CS 106B, Lecture 15
Classes and Stack Implementation

Plan for Today

- Continuing discussion of pointers from yesterday
- Arrays
- Classes in C++
- Putting it together: implementing Stack
- Templates: generalizing containers

Plan for Today

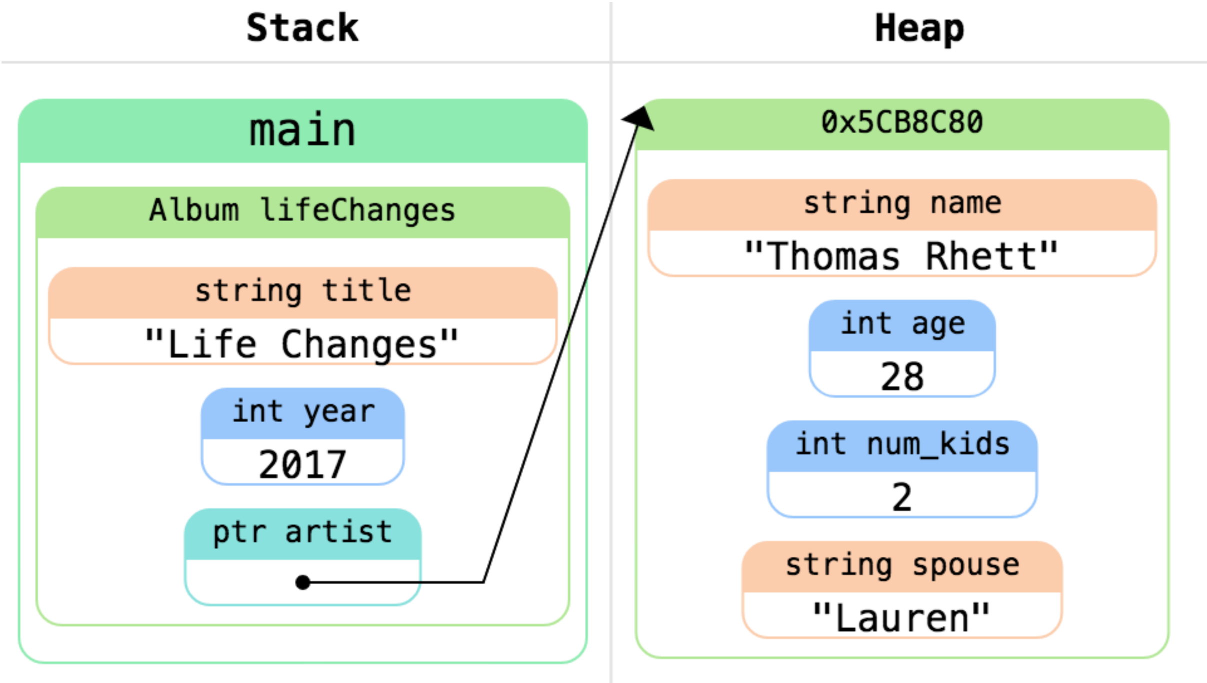
- Continuing discussion of pointers from yesterday
- Arrays
- Classes in C++
- Putting it together: implementing Stack
- Templates: generalizing containers

Why declare on the Heap?

```
Album createAlbum() {
    Artist *thomas = new Artist{"Thomas Rhett", 28, 2, "Lauren"};
    Album lifeChanges{"Life Changes", 2017, thomas};
    return lifeChanges;
}
int main() {
    Album lifeChanges = createAlbum();
    // what does memory look like here?
    cout << lifeChanges.artist->name << endl;
    return 0;
}
```

Why declare on the Heap?

```
Album createAlbum() {
    Artist *thomas = new Artist{"Thomas Rhett", 28, 2, "Lauren"};
    Album lifeChanges{"Life Changes", 2017, thomas};
    return lifeChanges;
}
int main() {
    Album lifeChanges = createAlbum();
    cout << lifeChanges.artist->name;
    return 0;
}
```



Why declare on the Heap?

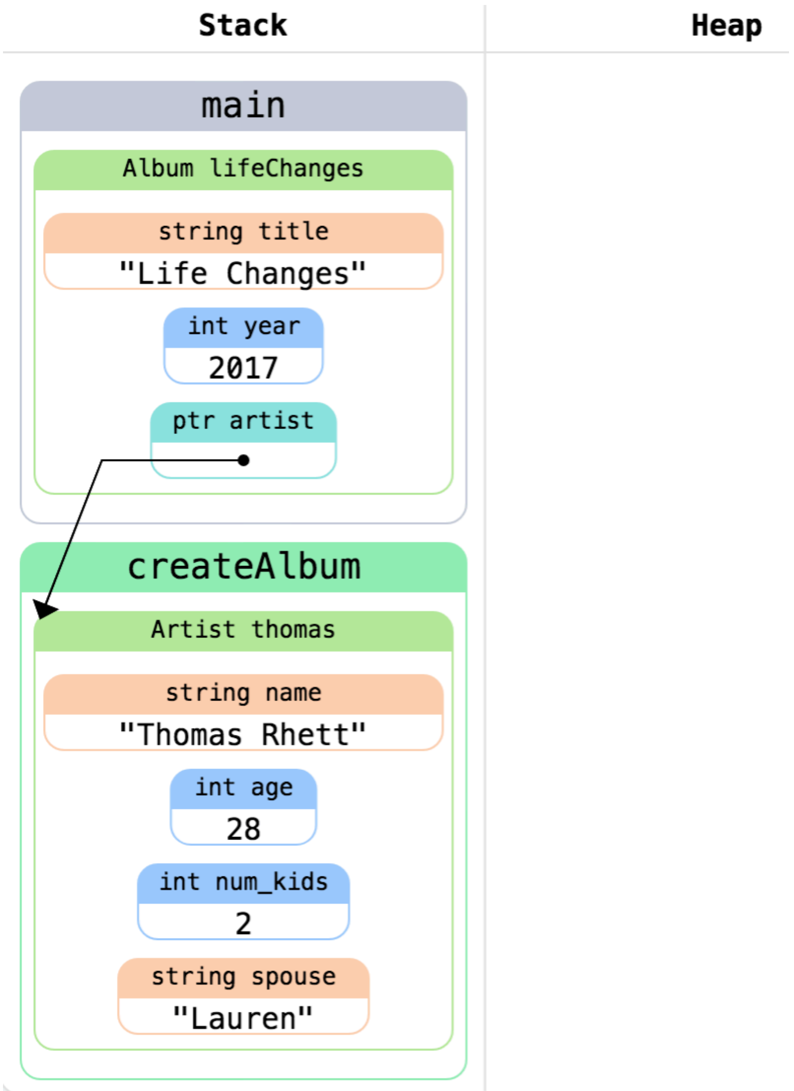
```
Album createAlbum() {
    Artist thomas{"Thomas Rhett", 28,
                  2, "Lauren"};
    Album lifeChanges{"Life Changes",
                      2017, &thomas};
    // what does memory look like here?
    return lifeChanges;
}

int main() {
    Album lifeChanges = createAlbum();
    cout << lifeChanges.artist->name;
}
```

Why declare on the Heap?

```
Album createAlbum() {
    Artist thomas{"Thomas Rhett", 28,
                  2, "Lauren"};
    Album lifeChanges{"Life Changes",
                      2017, &thomas};
    // what does memory look like here?
    return lifeChanges;
}

int main() {
    Album lifeChanges = createAlbum();
    cout << lifeChanges.artist->name;
}
```



Why declare on the Heap?

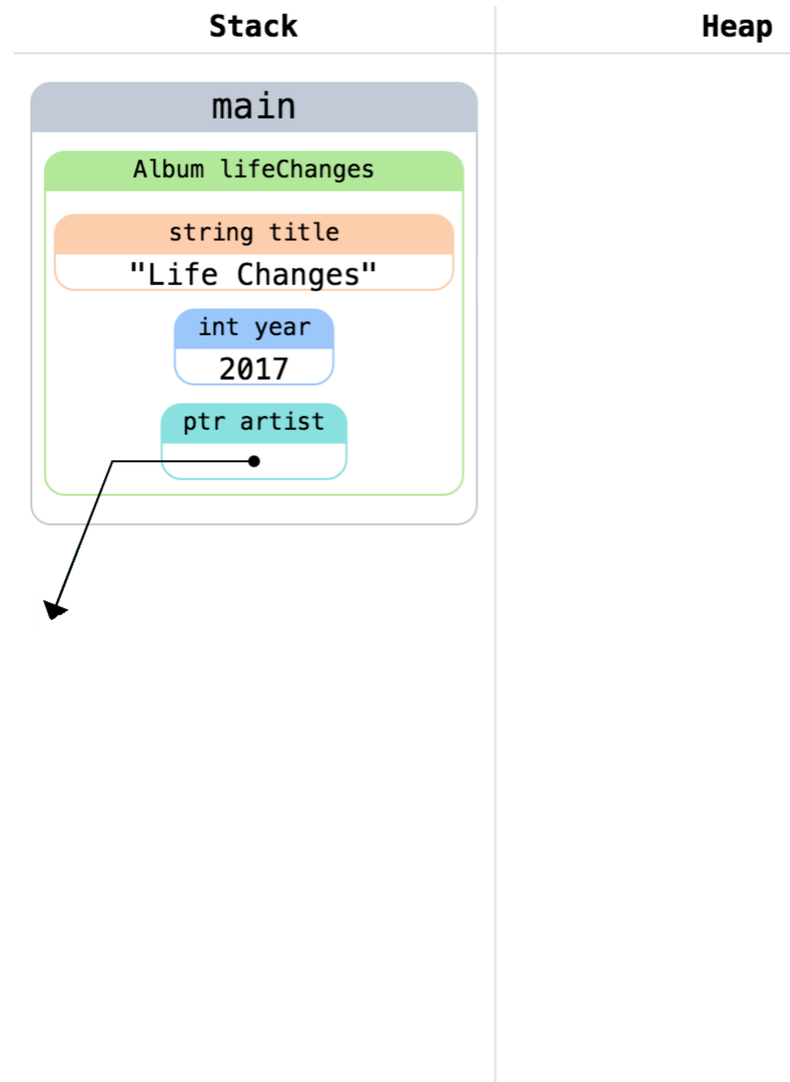
```
Album createAlbum() {
    Artist thomas{"Thomas Rhett", 28,
                  2, "Lauren"};
    Album lifeChanges{"Life Changes",
                      2017, &thomas};
    return lifeChanges;
}

int main() {
    Album lifeChanges = createAlbum();
    // what about here?
    cout << lifeChanges.artist->name;
}
```

Why declare on the Heap?

```
Album createAlbum() {
    Artist thomas{"Thomas Rhett", 28,
                  2, "Lauren"};
    Album lifeChanges{"Life Changes",
                      2017, &thomas};
    return lifeChanges;
}

int main() {
    Album lifeChanges = createAlbum();
    // what about here?
    cout << lifeChanges.artist->name;
}
```



Plan for Today

- Continuing discussion of pointers from yesterday
- Arrays
- Classes in C++
- Putting it together: implementing Stack
- Templates: generalizing containers

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 album

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 a several blocks of memory, not just one block
  - The blocks are stored next to each other
- Solution: array
- 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

Announcements

- Exam logistics
  - Midterm review session tomorrow in class. Bring questions/examples.
  - Highly Encouraged: Complete assignment 4 before the midterm – backtracking will be tested. Though Assn. 4 due date is Thursday, July 25th.

Plan for Today

- Continuing discussion of pointers from yesterday
- Arrays
- Classes in C++
- Putting it together: implementing Stack
- Templates: generalizing containers

Motivation

- So far in this course, we have **used** many collection classes:
  - Vector, Grid, Stack, Queue, Map, Set, HashMap, HashSet, Lexicon, ...

- Now let's explore how they are **implemented**.
  - We will start by implementing our own version of a **Stack class**.
    - To do so, we must learn about **classes**, **arrays**, and **memory** allocation.

  - After that, we will implement several other collections:
    - linked list
    - binary tree set, map;
    - hash table set, map
    - priority queue
    - ...

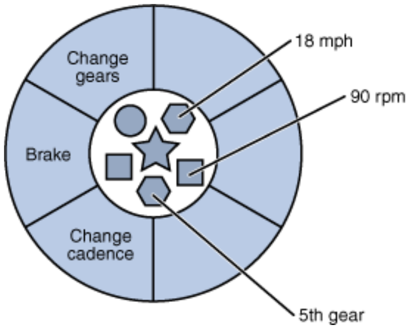
Classes and objects

- **class**: A template for a new type of objects.
  - Allows us to add new types to the language.
  - Examples: Date, Student, BankAccount

- **object**: Entity that combines **state** and **behavior**.
  - **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.
  - **abstraction**: Separation between concepts and details.

Elements of a class

- **member variables**: State inside each object.
  - Also called "instance variables" or "fields"
  - Each object has a copy of each member.



- **member functions**: Behavior inside each object.
  - Also called "methods"
  - Each object has a copy of each method.
  - The method can interact with the data inside that object.
- **constructor**: Initializes new objects as they are created.
  - Sets the initial state of each new object.
  - Often accepts parameters for the initial state of the fields.

Interface vs. code

- C++ separates classes into two kinds of code files:

  `.h`: A "header" file containing the **interface** (declarations).

  `.cpp`: A "source" file containing definitions or **implementation** (method bodies).

  - class `Foo` => must write both `Foo.h` and `Foo.cpp`.

- The content of .h files is `#included` inside .cpp files.
  - Makes them aware of declarations of code implemented elsewhere.
  - At compilation, all definitions are *linked* together into an executable.

Class declaration (.h)

```
#ifndef _classname_h
#define _classname_h

class ClassName {
public:                                    // in ClassName.h
    ClassName(parameters);                 // constructor

    returnType name(parameters);  // member functions
    returnType name(parameters);  // (behavior inside
    returnType name(parameters);  //  each object)
    returnType name(parameters) const;

private:
    type name;       // member variables
    type name;       // (data inside each object)
};

#endif
```

Protection in case multiple .cpp files include this .h, so that its contents won't get declared twice

function promises not to change any of the member variables

IMPORTANT: *must* put a semicolon at end of class declaration (argh)

Class example (v1)

```
// BankAccount.h

#ifndef _bankaccount_h
#define _bankaccount_h

class BankAccount {
public:
    BankAccount(string n, double d);  // constructor
    void deposit(double amount);      // methods
    void withdraw(double amount);
    void getBalance() const;
private:
    string name;        // each BankAccount object
    double balance;     // has a name and balance
};

#endif
```

BankAccount.cpp

```
#include "BankAccount.h"

BankAccount::BankAccount(string name, double initDeposit) {
    this->name = name;
    balance = initDeposit;
}

void BankAccount::deposit(double am
    balance += amount;
}

void BankAccount::withdraw(double amount) {
    balance -= amount;
}

void BankAccount::getBalance() const {
    return balance;
}
```

Include Header
Include the .h file for the class, as well as other files your class implementation needs

BankAccount.cpp

```
#include "BankAccount.h"

BankAccount::BankAccount(string name, double initDeposit) {
    this->name = name;
    balance = initDeposit;
}

void BankAccount::deposit(double amount) {
    balance += amount;
}

void BankAccount::withdraw(double a
    balance -= amount;
}

void BankAccount::getBalance() cons
    return balance;
}
```

Constructor

Initialize the member variables

Notice that each method name is prepended by the **classname::**

the **this** keyword indicates the object, to differentiate from the local variable

BankAccount.cpp

```
#include "BankAccount.h"

BankAccount::BankAccount(string name, double i
    this->name = name;
    balance = initDeposit;
}

void BankAccount::deposit(double amount) {
    balance += amount;
}

void BankAccount::withdraw(double amount) {
    balance -= amount;
}

void BankAccount::getBalance() const {
    return balance;
}
```

Methods

Methods are also prepended by the classname

They can directly access the member variables

BankAccount.cpp

```
#include "BankAccount.h"

BankAccount::BankAccount(string na
    this->name = name;
    balance = initDeposit;
}

void BankAccount::deposit(double a
    balance += amount;
}

void BankAccount::withdraw(double amount) {
    balance -= amount;
}

void BankAccount::getBalance() const {
    return balance;
}
```

Const Methods

Const methods should have **const** at the end, and they should not change the member variables or call non-const member functions

Using objects

```
// client code in bankmain.cpp
BankAccount ba1("Tyler", 1.25);
ba1.deposit(2.00);

BankAccount ba2("Kate", 9999.00);
ba2.withdraw(500.00);
```

```
ba1
name    = "Tyler"
balance = 3.25

ba2
name    = "Kate"
balance = 9499.00
```

- An object groups multiple variables together.
  - Each object contains a `name` and `balance` field inside it.
  - We can get/set them individually.
  - Code that uses your objects is called *client* code.

The implicit parameter

- **implicit parameter**:
  The object on which a member function is called.
  - During the call `ba1.deposit(...)`,
    the object named `ba1` is the implicit parameter.
  - During the call `ba2.withdraw(...)`,
    the object named `ba2` is the implicit parameter.
  - The member function can refer to <u>that object's</u> member variables.
    - We say that it executes in the *context* of a particular object.
    - The function can refer to the data of the object it was called on.
    - It behaves as if each object has its own *copy* of the member functions.

Plan for Today

- Continuing discussion of pointers from yesterday
- Arrays
- Classes in C++
- Putting it together: implementing Stack
- Templates: generalizing containers

A Stack Class

- Recall: a Stack has O(1) push and pop operations
- Only need to add to the end
- Idea: we need the implementation of stack to store all the elements the client added
- How could we implement a stack using an array?

How Stack works

- Inside a Stack is an **array** storing the elements you have added.
  - Typically the array is larger than the data added so far, so that it has some extra slots in which to put new elements later.
    - We call this an *unfilled array*.

```
Stack<int> s;
s.push(42);
s.push(-5);
s.push(17);
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	42	-5	17	?	?	?	?	?	?	?
size	3	*capacity*	10							

Resize when out of space

```
// grows array to twice the capacity if needed
void ArrayStack::checkResize() {
    if (size == capacity) {
        // create bigger array and copy data over
        int* bigger = new int[2 * capacity]();
        for (int i = 0; i < capacity; i++) {
            bigger[i] = elements[i];
        }
        delete[] elements;
        elements = bigger;
        capacity *= 2;
    }
}
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*
value	3	8	9	7	5	12	4	8	1	6	**75**	0	0	0	0	0	0	0	0	0

size **11** *capacity* **20**

Plan for Today

- Continuing discussion of pointers from yesterday
- Arrays
- Classes in C++
- Putting it together: implementing Stack
- Templates: generalizing containers

Template class

- **Template class**: A class that accepts a type parameter(s).
  - In the header and cpp files, mark each class/function as templated.
  - Replace occurrences of the previous type `int` with T in the code.

```
// ClassName.h
template<typename T>
class ClassName {
    ...
};

// ClassName.cpp
template<typename T>
type ClassName::name(parameters) {
    ...
}
```

Template .h and .cpp

- Because of an odd quirk with C++ templates, the separation between .h header and .cpp implementation must be reduced.
  - Either write all the bodies in the .h file (suggested),
  - Or `#include` the .cpp at the end of .h file to join them together.

```
// ClassName.h
#ifndef _classname_h
#define _classname_h

template<typename T>
class ClassName {
    ...
};

#include "ClassName.cpp"
#endif   // _classname_h
```

Overflow Slides

- Making objects Printable
- Destructors
- Class Constants

Operator Overloading

- **operator overloading**: Redefining the behavior of a common operator in the C++ language.

- Syntax:

```
returnType operator op(parameters); // .h
returnType operator op(parameters) { // .cpp
    statements;
}
```

  – For example, `a + b` becomes `operator+(Foo& a, Foo& b)`

Make Objects Printable

- Make it easy to print your object to cout, overload <<

```
ostream& operator <<(ostream& out, Type& name) {
    statements;
    return out;
}
```

  - `ostream` is a base class that represents `cout`, file output streams, ...

Example <<

```
// BankAccount.h
class BankAccount {
   ...
};

ostream& operator <<(ostream& out, BankAccount& ba);
```

```
// BankAccount.cpp
ostream& operator <<(ostream& out, BankAccount& ba) {
    out << ba.getName() << ": $" << ba.getBalance();
    return out;
}
```

Example ==

```
// BankAccount.h
class BankAccount {
    ...
};

bool operator ==(const BankAccount& ba1,
                 const BankAccount& ba2);
```

```
// BankAccount.cpp
bool operator ==(const BankAccount& ba1,
                 const BankAccount& ba2) {
    return ba1.getName() == ba2.getName()
        && ba1.getBalance() == ba2.getBalance();
}
```

Destructor

```
// ClassName.h
~ClassName();
```

```
// ClassName.cpp
ClassName::~ClassName() { ...
```

- **destructor**: Called when the object is deleted by the program.
  (when the object falls out of { } scope)

  – Useful if your object needs to free any memory as it dies.
    - `delete` any pointers stored as private members
    - `delete[]` any arrays stored as private members

Class Constants

- **class constant**: An unmodifiable static variable in the .h file.
  - Assign its value in the .cpp, outside of any method.
    - Don't write `static` when assigning the value in the .cpp.
  - For integral types, you can actually assign the variable in the .h file.

```
// BankAccount.h
class BankAccount {
    static const int BANK_ROUTING_NUM = 006029593;
    static const double INTEREST_RATE;
};
```

```
// BankAccount.cpp
// set the constant to store 3.25%
const double BankAccount::INTEREST_RATE = 0.0325;
```