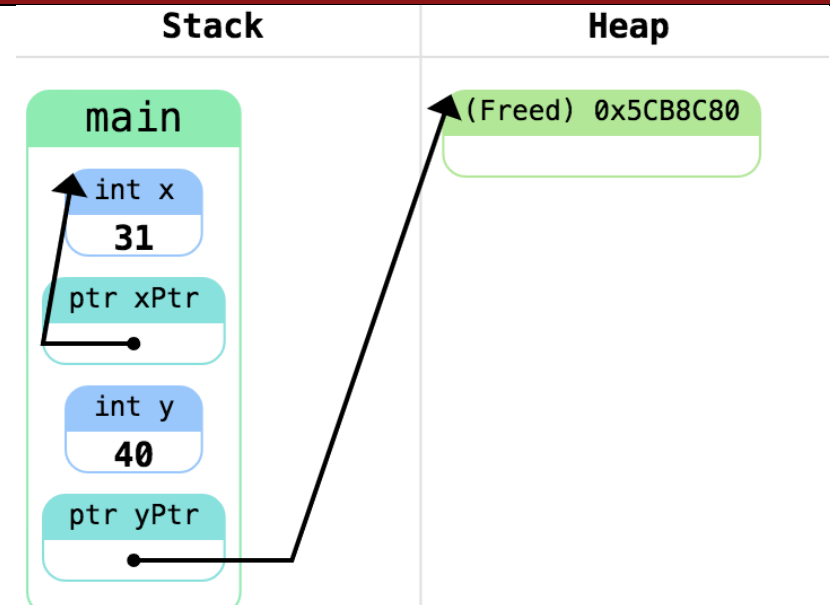
## Orphaned Memory:

If we lose all the pointers to a block of heap-allocated memory, we say it's "orphaned"

There's no way to access it or tell the computer we're done using it – that slows the computer down

# Code Trace Continued

```
int x = 22;  
int y = 39;  
int *xPtr;  
xPtr = &x;  
x += 9;  
int *yPtr = &y;  
(*yPtr)++;  
yPtr = new int;  
*yPtr = 8;  
delete yPtr;
```



## Freeing Memory:

To tell the computer we don't need the heap memory anymore, we call **delete**

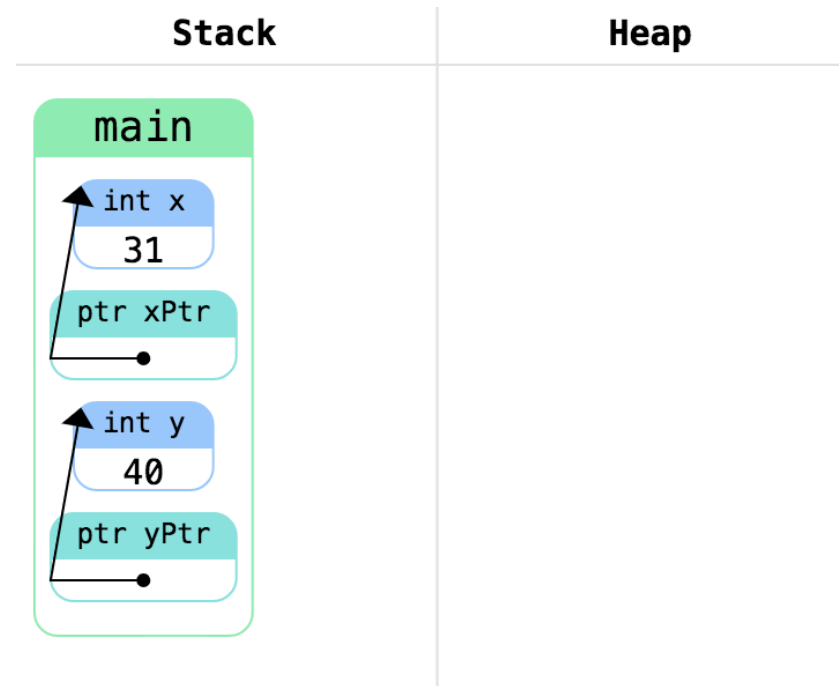
Every **new** needs a **delete**

If we dereference freed memory, unpredictable behavior (crash!)

Stack memory is automatically freed when the function ends

# Code Trace Continued

```
int x = 22;  
int y = 39;  
int *xPtr;  
xPtr = &x;  
x += 9;  
int *yPtr = &y;  
(*yPtr)++;  
yPtr = new int;  
*yPtr = 8;  
delete yPtr;  
yPtr = &y;
```



## Reassigning Pointers:

After freeing the memory, we can reassign the pointer without **leaking memory**

Calling delete changed the pointee not the pointer

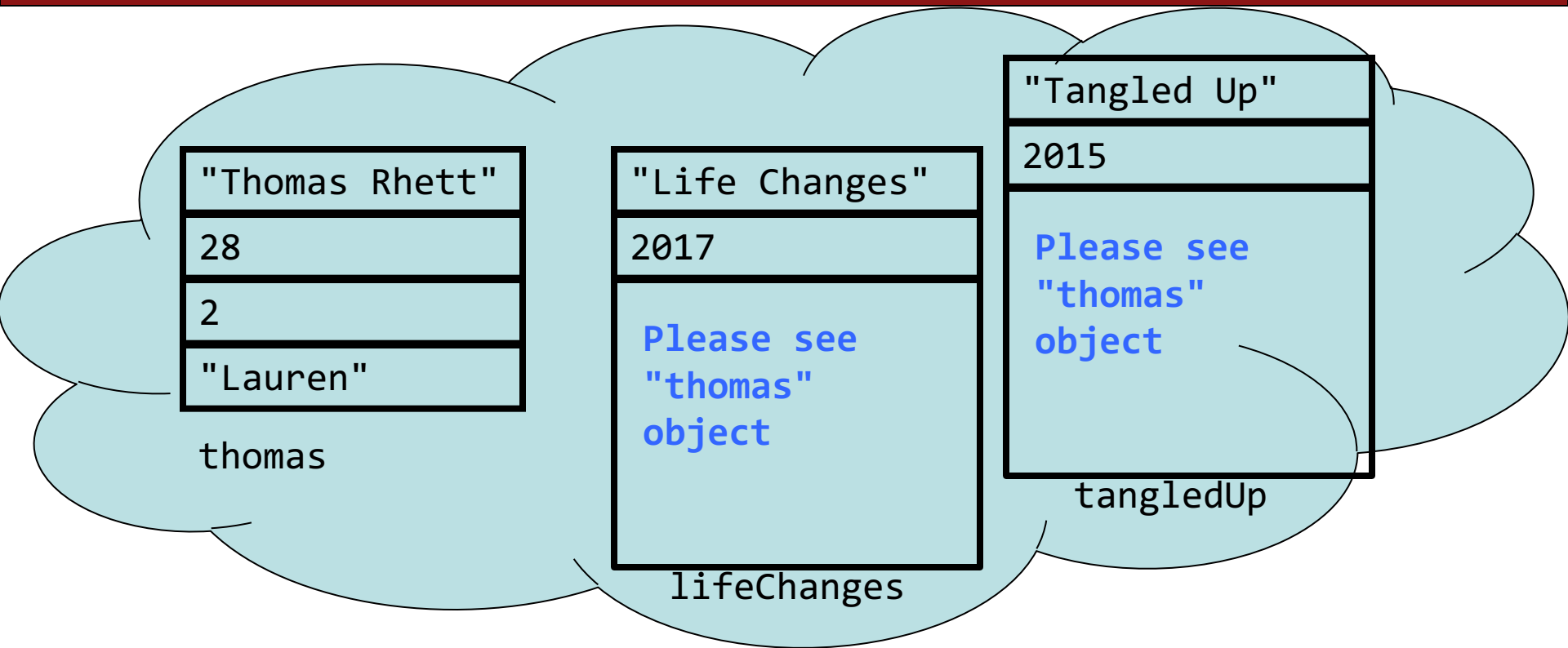
# Pointers and the Heap

- Creating a variable on the heap uses the **new** keyword
  - Allocates memory on the heap and returns the location to store in the pointer
  - Note: the pointer itself is still a local variable (it has a name)

```
Album* lifeChanges = new Album;
```

- Freeing memory – everything created must be destroyed
  - The Album will exist even if lifeChanges goes out of scope or changes values
    - "orphaning memory" – the Album isn't pointed to by anything anymore
    - When memory is orphaned, we say the program has a **memory leak**
    - Can cause your program to slow down
  - To free the Album, use the **delete** keyword **on the pointer**  
`delete lifeChanges; // lifeChanges can be reassigned now`

# Album improvements



– What should the Album struct look like?

# The Album Struct

```
struct Album {  
    string title;  
    int year;  
  
    Artist *artist;  
};
```

```
struct Artist {  
    string name;  
    int age;  
    int num_kids;  
    string spouse;  
};
```

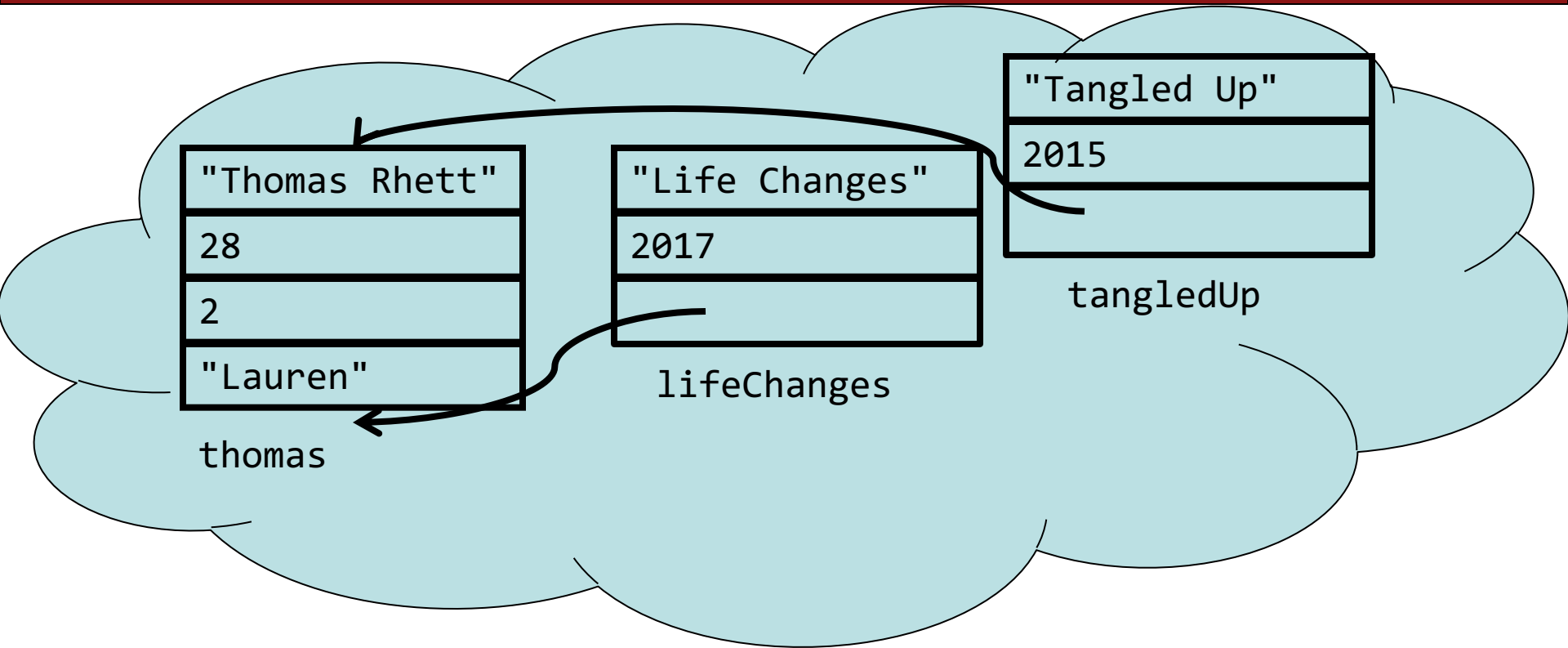
```
Artist *thomas = new Artist{"Thomas Rhett", 28, 2, "Lauren"};
```

```
Album *lifeChanges = new Album{"Life Changes", 2017, thomas};
```

```
Album *tangledUp = new Album{"Tangled Up", 2015, thomas};
```



# Album improvements



```
Artist *thomas = new Artist{"Thomas Rhett", 28, 2, "Lauren"};
Album *lifeChanges = new Album{"Life Changes", 2017, thomas};
Album *tangledUp = new Album{"Tangled Up", 2015, thomas};
cout << tangledUp->artist->spouse << endl; // "Lauren"
// later in the code, maybe in a different function
delete thomas; delete tangledUp; delete lifeChanges;
```

# Null/garbage pointers

- **null pointer**: Memory address 0; "points to nothing".
- **uninitialized pointer**: points to a random address.
  - If you dereference these, program will probably crash.

```
int x = 42;
int* p1 = nullptr;    // stores 0
int* p2;              // uninitialized
cout << p1 << endl;   // 0
cout << *p1 << endl;  // KABOOM
cout << *p2 << endl;  // KABOOM
```

0x7f8e20

0x7f8e24

0x7f8e28

x	42
p1	0x0
p2	0x??????

```
// testing for nullness
if (p1 == nullptr) {...} // true
if (p1)             {...} // false
if (!p1)            {...} // true
```

# Plan for Today

- How does the computer store memory? The stack and the heap
- Memory management and dynamic allocation – powerful tools that allows us to create **linked data structures** (next two weeks of the course)
  - Structs – an easy way to group variables together
  - Pointers and memory addresses – another way to refer to variables
  - Arrays
- Points are tricky! I highly encourage reading chapter 11.

# More Complicated Trace

```
struct Album {  
    string title;  
    int year;  
    string artist;  
};
```

```
int main() {  
    Album *myLibrary = makeLibrary();  
    // do something with library  
    delete[] myLibrary;  
    return 0;  
}
```

```
Album *makeLibrary() {  
    Album* library = new Album[3];  
    library[0] = {"Life Changes", 2017, "Thomas Rhett"};  
    library[1] = {"Montevallo", 2014, "Sam Hunt"};  
    library[2] = {"Not as Legit as Git", 2018, "Anand"};  
    return library;  
}
```

## Heap allocated memory persists:

One of the advantages of heap-allocated memory is it persists after the stack frame returns

# More Complicated Trace

```
struct Album {
    string title;
    int year;
    string artist;
};

int main() {
    Album *myLibrary = makeLibrary();
    // do something with library
    delete[] myLibrary;
    return 0;
}

Album *makeLibrary() {
    Album* library = new Album[3];
    library[0] = {"Life Changes", 2017, "Thomas Rhett"};
    library[1] = {"Montevallo", 2014, "Sam Hunt"};
    library[2] = {"Not as Legit as Git", 2018, "Anand"};
    return library;
}
```

## Arrays:

This line creates an array of size 3 on the heap

Arrays are fixed-size – you can't make them bigger or smaller

That block is pointed to by the variable library

# More Complicated Trace

```
struct Album {
    string title;
    int year;
    string artist;
};

int main() {
    Album *myLibrary = makeLibrary();
    // do something with library
    delete[] myLibrary;
    return 0;
}

Album *makeLibrary() {
    Album* library = new Album[3];
    library[0] = {"Life Changes", 2017, "Thomas Rhett"};
    library[1] = {"Montevallo", 2014, "Sam Hunt"};
    library[2] = {"Not as Legit as Git", 2018, "Anand"};
    return library;
}
```

## Array Elements:

Arrays are originally uninitialized

You can access each element by index  
(just like Vector)

Returns the actual element **NOT** a  
**pointer**

# More Complicated Trace

```
struct Album {
    string title;
    int year;
    string artist;
};

int main() {
    Album *myLibrary = makeLibrary();
    // do something with library
    delete[] myLibrary;
    return 0;
}
```

```
Album *makeLibrary() {
    Album* library = new Album[3];
    library[0] = {"Life Changes", 2017, "Thomas Rhett"};
    library[1] = {"Montevallo", 2014, "Sam Hunt"};
    library[2] = {"Not as Legit as Git", 2018, "Anand"};
    return library;
}
```

## Deleting Arrays:

Just as **new** used the square brackets to create the array, you must call **delete** with square brackets to free the array's memory

# More Complicated Trace

```
struct Album {
    string title;
    int year;
    string artist;
};

int main() {
    int size;
    Album *myLibrary = makeLibrary(size);
    // do something with library using size
    delete[] myLibrary;
    return 0;
}
```

```
Album *makeLibrary(int &size) {
    Album* library = new Album[3];
    library[0] = {"Life Changes", 2017, "Thomas Rhett"};
    library[1] = {"Montevallo", 2014, "Sam Hunt"};
    library[2] = {"Not as Legit as Git", 2018, "Anand"};
    size = 3;
    return library;
}
```

## Array Sizes:

Arrays don't have a length field, so we need to store the size in a separate variable



# Arrays

- Sometimes, you want several blocks of memory, not just one block

- Declare an array of **fixed-size**

```
Type* arr = new T[size];
```

```
int *arr = new int[7];
```

- Freeing the array (notice the brackets):

```
delete[] arr;
```

- Warnings:

- Cannot change size (grow or shrink)
- No bounds-checking – the program will have undefined behavior (crash)
- Need to store size separately