This is 183

L13: Week 8 - Wednesday

Reminders

- Project 3 due Friday!
  - Submit today for 5% extra credit
  - Submit tomorrow for 2.5% extra credit
- Final projects
  - Project options released Friday!
    - Descriptions, video demonstrations
  - Start looking for a team now

Lecture Attendance and Exam 1

- Lecture attendance and Exam 1 grades **positively** correlate!
- **5% higher** average exam grades for those who regularly attend lecture
- **10% higher** for those who have full i>Clicker points compared to 0 points
  - More preparation before lecture!

correlation **!=** causation

… but still compelling

- Freshman internship opportunity
- Tomorrow, 2/25, 7pm 1670 BBB (Beyster)
- Bring a resume!
  - Make yours *tonight*!
  - https://careercenter.umich.edu/article/resume-resources

Last Time… on EECS 183

Streams
File I/O

Reading from Files

```
#include <fstream>
using namespace std;

int main() {
    int x;
    ifstream input_file;
    input_file.open("filename");
    input_file >> x;
    input_file.close();
}
```

<iostream> vs <fstream>

- `#include<iostream>`
- `cin` is already defined
- `cin` only works on keyboard input
- No extra step necessary

- `#include<fstream>`
- Must declare variable first
- Must use `open()` to specify file to read/write
- Must use `close()` to stop reading/writing file.

State Bits

- Good 
  Everything is great!
- Fail 
  Non-fatal Error - failed to read expected data

  Examples: *failed to convert type* or *file does not exist*
- Bad 
  Fatal Error - stream can no longer be used

  Example: *unplugging a USB drive*
- EOF 
  End of the stream encountered

Moral of the Story

- **DO NOT** trust input from a stream in a **fail** state.
  - That stream no longer works for any further input
  - The program **DOES NOT** stop or give an error message
  - The variables become undefined; their values depend on the compiler

Moral of the Story

- **DO NOT** use `cin.eof()` or `ifstream.eof()` as a condition

```
while (!inFile.eof()) { ... }
```

- Use these instead:

```
while (inFile >> x) { ... }
while (!inFile.fail()) { ... }
while (inFile.good()) { ... }
```

Resetting Streams

- If `cin` or `ifstream` goes into a fail state, what do we do?
  - Use **`cin.clear()`** or **`ifstream.clear()`** to reset it
- We can wrap it all up in a function

A Function to do the Work

```
void openFile(ifstream &ins) {
    string fileName;
    cout << "Enter filename: ";
    cin >> fileName;
    ins.open(fileName);
    while (!ins.good()) {
        ins.clear();
        cout << "Error in opening file";
        cout << "Enter filename: ";
        cin >> fileName;
        ins.open(fileName);
    }
}
```

Fail State

Input: `a 3`

```
int sum = 0;
int count = 0;
int number;

while (cin >> number && number != 0) {
    sum += number;
    count++;
}

if (cin.fail()) {
    cin.clear();
    string str;
    getline(cin, str);
}
```

Fail State

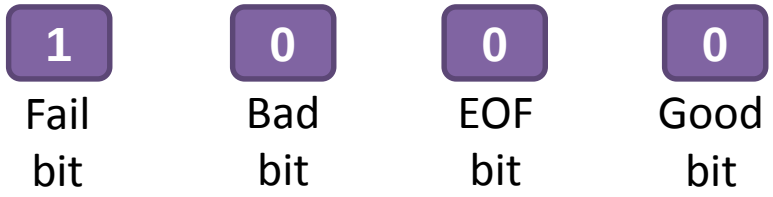
Input: **a 3**

```
int sum = 0;
int count = 0;
int number;

while (cin >> number && number != 0) {
    sum += number;
    count++;
}
cout << static_cast<double>(sum) / count;
if (cin.fail()) {
    cin.clear();
    string str;
    getline(cin, str);
}
```

Fail State

Input: `a 3`

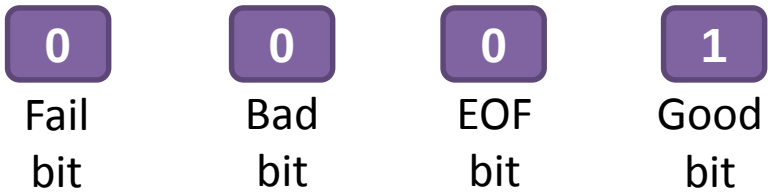
```
int sum = 0;
int count = 0;
int number;

while (cin >> number && number != 0) {
    sum += number;
    count++;
}

if (cin.fail() ) {
    cin.clear();
    string str;
    getline(cin, str);
}
```



Fail State

Input: `a 3`

```
int sum = 0;
int count = 0;
int number;

while (cin >> number && number != 0) {
    sum += number;
    count++;
}

if (cin.fail() ) {
    cin.clear();
    string str;
    getline(cin, str);
}
```



Fail State

Input: **a 3**

```
int sum = 0;
int count = 0;
int number;

while (cin >> number && number != 0) {
    sum += number;
    count++;
}
if (cin.fail() ) {
    cin.clear();
    string str;
    getline(cin, str);
}
```

i>Clicker #1 (W12, Exam 3)

```
int x = 9;
while (!(cin >> x)) {
   cin.clear();
}
cout << x;
```

User input:

Console

```
x  3
```

What prints?

A. 3
B. 9
C. x
D. The evaluation goes into an infinite loop

i>Clicker #1 (W12, Exam 3)

```
int x = 9;
while (!(cin >> x)) {
   cin.clear();
}
cout << x;
```

User input:

Console

```
x  3
```

What prints?

A. 3
B. 9
C. x
D. The evaluation goes into an infinite loop

This infinite loop is because `cin >> x` keeps reading in the same bad data.

Need `getline(cin, junk);` to clear it.

i>Clicker #2

- If `infile.txt` contains the following, what prints?

`infile.txt` **1 2 3 4**

```
ifstream infile;
infile.open("infile.txt");
int num = 0;
int sum = 0;
while (infile >> num) {
    sum += num;
}
cout << sum;
```

A. 6
B. 10
C. 14
D. None of the above

i>Clicker #2

- If `infile.txt` contains the following, what prints?

`infile.txt` **1 2 3 4**

```
ifstream infile;
infile.open("infile.txt");
int num = 0;
int sum = 0;
while (infile >> num) {
    sum += num;
}
cout << sum;
```

A. 6
B. 10
C. 14
D. None of the above

Custom Data Types

Structured Types

A Motivating Example

- Imagine you have to write a grade book for 43,000 students
- Each student needs to have:
  - First Name
  - Last Name
  - Uniqname
  - UMID number
  - …
- How would you store all this information?

A Motivating Example

```
string first_names[43000];
string last_names[43000];
string uniqnames[43000];
int umids[43000];
…
```

A Motivating Example

```
void delete_student(string arr[43000],
                    int index) {
    for (int i = index; i < 43000 - 1; i++) {
        arr[i] = arr[i+1];
    }
}
int main() {
    delete_student(first_names, 5309);
    delete_student(last_names, 5309);
    delete_student(uniqnames, 5309);
    // and so on...
    return 0;
}
```

A Motivating Example

- What does this look like in memory?

first_names	"Kevin"	"Madeline"	"Helen"	...
last_names	"Lee"	"Endres"	"Hagos"	...
uniqnames	"mrkevin"	"endremad"	"hahagos"	...
umids	86753009	55573442	09991120	...
statuses	UGRAD	UGRAD	GRAD	...
	[0]	[1]	[2]	

A Motivating Example

- What does this look like in memory?
  - What if instead of grouping by rows…

first_names	"Kevin"	"Madeline"	"Helen"	...
last_names	"Lee"	"Endres"	"Hagos"	...
uniqnames	"mrkevin"	"endremad"	"hahagos"	...
umids	86753009	55573442	09991120	...
statuses	UGRAD	UGRAD	GRAD	...
	[0]	[1]	[2]	

A Motivating Example

- What does this look like in memory?
  - What if instead of grouping by rows…we group by cols

first_names	"Kevin"	"Madeline"	"Helen"	...
last_names	"Lee"	"Endres"	"Hagos"	...
uniqnames	"mrkevin"	"endremad"	"hahagos"	...
umids	86753009	55573442	09991120	...
statuses	UGRAD	UGRAD	GRAD	...
	[0]	[1]	[2]	

A Motivating Example

- What does this look like in memory?
  - What if instead of grouping by rows…
  - …we group by column instead?

	"Kevin"	"Madeline"	"Helen"	...
	"Lee"	"Endres"	"Hagos"	...
	"mrkevin"	"endremad"	"hahagos"	...
	86753009	55573442	09991120	...
	UGRAD	UGRAD	GRAD	...
	2013	2013	2015	...
	...	...	...	...

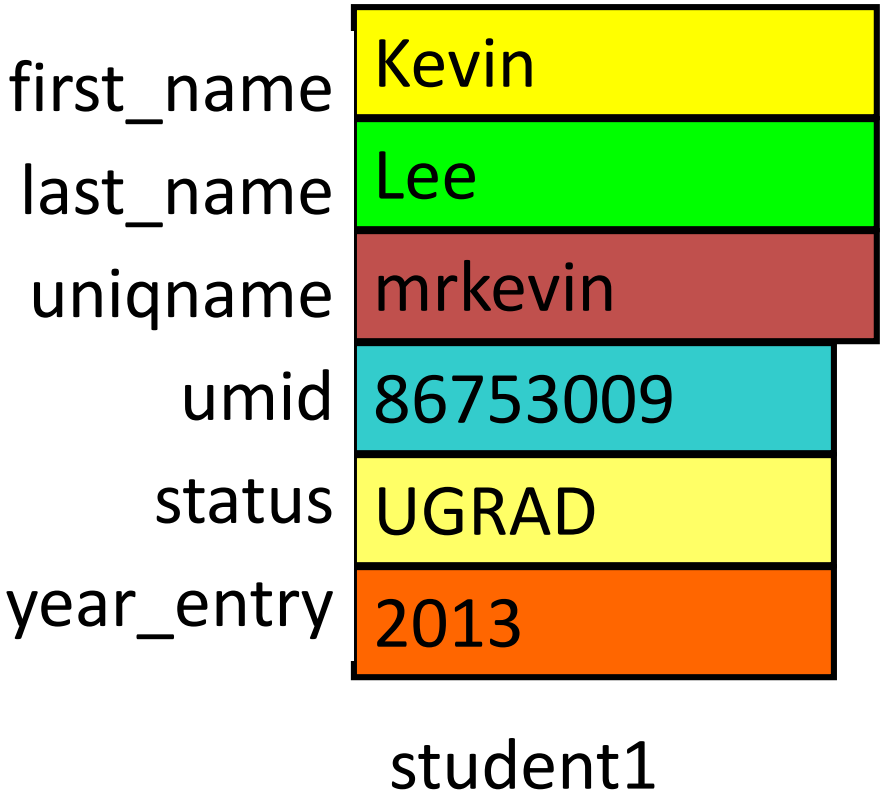
A Motivating Example

- What does this look like in memory?
  - What if instead of grouping by rows…we group by cols

	student[0]	student[1]	student[2]	
first_name	"Kevin"	"Madeline"	"Helen"	...
last_name	"Lee"	"Endres"	"Hagos"	...
uniqname	"mrkevin"	"endremad"	"hahagos"	...
umid	86753009	55573442	09991120	...
status	UGRAD	UGRAD	GRAD	...
year_entry	2013	2013	2015	...
	...	...	...	...

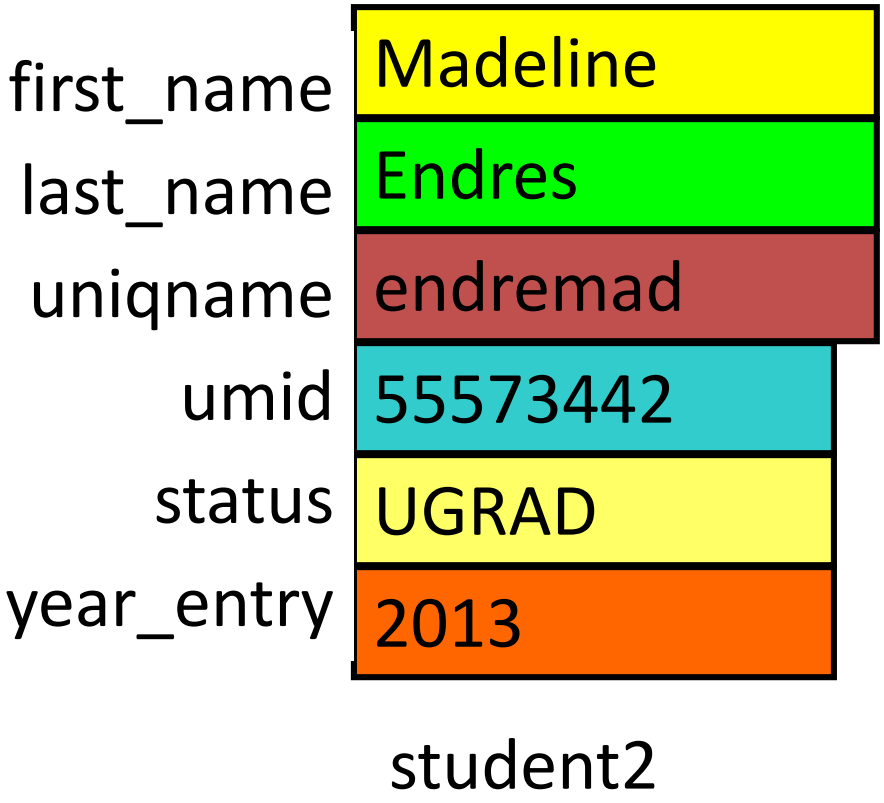
Picturing a single student

- Group together all data for one student



Picturing a 2nd student

- Group together all data for one student



Why Structured Types

- Basic data types: **`int`**, **`double`**, **`char`**, etc.

- **Array** – *homogenous* group of basic data type items

  e.g., `int arr[100];`

Won't Work – mixed datatypes

class (ADT record)

a `class` groups related data together

```
class Student {
  public:
    string first_name;
    string last_name;
    string uniqname;
    int umid;
    int status;
    int year_entry;
};
```

class (ADT record)

a `class` groups related data together

```
class Student {
  public:
    string first_name;
    string last_name;
    string uniqname;
    int umid;
    int status;
    int year_entry;
};
```

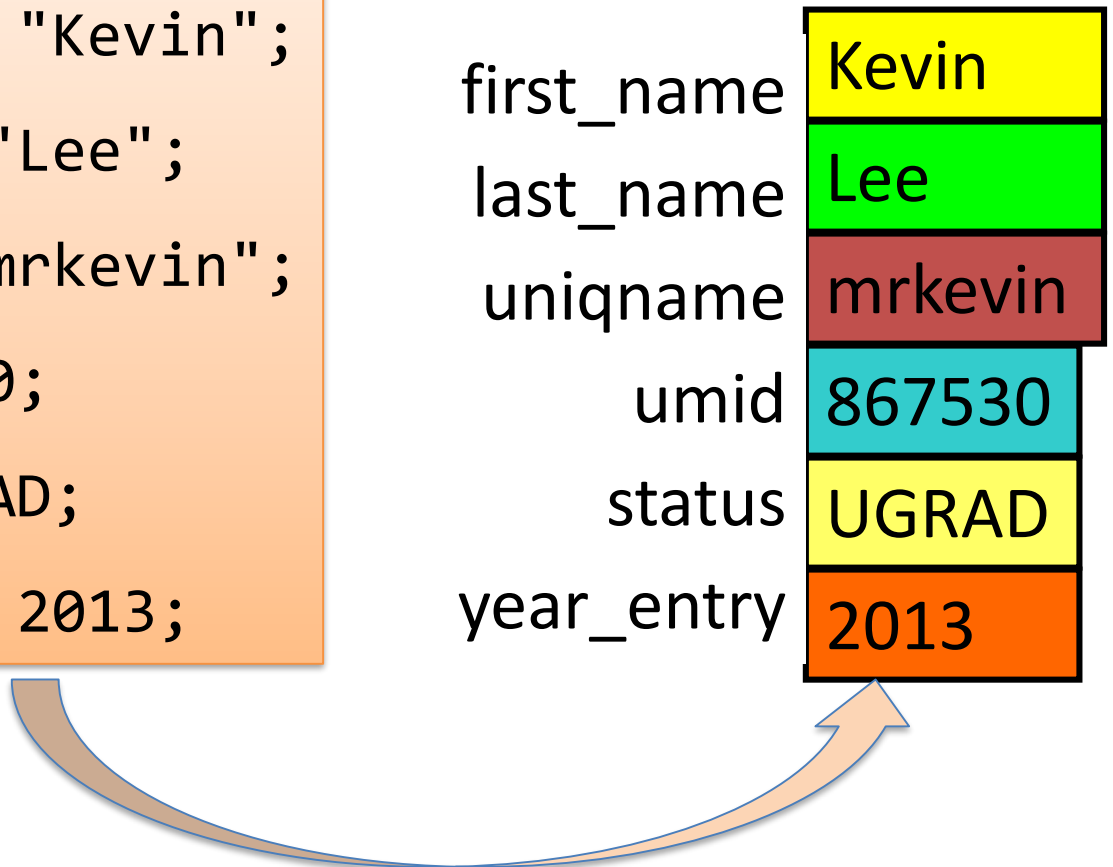
Declared normally

```
Student student1,
        student2;
```

Using class's

Access components using the "dot operator"

```
student1.first_name = "Kevin";
student1.last_name = "Lee";
student1.uniqname = "mrkevin";
student1.umid = 867530;
student1.status = UGRAD;
student1.year_entry = 2013;
```



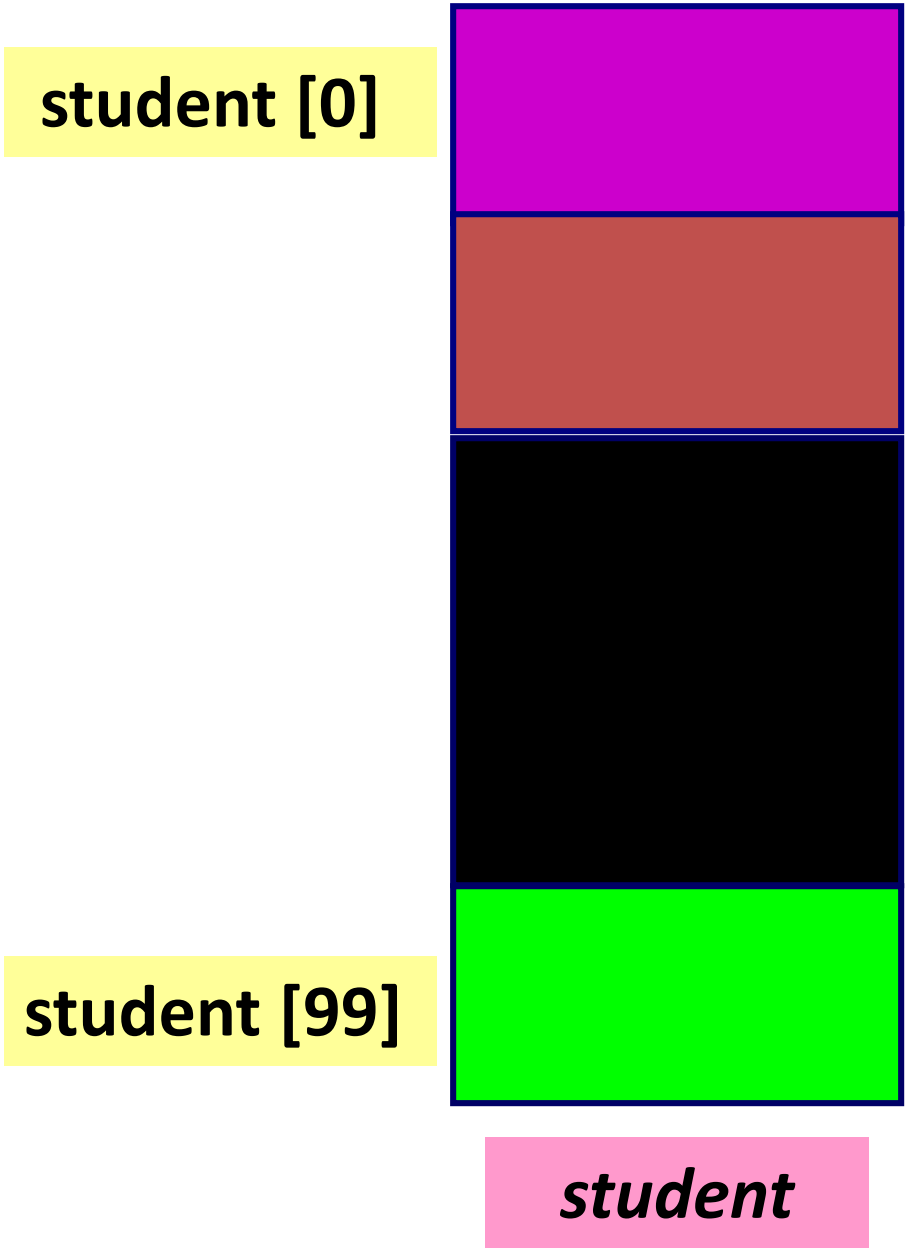
Use class's

```
// assign student1 to student3
student3 = student1;
```

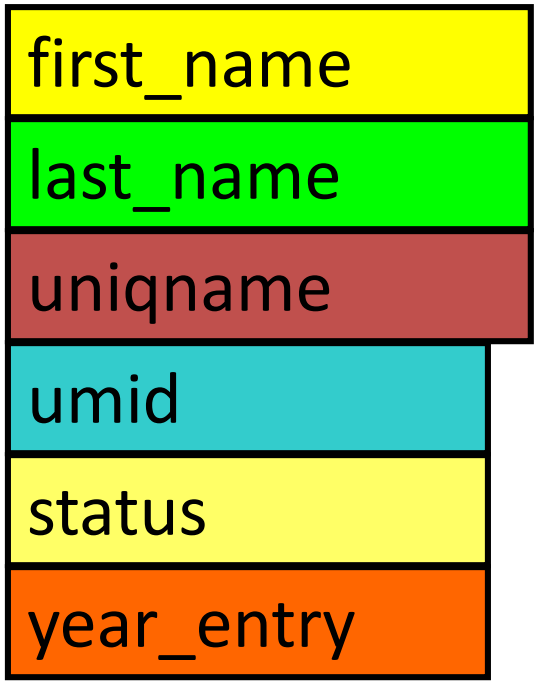
Declaring an array of `class`'s

```
const int NUM_STUDENTS = 100;

Student students[NUM_STUDENTS];
```

the array



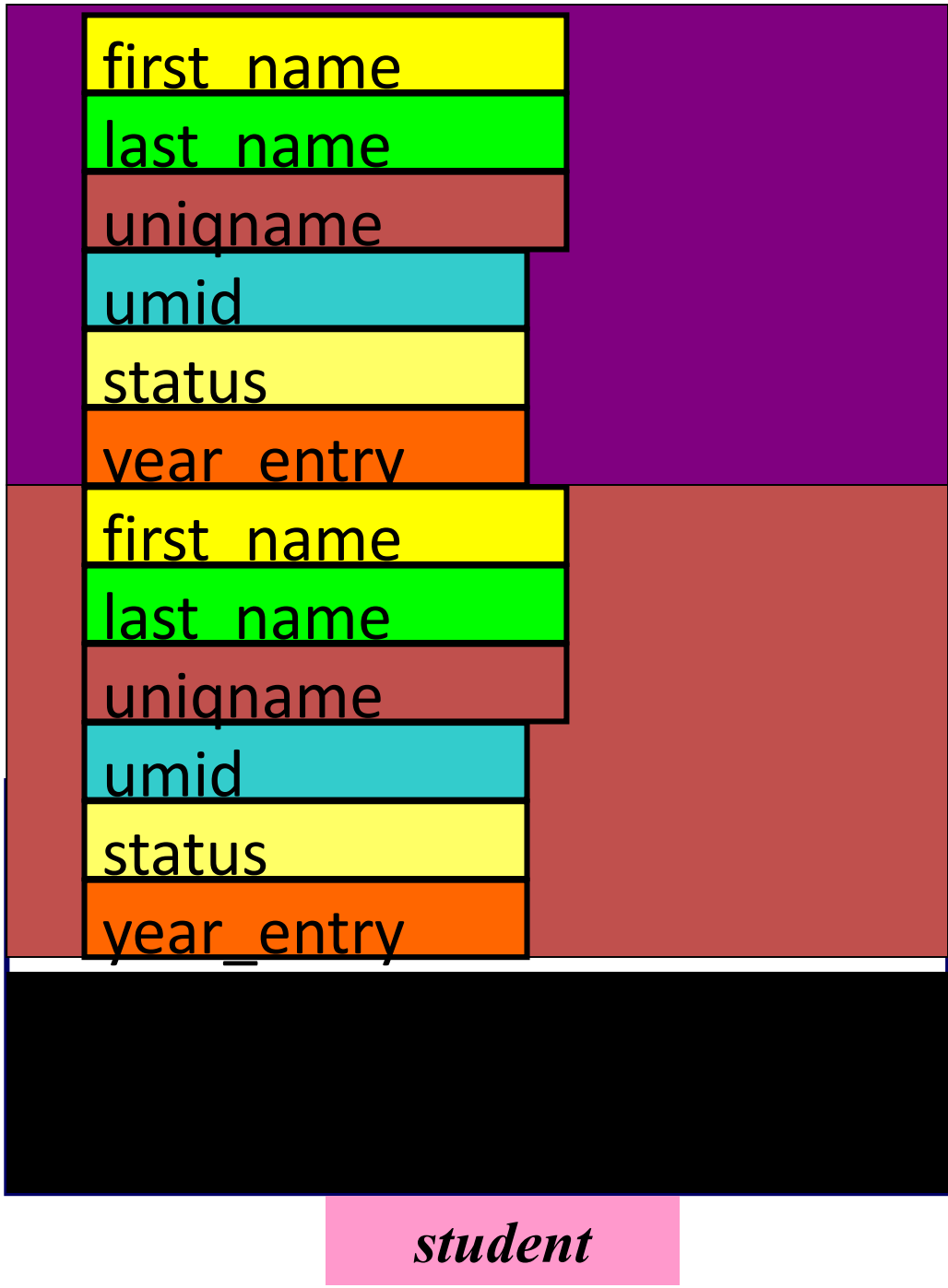
one "Student"



the array

student [0]

student [1]

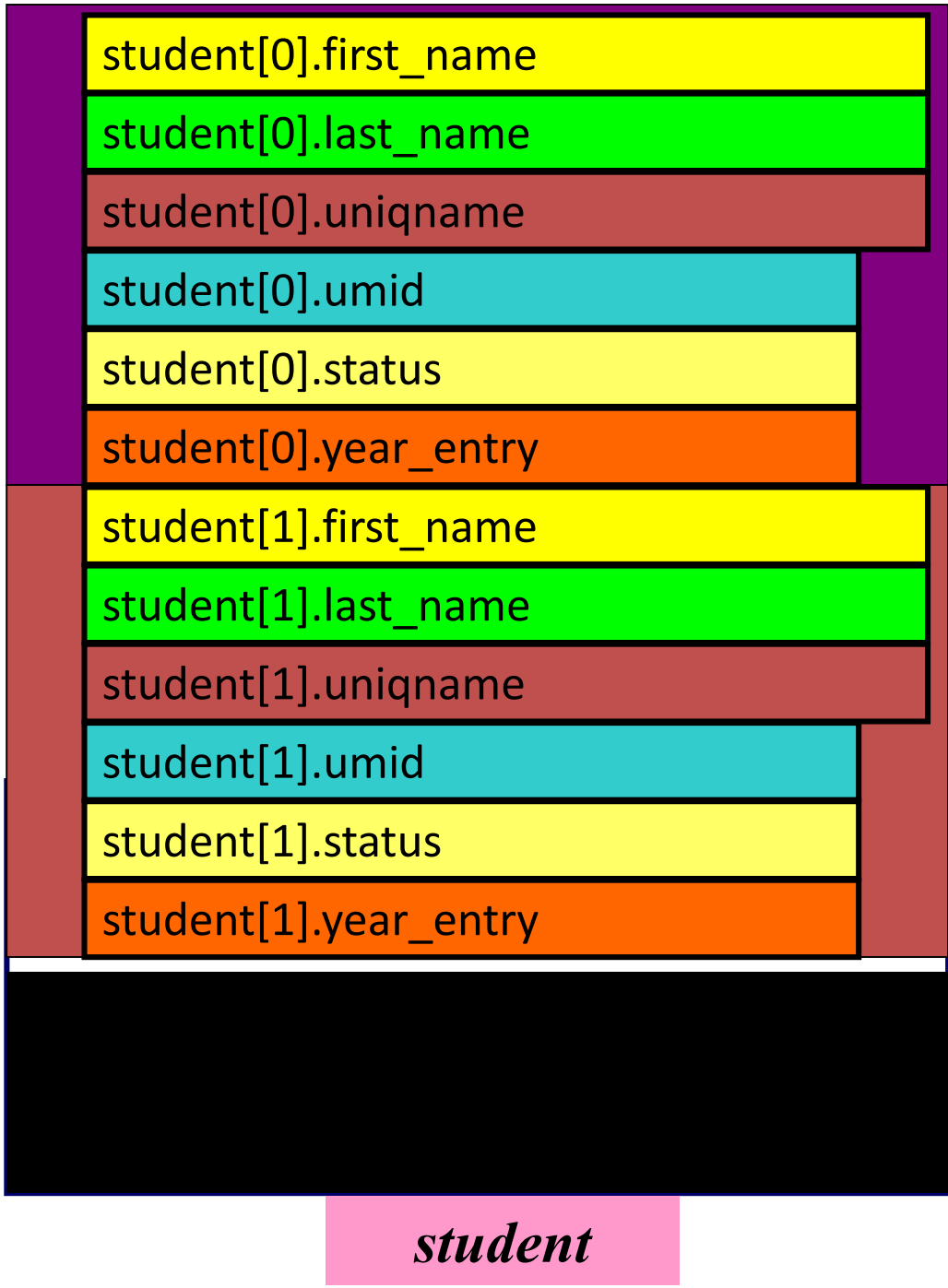


the array

student [0]

student [1]

access using the dot operator

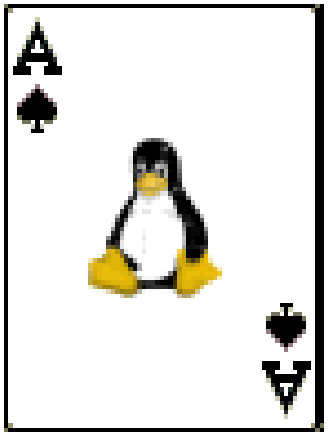


Why Structured Types

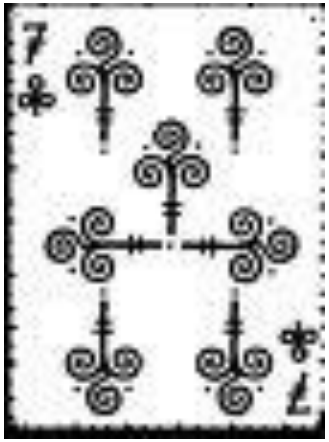
- Group items together

- Cards:
  - Rank
    - A, K, Q, J, 10, 9, 8, 7, 6, 5, 4, 3, 2
  - Suit
    - Clubs, Diamonds, Hearts, Spades

Why Structured Types



Rank: A
Suit: Spades

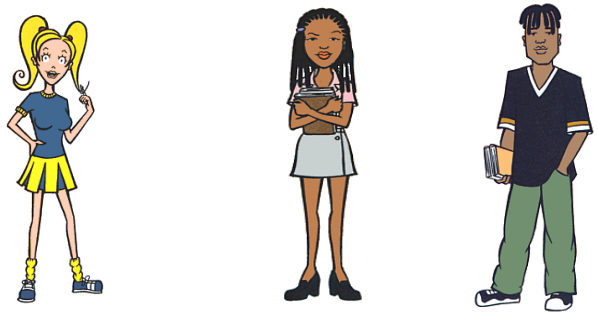


Rank: 7
Suit: Clubs

Rank: King
Suit: Hearts

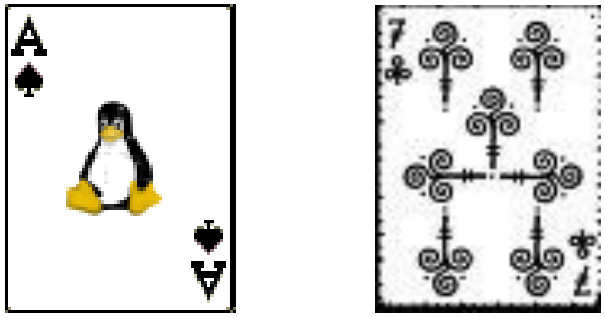
Classes vs. Instances

- Every student has:
  - A first name
  - A last name
  - A uniqname
  - …



Specific students

- Every playing card has:
  - A rank (Ace - King)
  - A suit (♠ ♥ ♣ ♦)

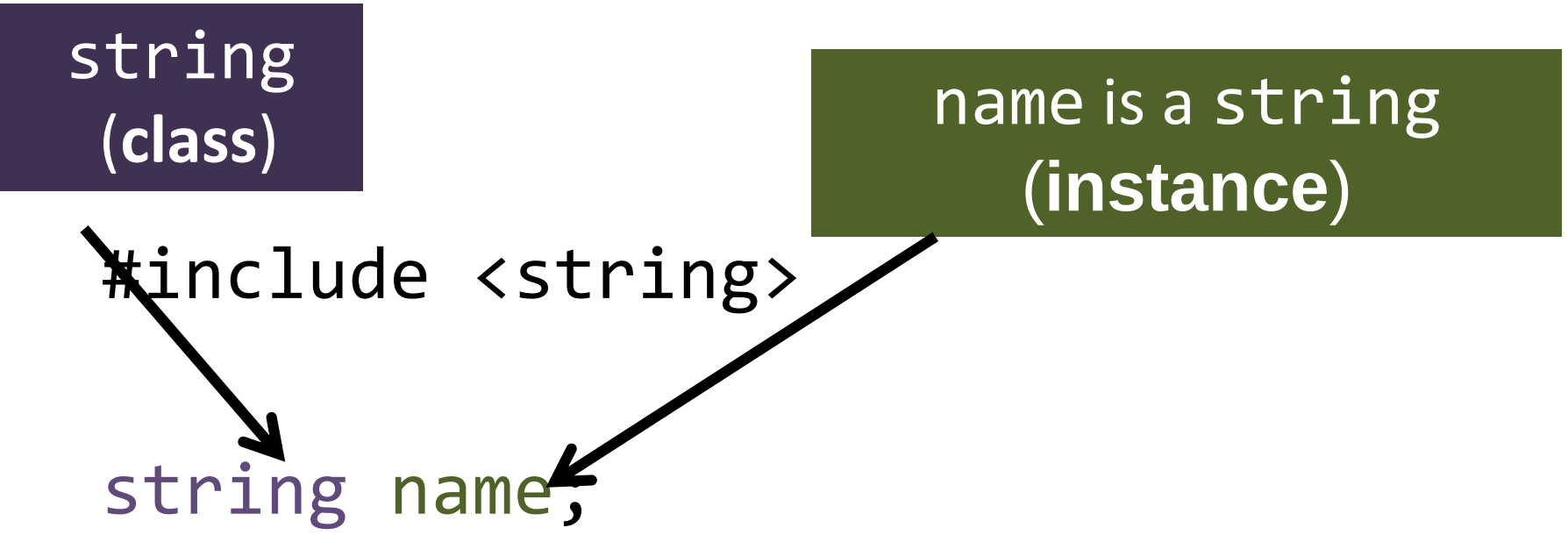


Specific cards

Familiar Classes

```
#include <string>

string name;
```

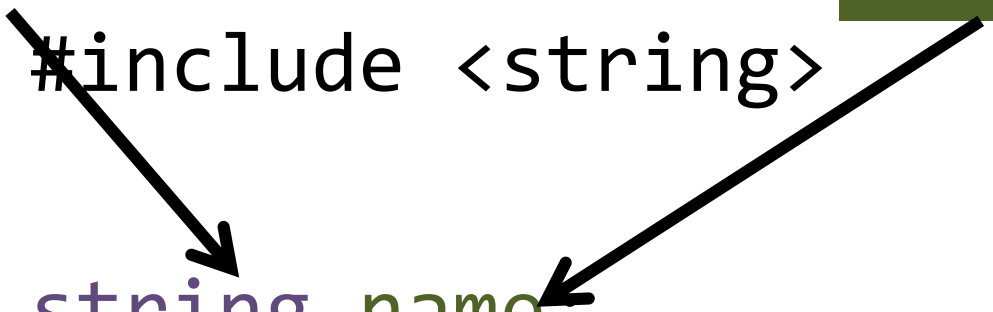
Familiar Classes



Familiar Classes

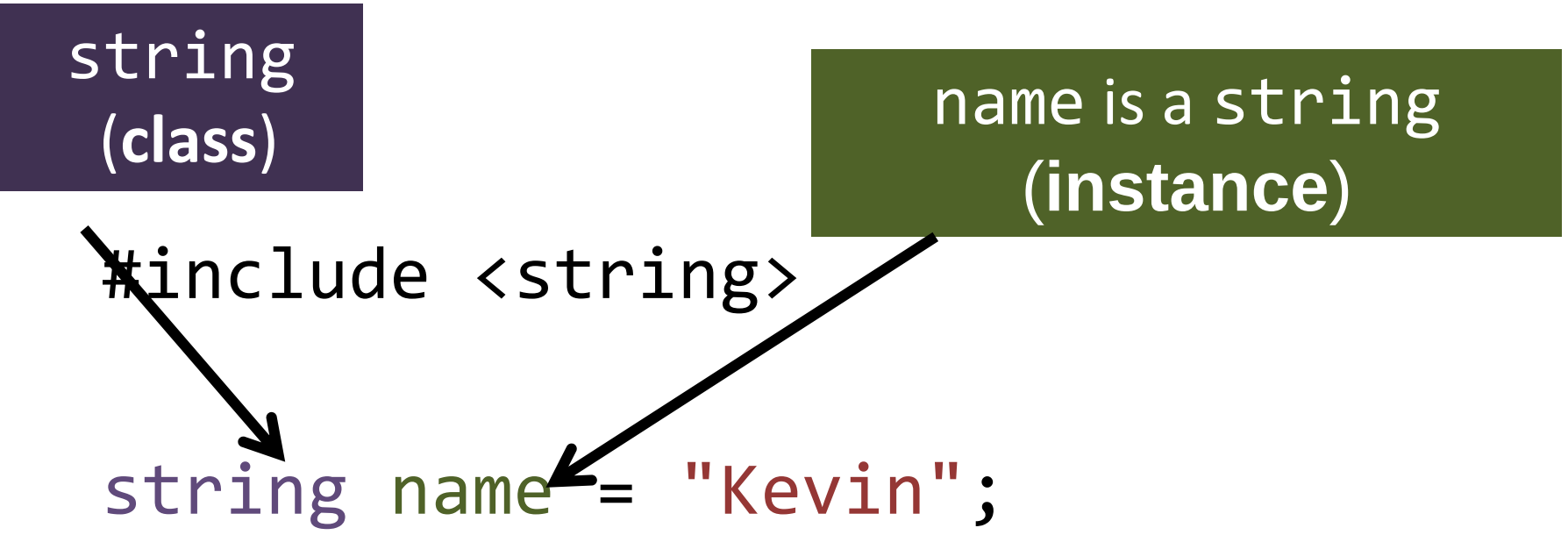
tells compiler how to allocate

actual space in memory

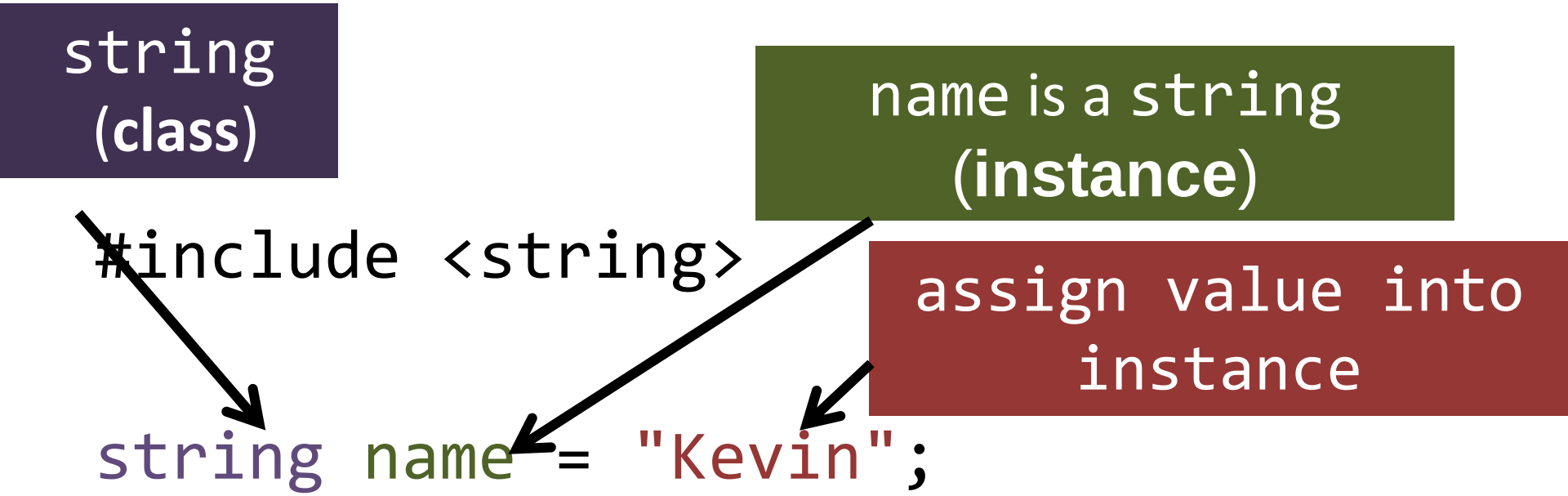


```
#include <string>

string name;
```

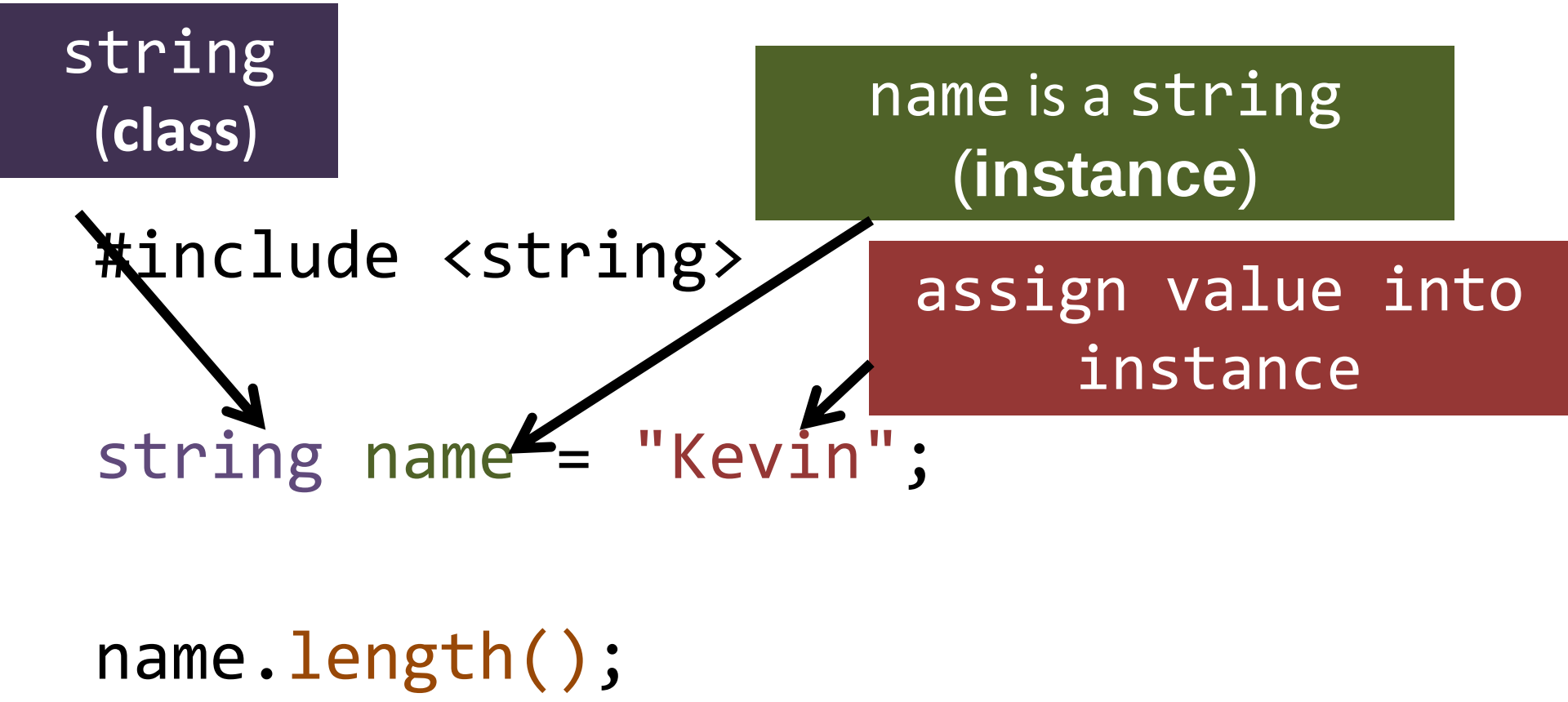
Familiar Classes



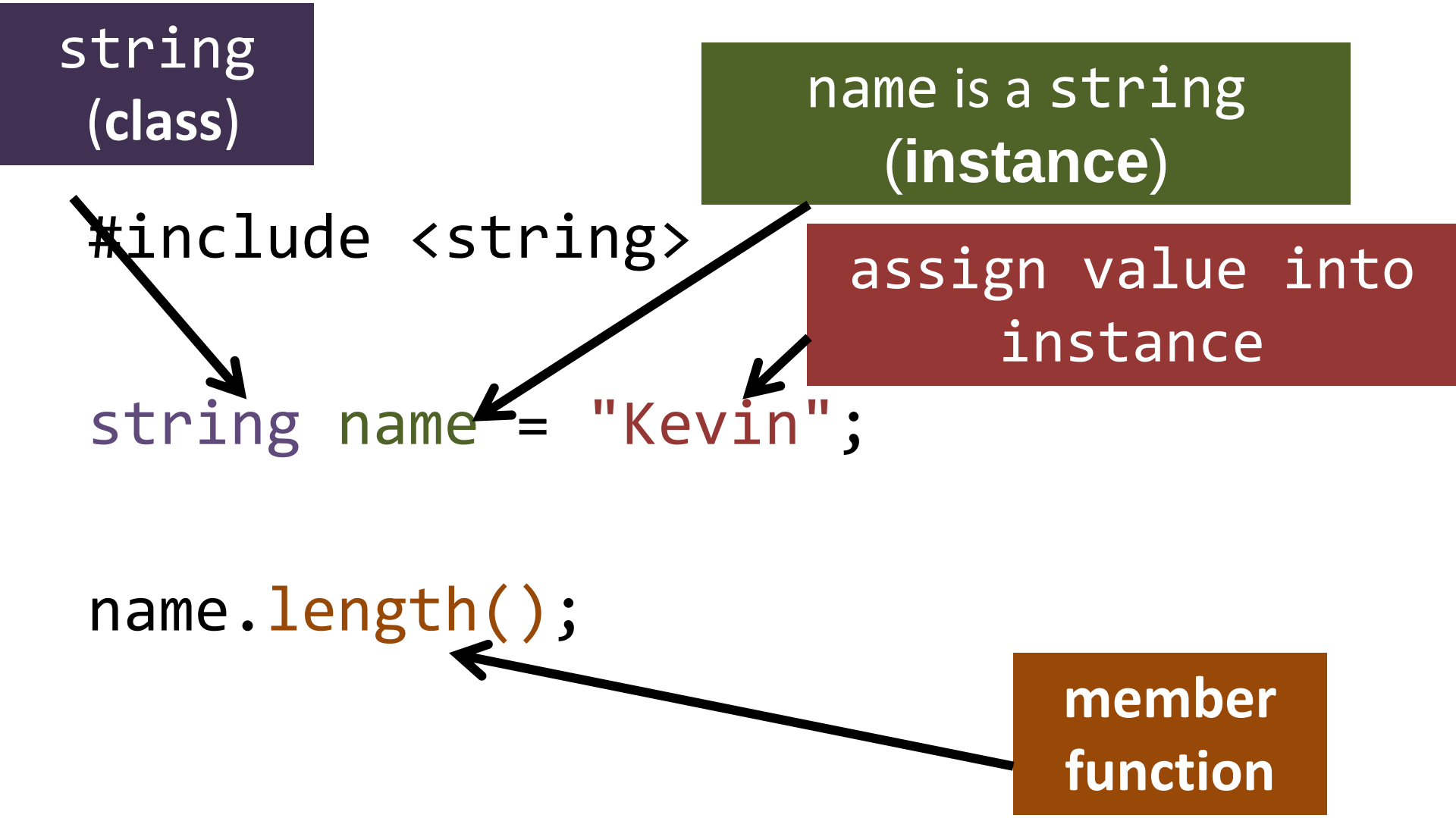
Familiar Classes



Familiar Classes



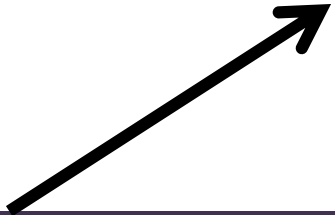
Familiar Classes



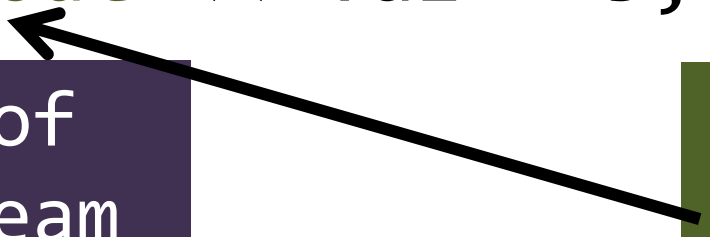
Familiar Classes

- You have already seen some classes

```
#include <iostream>

int val;
cin >> val;
cout << val + 5;
```



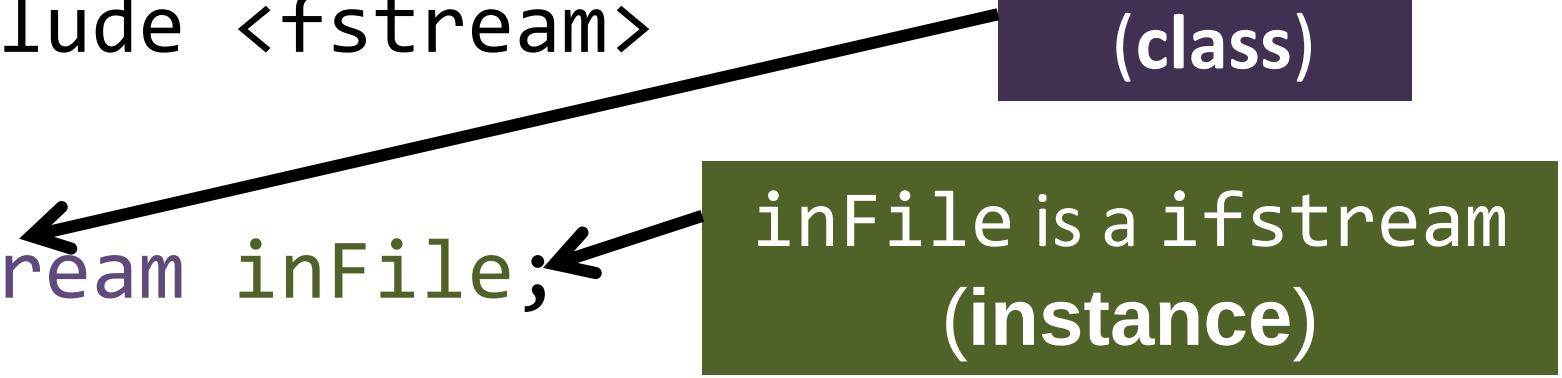
instance of class istream



instance of class ostream

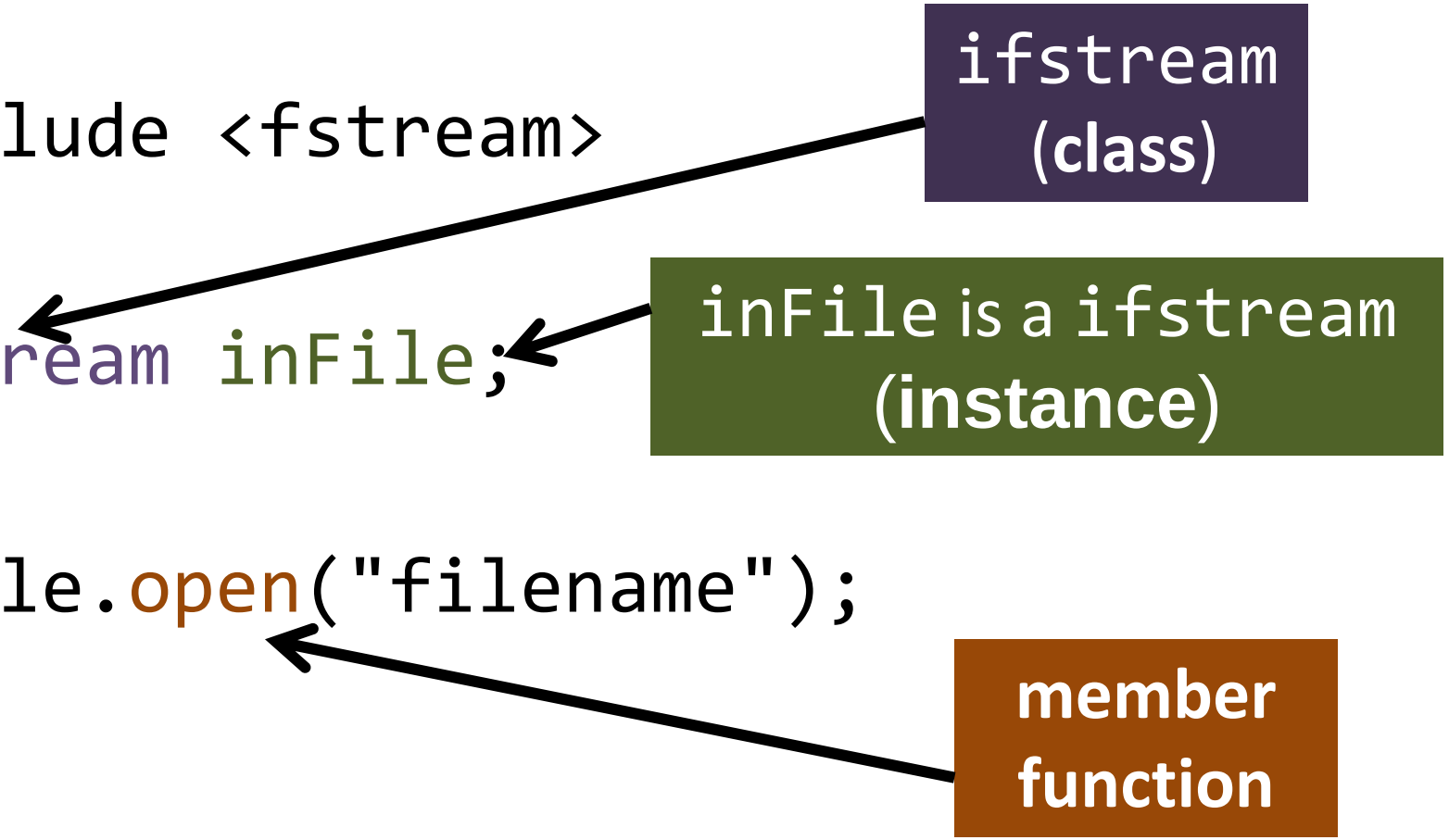
Familiar Classes

```
#include <fstream>

ifstream inFile;
```



Familiar Classes

```
#include <fstream>

ifstream inFile;

inFile.open("filename");
```



So much New Terminology

instance

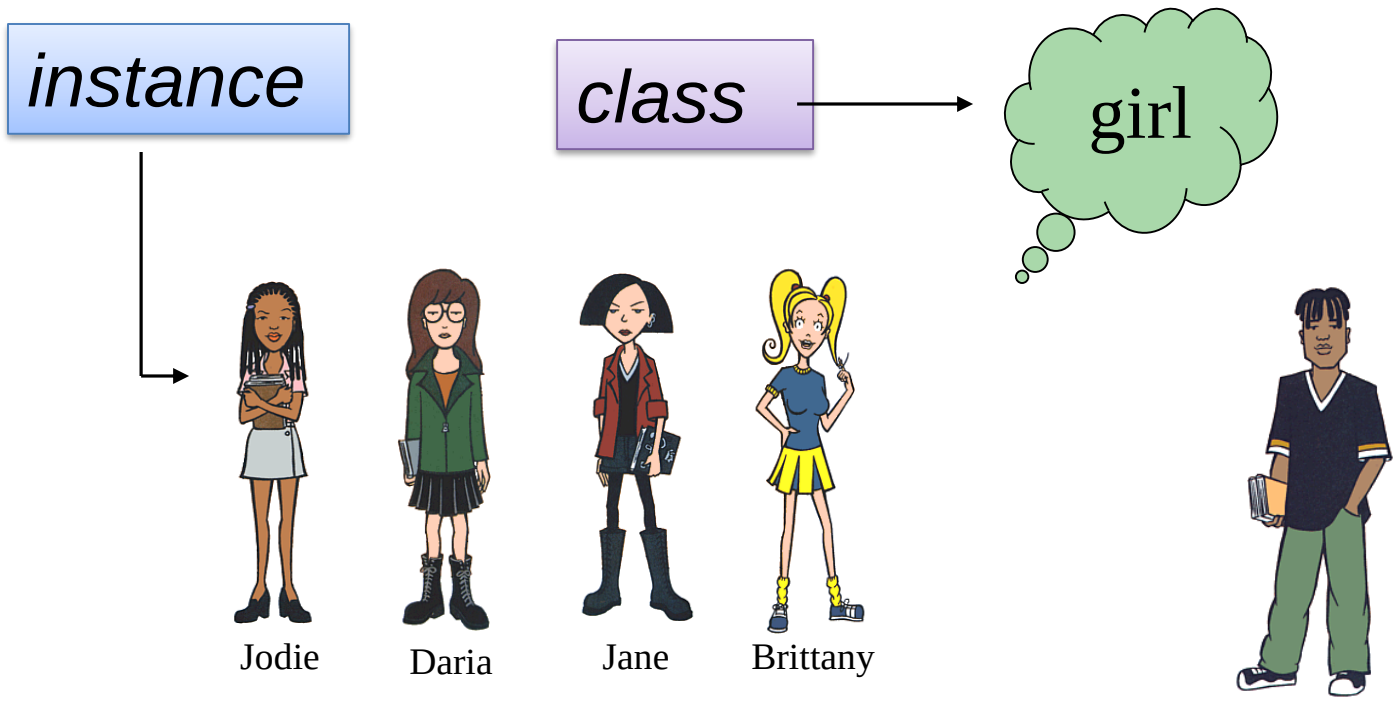
class

object

member data

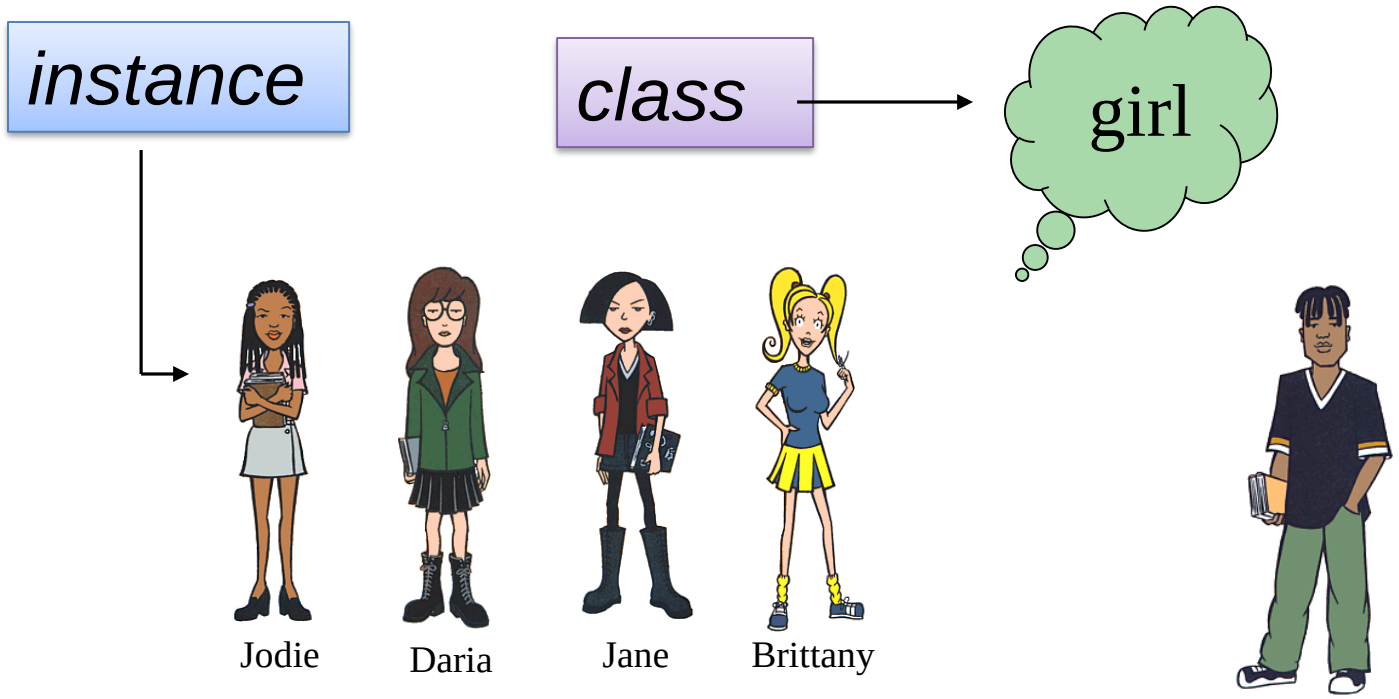
member function

class vs. instance of class



class vs. instance of class

- Classes reflect concepts
- Instance is a single and unique unit of the class



class vs. instance of class

class

class vs. instance of class

class

Instances of house described by blueprint

Classes vs. Instances

- In general, if you can say "A is a B", then A is an instance of class B
- Examples:
  - Maddy is a Student
  - The Ace of Spades is a Playing Card
  - My Ford is a Car
  - The University of Michigan is a School
- Can you think of some other ones?

Classes vs. Instances

- Be careful! If you can say “all As are Bs”, then A is probably NOT an instance of class B
- Examples (NOT instances):
  - Government is an Organization
  - Sheep is an Animal
  - Honda is a Car
  - University is a School

- Can you think of some other ones?

i>Clicker #3

- Which of these is NOT an instance-class relationship?

A. Earth, Planet
B. The Diag, Location
C. USA, Country
D. Beagle, Dog

i>Clicker #3

- Which of these is NOT an instance-class relationship?

A. Earth, Planet
B. The Diag, Location
C. USA, Country
D. Beagle, Dog

Why Classes?

1. **Keeps related data and functionality together - *encapsulation***

- Cars don't need `first_names`
- Only `ifstreams` and `ofstreams` need `open()`
  - Does not make sense to do `cin.open()`

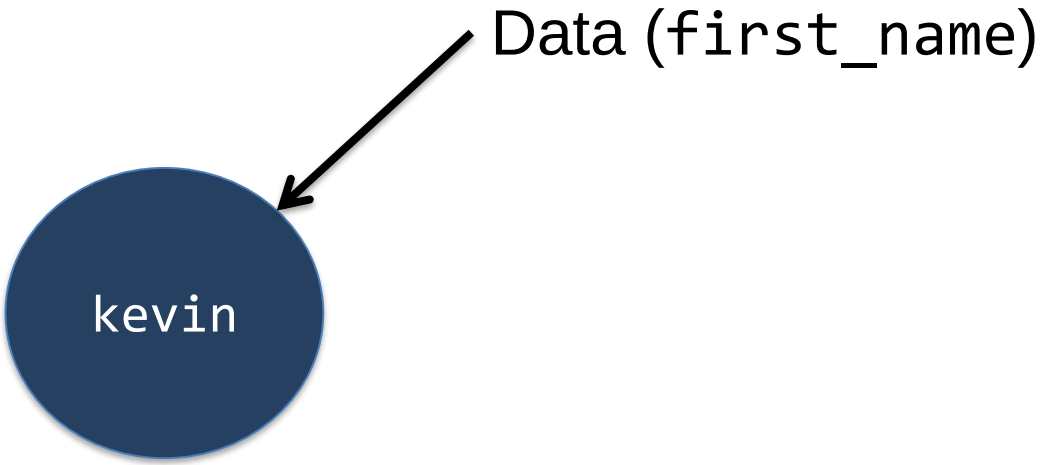
Why Classes?

2. Separates interface from implementation

- You can drive a car without knowing how the engine works
- Professors can ask for my GPA without being able to change my name

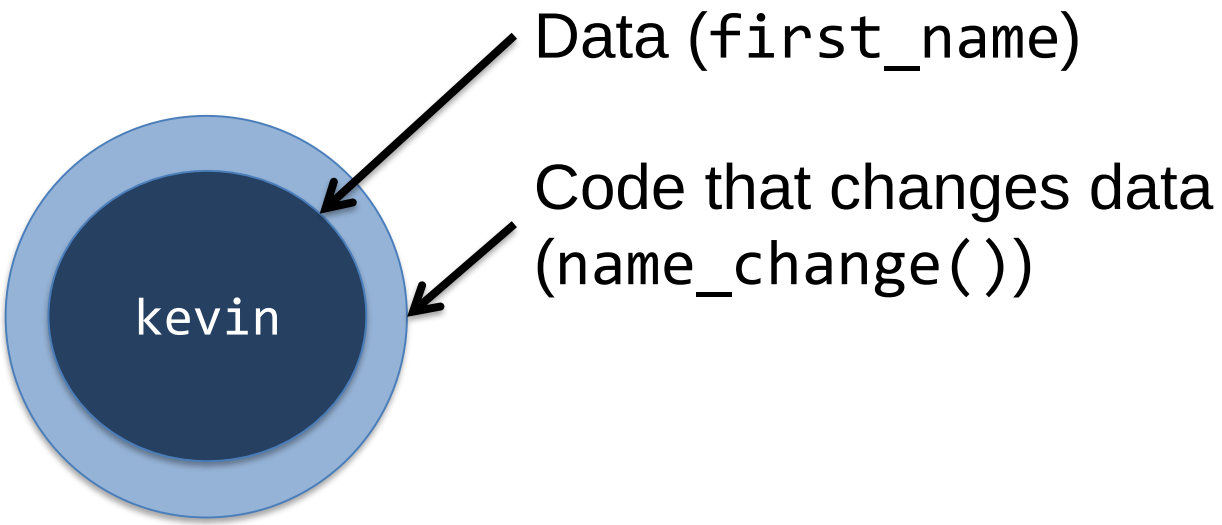
Why Classes?

2. Separates interface from implementation



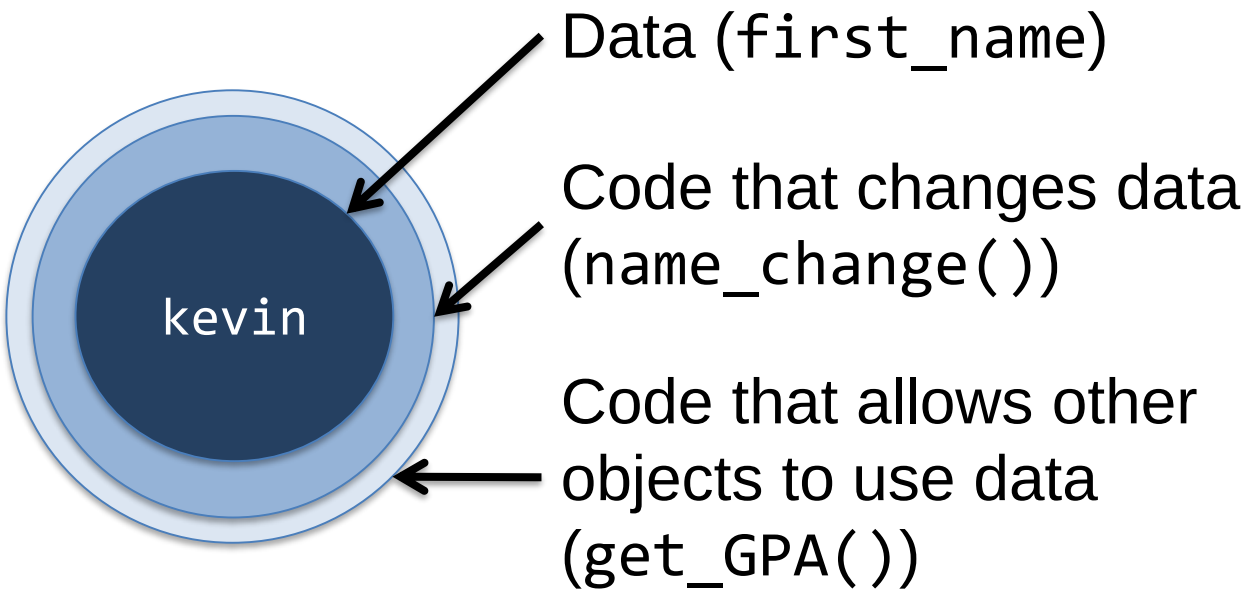
Why Classes?

2. Separates interface from implementation



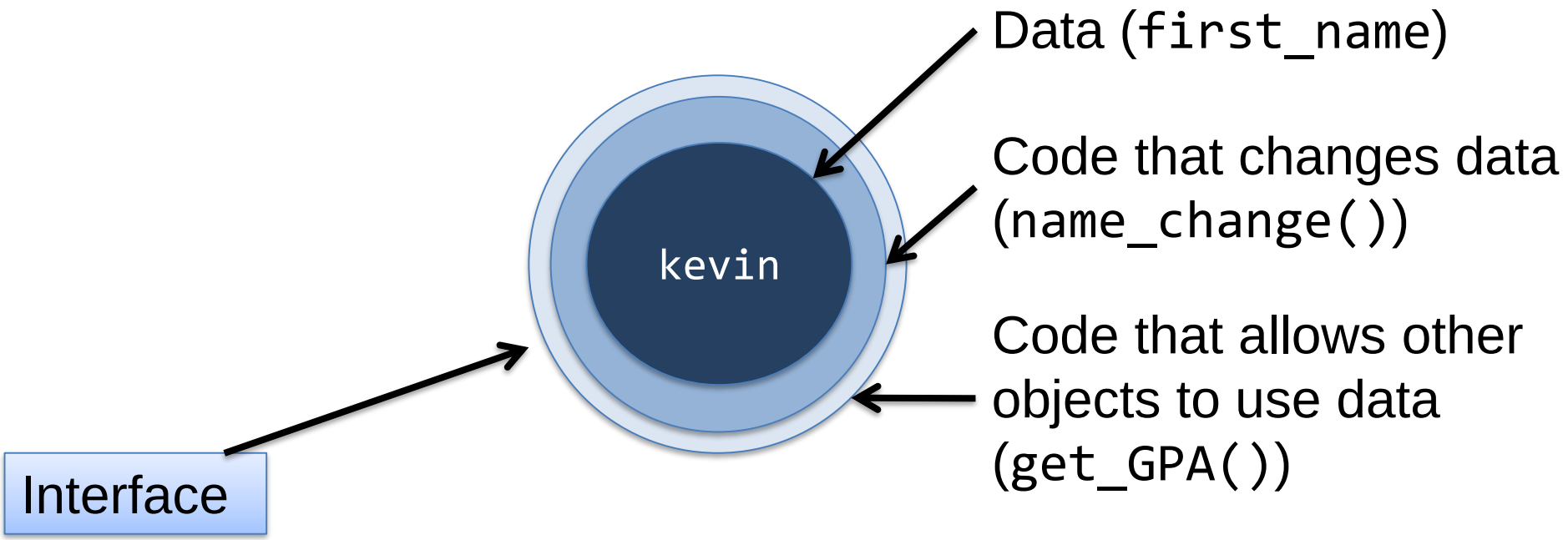
Why Classes?

2. Separates interface from implementation



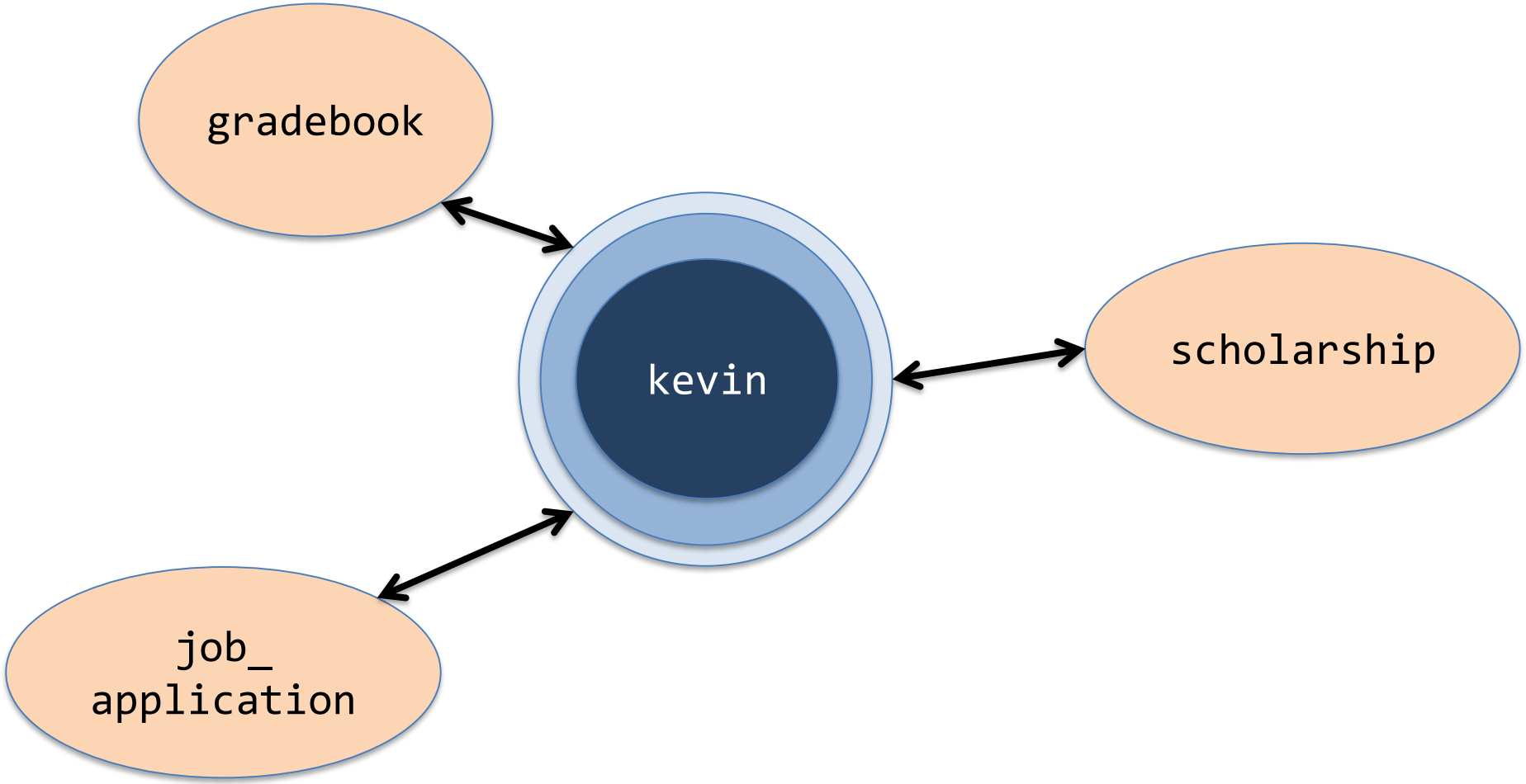
Why Classes?

2. Separates interface from implementation



Why Classes?

2. Separates interface from implementation



Why Classes?

3. Avoids code duplication

- Undergraduate, Graduate all have a `get_GPA()` function
- We can just say that all Students have a `get_GPA()` function
- This is called *inheritance*, which you don't have to know for 183

Summary

- **Classes** are a way to group heterogeneous data
  - Together with relevant functions

- We can have one class (`Student`) and multiple **instances** (`kevin_lee`, `maddy_endres`, etc.)

- Benefits of classes:
  1. Keeps related data and functionality together
  2. Separate interface from implementation
  3. Avoid code duplication

Intermission

A program that uses lots of classes and instances is called *object-oriented*

The Details

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

The De

This is a C++ keyword, to signify a class definition.

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

The D

This class is called Student; by convention, class names start in Upper Case

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

T

This is another C++ keyword. This means the following members can be read/written by other objects.

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

All students have a `first_name`, which is a `string`. This is called a **member variable**

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

The D

All students also have a last_name, which is also a string.

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

The

Note the semicolon. This is actually a statement telling C++ that a Student class exists

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

The D

Create an *instance* of Student called kevin.

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

Set the first_name and last_name of kevin to "Kevin" and "Lee" respectively.

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

Get the first_name of kevin.

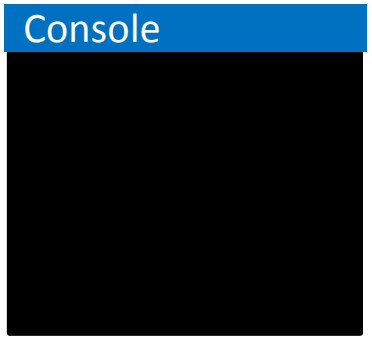
The Details

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

The Details

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

Console



The Details

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

Console

```
Kevin Lee
```

The Details

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student kevin;
    kevin.first_name = "Kevin";
    kevin.last_name = "Lee";
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

Console

```
Kevin Lee
```

i>Clicker #4

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student helen = {"Maddy", "Endres"};
    helen.first_name = "Kevin";
    helen.last_name = "Lee";
    cout << helen.first_name << " "
         << helen.last_name << endl;
}
```

What prints?

A. Helen Hagos
B. Maddy Endres
C. Kevin Lee
D. Something else
E. Compile error

i>Clicker #4

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student helen = {"Maddy", "Endres"};
    helen.first_name = "Kevin";
    helen.last_name = "Lee";
    cout << helen.first_name << " "
         << helen.last_name << endl;
}
```

What prints?

A. Helen Hagos
B. Maddy Endres
C. Kevin Lee
D. Something else
E. Compile error

The Details

```
class Student {
    public:
        string first_name;
        string last_name;
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

Console

```
Kevin Lee
```

Member Functions

This **member function** is just like the functions you already know

```
class Student {
    public:
        string first_name;
        string last_name;
        void print_name() {
            cout << first_name << " "
                 << last_name << endl;
        }
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    kevin.print_name();
}
```

Member Functions

```
class Student {
    public:
        string first_name;
        string last_name;
        void print_name() {
            cout << first_name << " "
                 << last_name << endl;
        }
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    kevin.print_name();
}
```

But it's called with a '.', like how we call `inFile.open("filename");`

Member Functions

Notice that first_name can be used directly, since it's in the *scope* of the Student class

```
class Student {
    public:
        string first_name;
        string last_name;
        void print_name() {
            cout << first_name << " "
                 << last_name << endl;
        }
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    kevin.print_name();
}
```

Member Functions

```
class Student {
    public:
        string first_name;
        string last_name;
        void print_name() {
            cout << first_name << " "
                 << last_name << endl;
        }
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    kevin.print_name();
}
```

Console

```
Kevin Lee
```

Public and Private Members

```
class Student {
    public:
        string first_name;
        string last_name;
        void print_name() {
            cout << first_name << " "
                 << last_name << endl;
        }
};

int main() {
    Student kevin = {"Kevin", "Lee"};
    kevin.first_name = "Maximilian";
    kevin.print_name();
}
```

Console

```
Maximilian Lee
```

Public and Private Members

```
class Student {
    public:
        string first_name;
        string last_name;
        double gpa;
};

int main() {
    Student kevin = {"Kevin", "Lee", 1.3};

    cout << kevin.gpa << endl;
}
```

Console

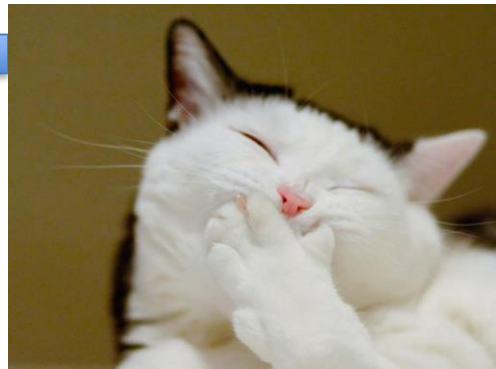
```
1.3
```

Public and Private Members

```
class Student {
    public:
        string first_name;
        string last_name;
        double gpa;
};

int main() {
    Student kevin = {"Kevin", "Lee", 1.3};
    kevin.gpa = 4.0;
    cout << kevin.gpa << endl;
}
```

Console

```
4.0
```



Public and Private Members

Private is another C++ keyword.

This means the following members **CANNOT** be seen by other objects.

```
class Student {
    private:
        string first_name;
        string last_name;
        double gpa;
};

int main() {
    Student kevin = {"Kevin", "Lee", 1.3};
    kevin.gpa = 4.0;
    cout << kevin.gpa << endl;
}
```

Public and Private Members

```
class Student {
    private:
        string first_name;
        string last_name;
        double gpa;
};

int main() {
    Student kevin = {"Kevin"
    kevin.gpa = 4.0;
    cout << k
}
```

Compile Error

nd

Constructors

```
class Student {
    private:
        string first_name;
        string last_name;
        double gpa;
};

int main() {
    Student kevin = {"Kevin", "Lee", 1.3};

    cout << kevin.gpa << endl;
}
```

Constructors

Constructors

- A **Constructor** is a member function that **has the same name as the class** and **no return type**.

- It's what creates the instance of the class.

Constructors

```
class Student {
    public:
        Student(string first, string last) {
            first_name = first;
            last_name = last;
        }
    private:
        string first_name;
        string last_name;
};
int main() {
    Student kevin("Kevin", "Lee");
}
```

This constructor sets the first and last name of a student.

Constructors

We “call” the function when we initialize kevin.

```
class Student {
    public:
        Student(string first, string last) {
            first_name = first;
            last_name = last;
        }
    private:
        string first_name;
        string last_name;
};
int main() {
    Student kevin("Kevin", "Lee");
}
```

Getters

```
class Student {
    public:
        Student(string first,
                string last) {
            first_name = first;
            last_name = last;
        }
    private:
        string first_name;
        string last_name;
};
int main() {
    Student kevin("Kevin", "Lee");
    cout << kevin.first_name << " "
         << kevin.last_name << endl;
}
```

Getters

```
class Student {
    public:
        Student(string first,
                string last) {
            first_name = first;
            last_name = last;
        }
        void print_name() {
            cout << first_name << " "
                 << last_name << endl;
        }
    private:
        string first_name;
        string last_name;
};
```

Get

Member functions can always access private member variables

```
class Student {
    public:
        Student(string first,
                string last) {
            first_name = first;
            last_name = last;
        }
        void print_name() {
            cout << first_name << " "
                 << last_name << endl;
        }
    private:
        string first_name;
        string last_name;
};
```

Getters

A **getter** is a function that returns the value in a private variable.

```
class Student {
    public:
        Student(string first,
                string last) {
            first_name = first;
            last_name = last;
        }
        string get_first_name() {
            return first_name;
        }
    private:
        string first_name;
        string last_name;
};
```

Getters

A **getter** is a function that returns the value in a private variable.

```
class Student {
    public:
        Student(string first,
                string last) {
            first_name = first;
            last_name = last;
        }
        string get_first_name() {
            return first_name;
        }
        string get_last_name() {
            return last_name;
        }
    private:
        string first_name;
        string last_name;
};
```

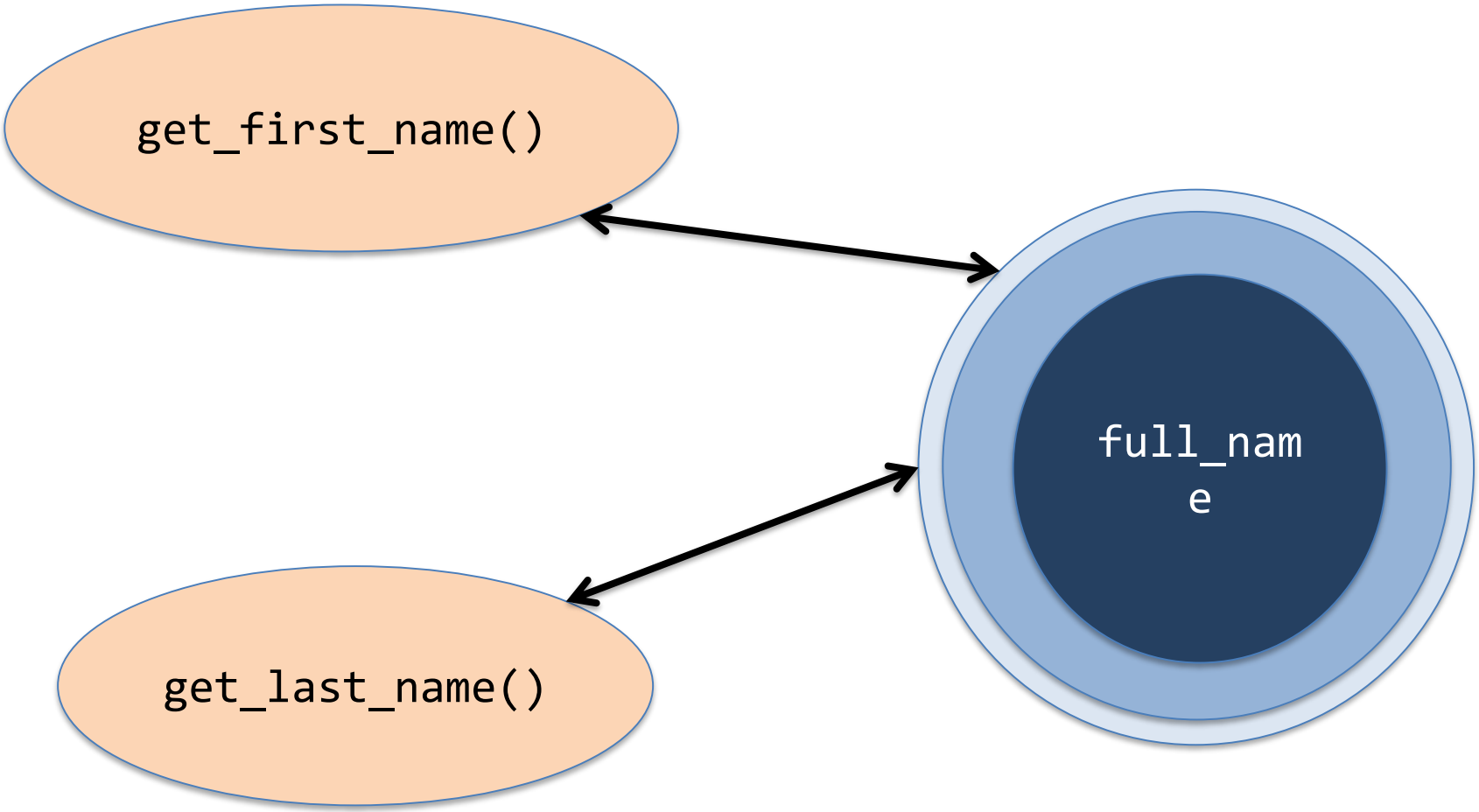
Getters

```
class Student {
    public:
        ...
        string get_first_name() {
            return first_name;
        }
        string get_last_name() {
            return last_name;
        }
    ...
};

int main() {
    Student kevin("Kevin", "Lee");
    cout << kevin.get_first_name() << " "
         << kevin.get_last_name() << endl;
}
```

Console

```
Kevin Lee
```

Visualization



i>Clicker #5

```
class Student {
    public:
        Student(string name,
                int g) {
            full_name = name;
            gpa = g;
        }
    private:
        string full_name;
        int gpa;
};

int main() {
    Student maddy("Madeline Endres", 4.0);
    maddy.gpa = 3.5;
    cout << maddy.gpa << endl;
}
```

What does this program do?

A. Prints 4.0
B. Prints 3.5
C. There will be a compile error
D. There will be a runtime error
E. None of the above

i>Clicker #5

```
class Student {
    public:
        Student(string name,
                int g) {
            full_name = name;
            gpa = g;
        }
    private:
        string full_name;
        int gpa;
};

int main() {
    Student maddy("Madeline Endres", 4.0);
    maddy.gpa = 3.5;
    cout << maddy.gpa << endl;
}
```

What does this program do?

A. Prints 4.0
B. Prints 3.5
C. There will be a compile error
D. There will be a runtime error
E. None of the above

i>Clicker #6

```
class Student {
    public:
        Student(string name,
                int g) {
            full_name = name;
            gpa = g;
        }
    private:
        string full_name;
        int gpa;
        void print_gpa() {
            cout << full_name << ": " << gpa << endl;
        }
};

int main() {
    Student maddy("Madeline Endres", 4.0);
    maddy.print_gpa();
}
```

What does this program do?

A. Prints 4.0
B. There will be a compile error
C. There will be a runtime error
D. None of the above

i>Clicker #6

```
class Student {
    public:
        Student(string name,
                int g) {
            full_name = name;
            gpa = g;
        }
    private:
        string full_name;
        int gpa;
        void print_gpa() {
            cout << full_name << ": " << gpa << endl;
        }
};

int main() {
    Student maddy("Madeline Endres", 4.0);
    maddy.print_gpa();
}
```

What does this program do?

A. Prints 4.0
B. There will be a compile error
C. There will be a runtime error
D. None of the above

Setters

Sometimes you'll also see **setters**, which change private variables.

```
class Student {
    public:
        ...
        string get_full_name() {
            return full_name;
        }
        void set_full_name(string name) {
            full_name = name;
        }
    private:
        string full_name;
};

int main() {
    Student kevin;
    kevin.set_full_name("Kevin Lee");
}
```

Setters

If you have both getters and setters, why not just make the variable public?

```
class Student {
    public:
        ...
        string get_full_name() {
            return full_name;
        }
        void set_full_name(string name) {
            full_name = name;
        }
    private:
        string full_name;
};

int main() {
    Student kevin;
    kevin.set_full_name("Kevin Lee");
}
```

Compiling Classes

- We can also separate the declaration of a class from its definition
  - This is another use of **.h** header files

Compiling Classes

In Student.h:

```
#include <string>
using namespace std;

class Student {
    public:
        Student(string first_name,
                string last_name);
        string get_full_name();
        void set_full_name(string name);
        void print_name();
    private:
        string full_name;
};
```

Compiling Classes

In `Student.cpp`:

```
#include <string>
#include <iostream>
#include "Student.h"
using namespace std;

Student::Student(string first_name,
                 string last_name) {
    full_name = first_name + " " + last_name;
}
string Student::get_full_name() {
    return full_name;
}
void Student::set_full_name(string name) {
    full_name = name;
}
void Student::print_name() {
    cout << full_name << endl;
}
```

Compiling Classes

In `Student.cpp`:

```
#include <string>
#include <iostream>
#include "Student.h"
using namespace std;

Student::Student(string first_name,
                 string last_name) {
    full_name = first_name + " " + last_name;
}
string Student::get_full_name() {
    return full_name;
}
void Student::set_full_name(string name) {
    full_name = name;
}
void Student::print_name() {
    cout << full_name << endl;
}
```

The `Student::` tells C++ that the function is part of the `Student` class.

Compiling Classes

In `main.cpp`:

```
#include <string>
#include <iostream>
#include "Student.h"
using namespace std;

int main() {
    Student helen("Helen", "Hagos");
    helen.print_name();
    cout << helen.get_full_name() << endl;
}
```

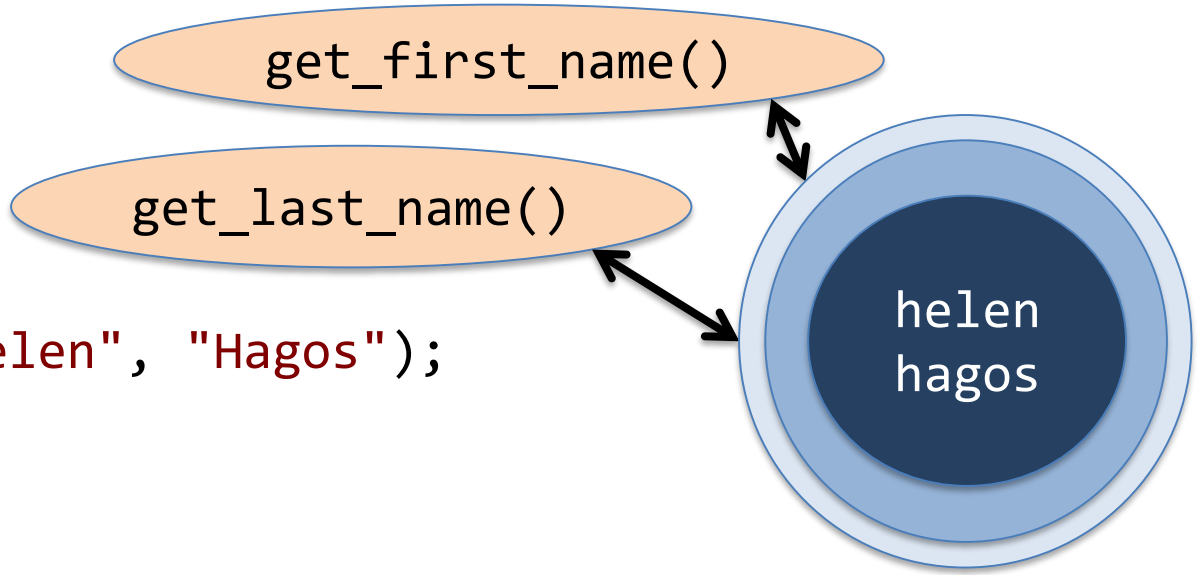
Console

```
Helen Hagos
Helen Hagos
```

Visualization

In `main.cpp`:

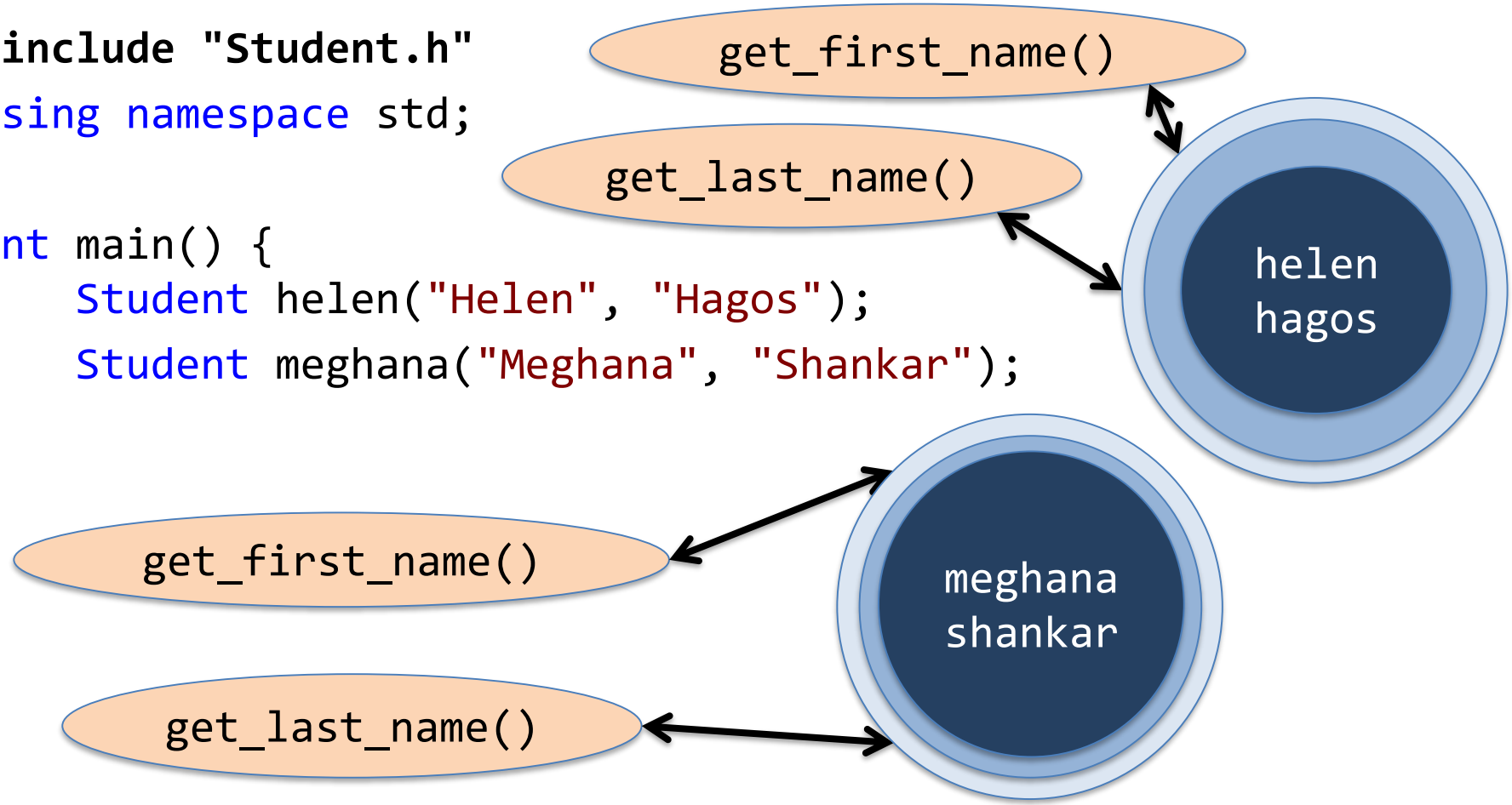
```
#include <string>
#include <iostream>
#include "Student.h"
using namespace std;

int main() {
    Student helen("Helen", "Hagos");
}
```



Visualization

In `main.cpp`:

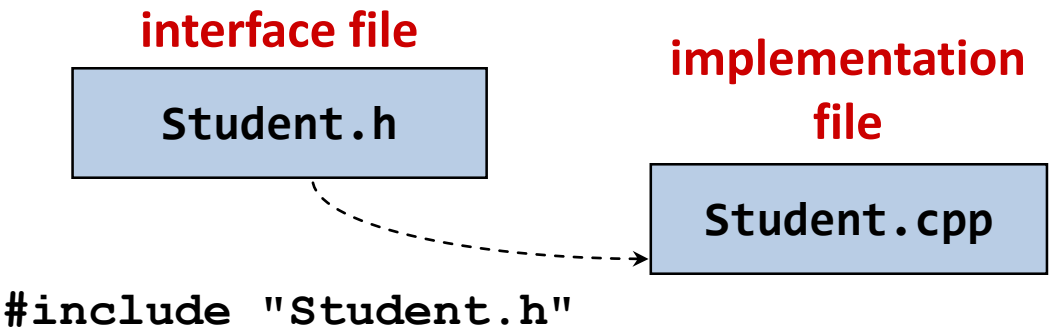
```
#include <string>
#include <iostream>
#include "Student.h"
using namespace std;

int main() {
    Student helen("Helen", "Hagos");
    Student meghana("Meghana", "Shankar");
}
```



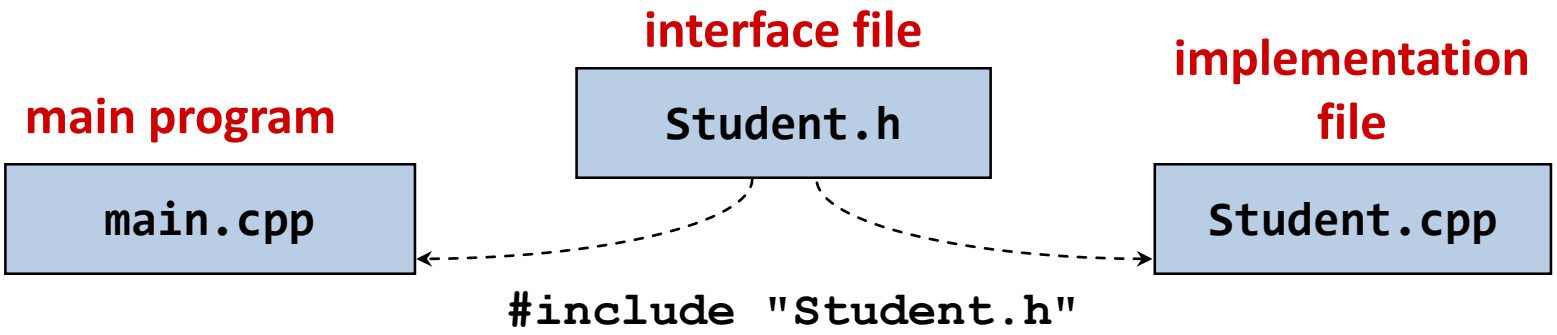
Separate Compilation and Linking of Files

interface file

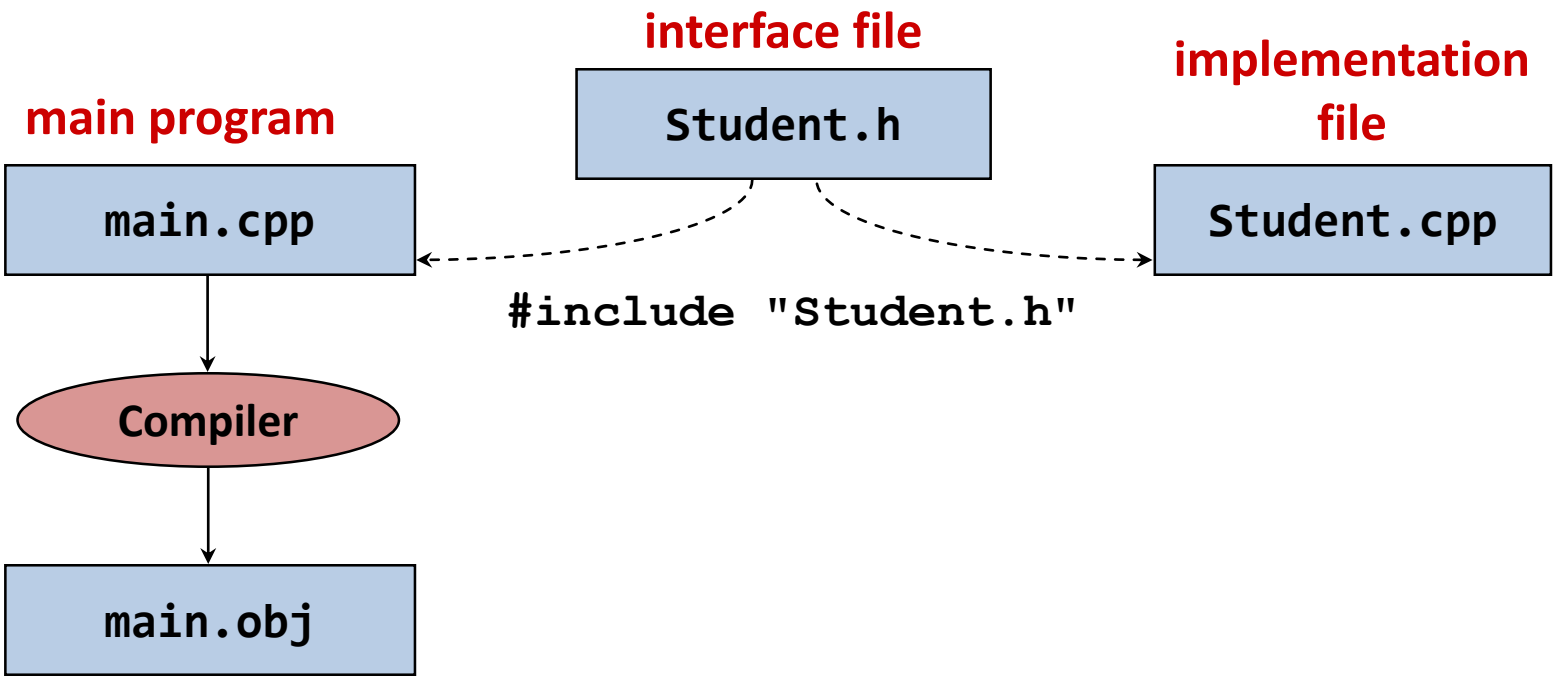
Separate Compilation and Linking of Files



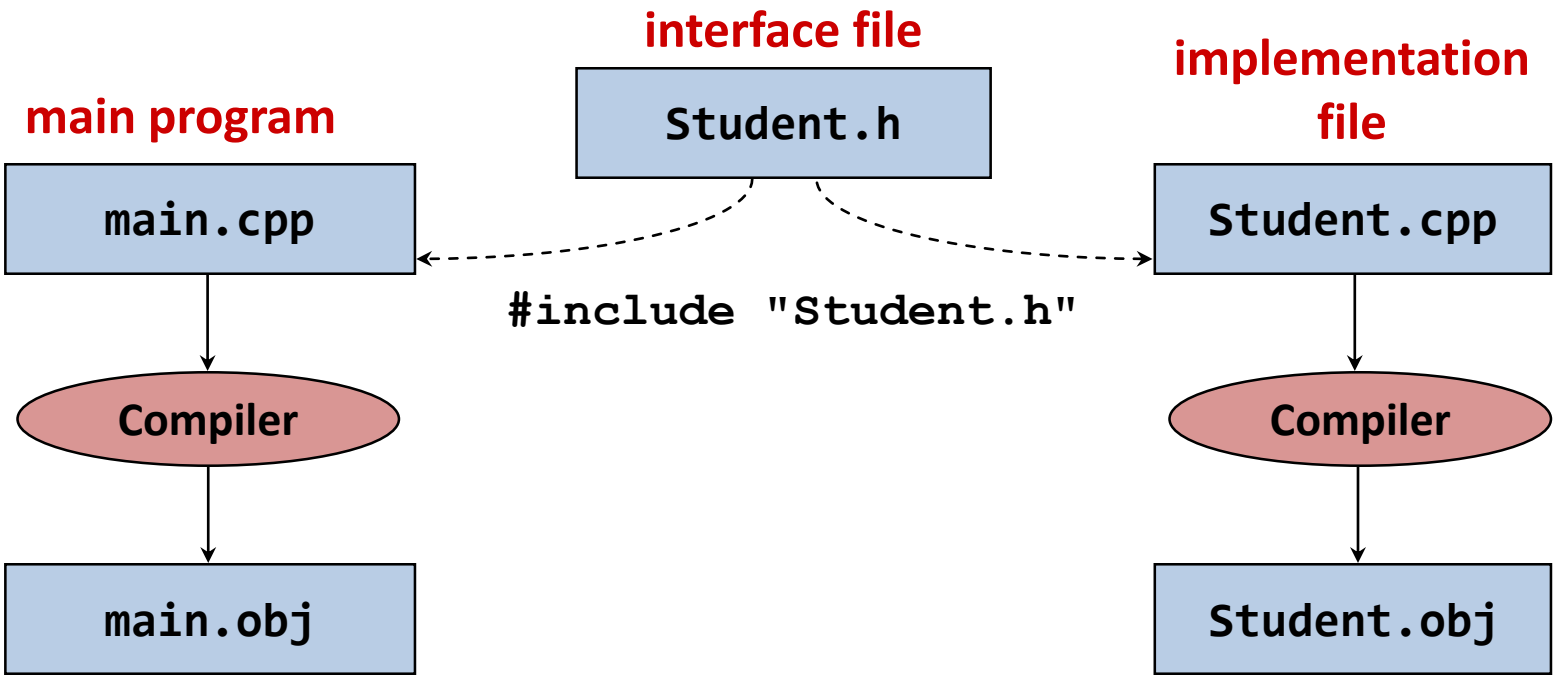
Separate Compilation and Linking of Files



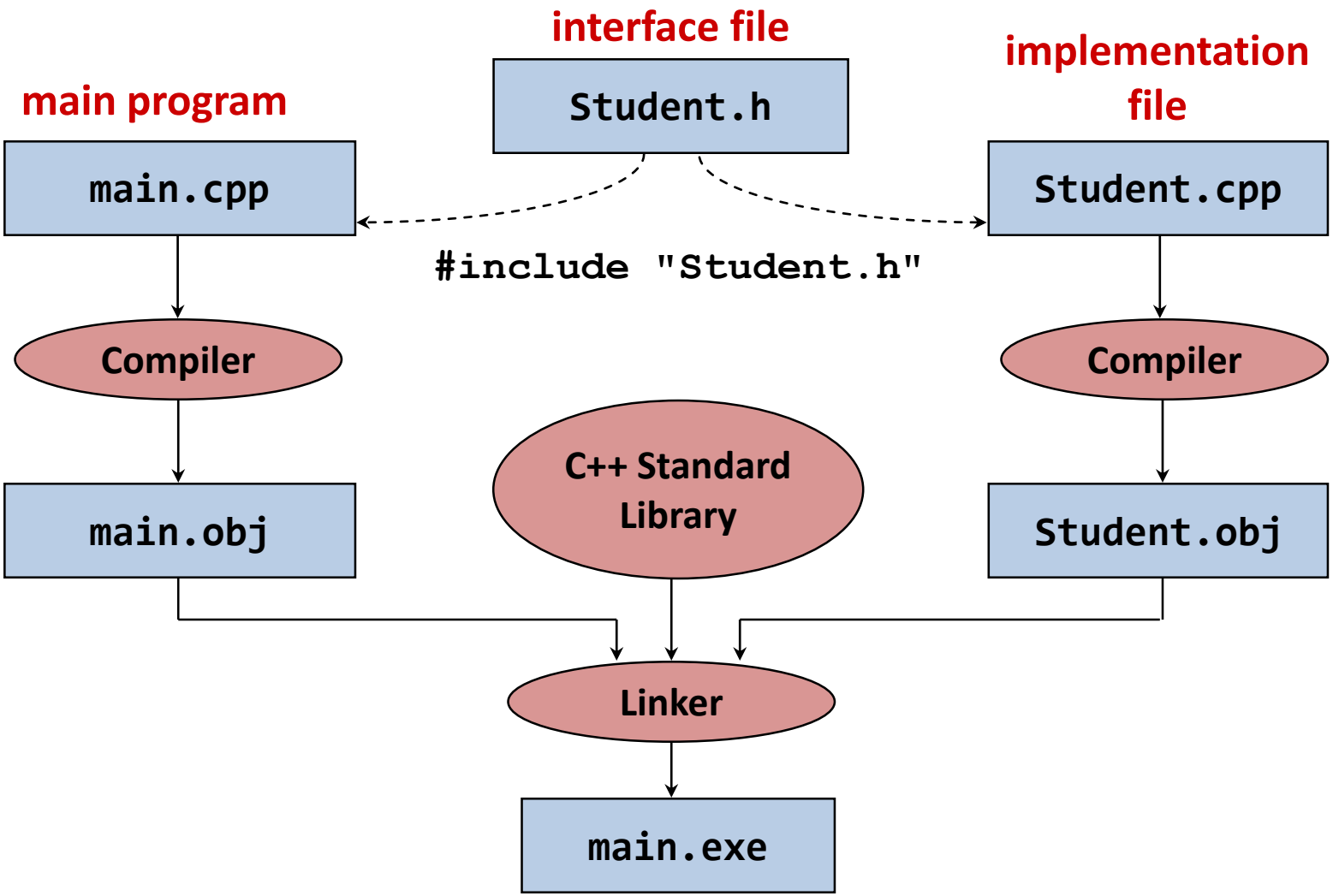
Separate Compilation and Linking of Files



Separate Compilation and Linking of Files



Separate Compilation and Linking of Files



Summary

- Classes have **public** and **private** members
- For private members, we need **constructors** to initialize an instance of the class
- We can then use **getters** and **setters** to change that data.
- We can separate a class into:
  - **declarations** (in the .h file) and
  - **definitions** (in the .cpp file)

Next Class: More Classes!

Member Functions

(Classes never end…)