CSCE 121
Introduction to Program Design & Concepts

Dynamic Memory

Dr. Tim McGuire

Grateful acknowledgment to Dr. Philip Ritchey and Dr. Michael Moore for some of the material on which these slides are based.

Dynamic Memory Allocation

- Can allocate storage for a variable while program is running
- Computer returns address of newly allocated variable
- Uses `new` operator to allocate memory:

```
double *dptr = nullptr;
dptr = new double;
```

- `new` returns address of memory location

Dynamic Memory Allocation

- Can also use `new` to allocate array:

```
const int SIZE = 25;
arrayPtr = new double[SIZE];
```

- Can then use `[]` or pointer arithmetic to access array:

```
for(i = 0; i < SIZE; i++)
    arrayptr[i] = i * i;
```

or

```
for(i = 0; i < SIZE; i++)
    *(arrayptr + i) = i * i;
```

- Program will terminate if not enough memory available to allocate

Releasing Dynamic Memory

- Use `delete` to free dynamic memory:

```
delete fptr;
```

- Use `[]` to free dynamic array:

```
delete [] arrayptr;
```

- Only use `delete` with dynamic memory!

Recall: Memory Layout

Stack and heap
grow toward each other.

Stack

- Recall:
  - As stack grows / shrinks, items are automatically cleared from Memory
    - i.e. when a function ends,
      all of its objects (variables) are cleared from Memory

- Sometimes we want objects to live on after a function is finished.
  - Put on the Heap / Free Store

Heap aka Dynamic Memory

- How to use the heap?
  - Use '**new**'
    - Gets memory from the heap
    - Returns a pointer
  - Use '*' to dereference the pointer
  - Initialize with **nullptr** (i.e. 0) – *See Null Pointer note in zyBook.*

```
int i = 7; // put item on the stack
int* j = nullptr;
j = new int(11); // put item on the heap
cout << “Value in i: " << i << endl;
cout << "Address of i: " << &i << endl;
cout << “Value in j: " << j << endl;
cout << "Address of j: " << &j << endl;
cout << "*j (value at address pointed to in j): " << *j << endl;
int* k = new int[5]; // allocate an array on the heap
```

Heap

- If we put something on the heap we also have to remove it.
  - If we don't we might have a memory leak.
  - More on memory management / challenges with pointers later.

- How to remove from the heap?
  - Use '`delete`'
  - Use '`delete[]`' if deleting an array

```
delete j; // remove item from the heap
          // j still points to the memory in the heap
          // that can be a problem
```

Only use `delete` with dynamic memory!

Notes on new / delete

- If you delete memory that has already been deleted, then an exception will likely occur.
- If you delete a pointer that is set to **`nullptr`**, then no error occurs. (The **`delete`** operator is designed to have no effect when used on a null pointer.)
- If you try to dereference a pointer that has been deleted, then an exception will likely occur.
- So try to set the pointer to **`nullptr`** after you use delete.

Dynamic Memory Allocation

dynamicArray.cpp

```
1  // This program totals and averages the sales figures for any number of days.
2  // The figures are stored in a dynamically allocated array.
3  // Based on an original from Tony Gaddis
4
5  #include <iostream>
6  #include <iomanip>
7  using namespace std;
8
9  int main()
10 {
11     double *sales = nullptr,  // To dynamically allocate an array (must use *, not [])
12     total = 0.0,              // Accumulator
13     average;                  // To hold average sales
14     int numDays,              // To hold the number of days of sales
15         count;                // Counter variable
16
17
18     // Get the number of days of sales.
19     cout << "How many days of sales figures do you wish ";
20     cout << "to process? ";
21     cin >> numDays;
22
23     // Dynamically allocate an array large enough to hold
24     // that many days of sales amounts.
25     sales = new double[numDays];
```

Dynamic Memory Allocation

```
26
27      // Get the sales figures for each day.
28      cout << "Enter the sales figures below.\n";
29      for (count = 0; count < numDays; count++)
30      {
31      cout << "Day " << (count + 1) << ": ";
32      cin >> sales[count];
33      }
34
35      // Calculate the total sales
36      for (count = 0; count < numDays; count++)
37      {
38      total += sales[count];
39      }
40
41      // Calculate the average sales per day
42      average = total / numDays;
43
44      // Display the results
45      cout << fixed << showpoint << setprecision(2);
46      cout << "\n\nTotal Sales: $" << total << endl;
47      cout << "Average Sales: $" << average << endl;
48
49      // Free dynamically allocated memory
50      delete [] sales;
51      sales = nullptr;  // make sales a null pointer
52
53      return 0;
54  }
```

mcguire@CSCE-1PSF1G2: /mnt/c/windows/system32

```
Tue Oct 12 20:35:00 mcguire DynamicMemory$ g++ dynamicArray.cpp
Tue Oct 12 20:35:18 mcguire DynamicMemory$ ./a.out
How many days of sales figures do you wish to process? 5
Enter the sales figures below.
Day 1: 898.63
Day 2: 652.32
Day 3: 741.85
Day 4: 852.96
Day 5: 921.37


Total Sales: $4067.13
Average Sales: $813.43
Tue Oct 12 20:36:04 mcguire DynamicMemory$
```

Pointers to Structures

- A structure variable has an address
- Pointers to structures are variables that can hold the address of a structure:

```
Student *stuPtr;
```

- Can use `&` operator to assign address:

```
stuPtr = &stu1;
```

- Structure pointer can be a function parameter

Accessing Structure Members via Pointer Variables

- Must use `()` to dereference pointer variable, not field within structure:

```
cout << (*stuPtr).studentID;
```

- Can use structure pointer operator to eliminate `()` and use clearer notation:

```
cout << stuPtr->studentID;
```

Code Segment showing use of structure pointers

dynamicStructure.cpp

```
34  }
35
36  //***************************************************************
37  // Definition of function getData. Uses a pointer to a  *
38  // Student structure variable. The user enters student  *
39  // information, which is stored in the variable.        *
40  //***************************************************************
41
42  void getData(Student *s)
43  {
44     // Get the student name.
45     cout << "Student name: ";
46     getline(cin, s->name);
47
48     // Get the student ID number.
49     cout << "Student ID Number: ";
50     cin >> s->idNum;
51
52     // Get the credit hours enrolled.
53     cout << "Credit Hours Enrolled: ";
54     cin >> s->creditHours;
55
56     // Get the GPA.
57     cout << "Current GPA: ";
58     cin >> s->gpa;
59  }
```

CSCE 121

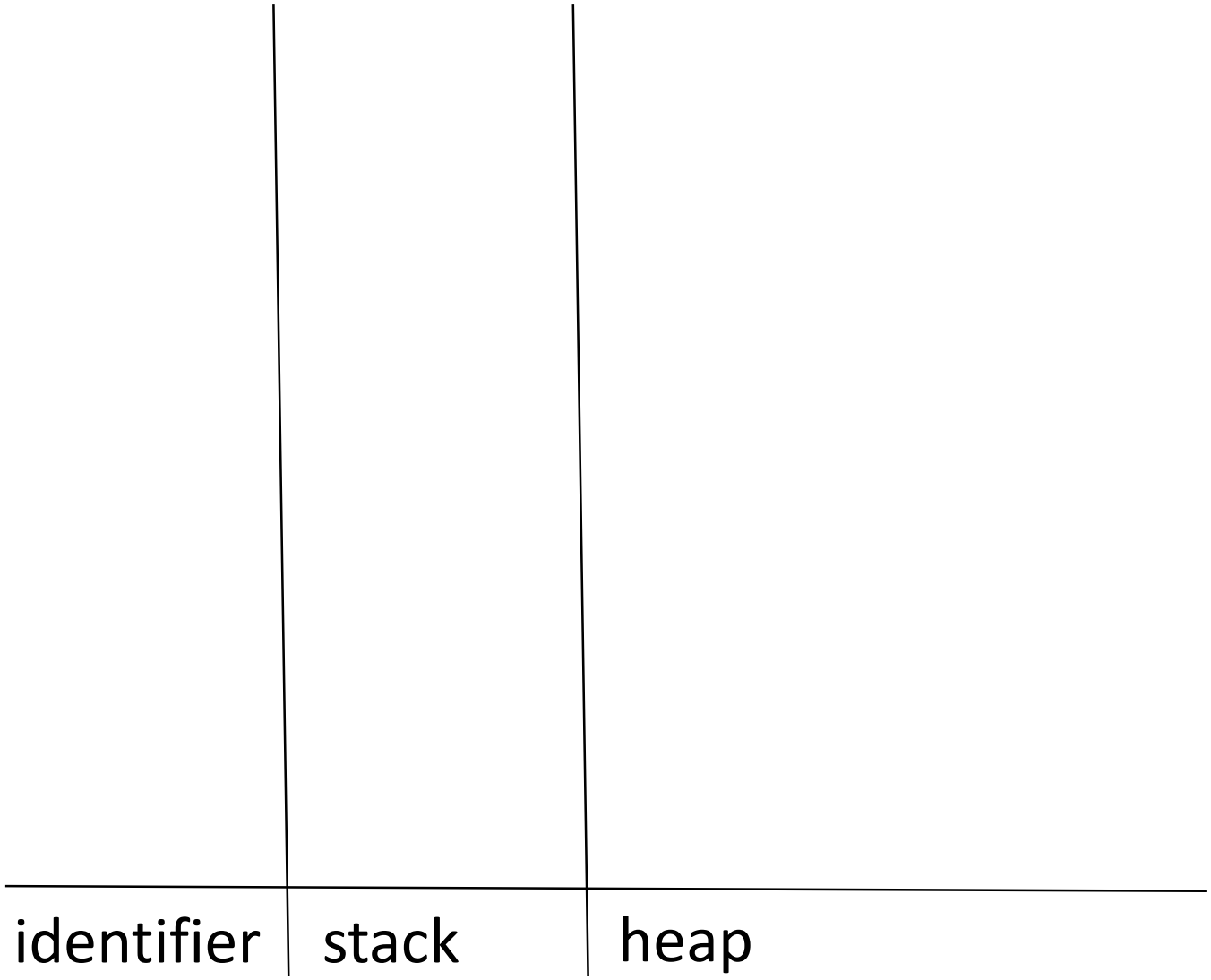
Introduction to Program Design & Concepts

Memory Diagrams 4
How Dynamic Memory Works

Dr. Tim McGuire

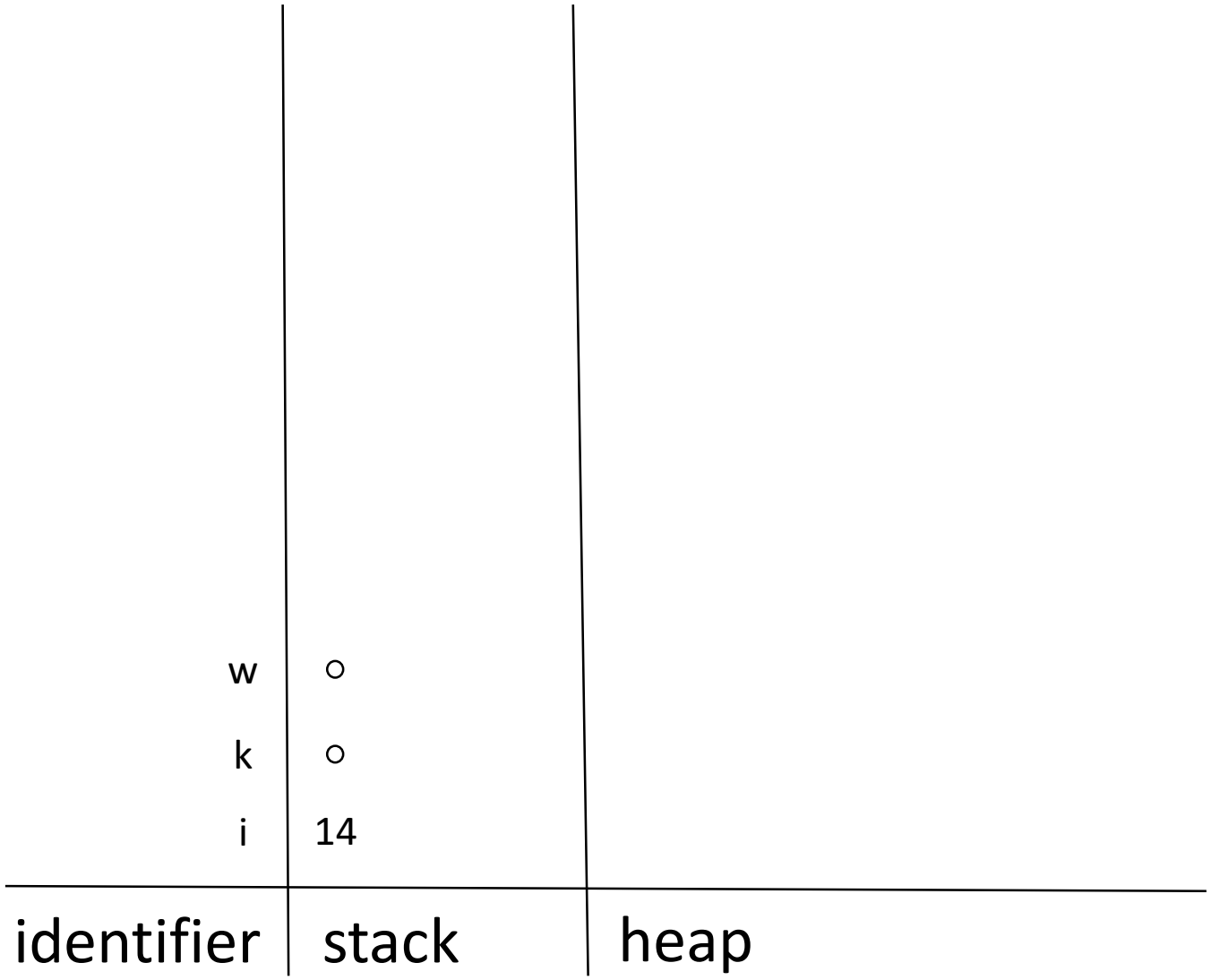
Grateful acknowledgment to Dr. Philip Ritchey and Dr. Michael Moore for some of the material on which these slides are based.

output



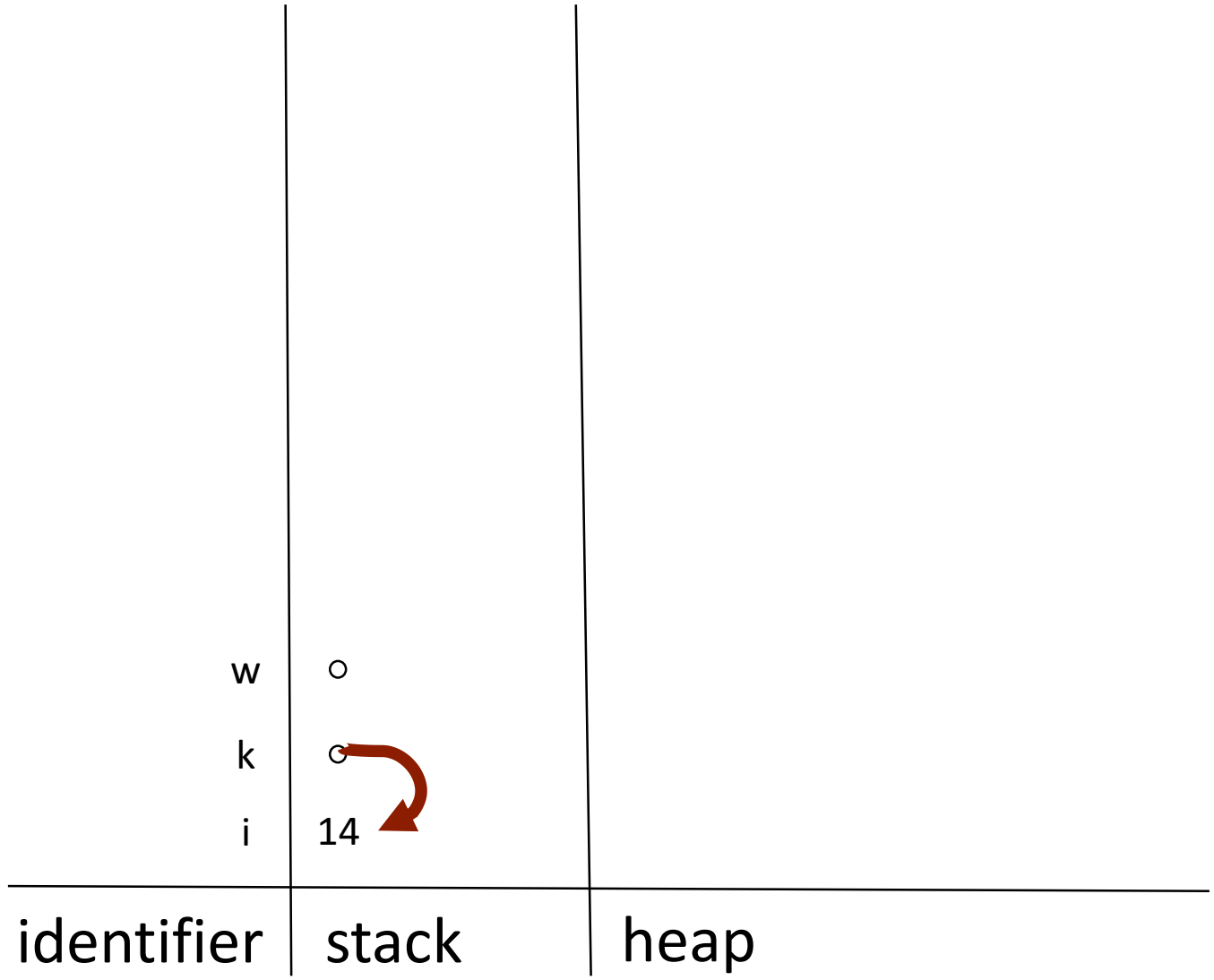
output

```
int main() {
    int i = 14;

    int* k = &i;

    k = new int(3);

    delete k;

    Date* w = new Date(7, 7, 2015);

    delete w;

    w = nullptr;
}
```



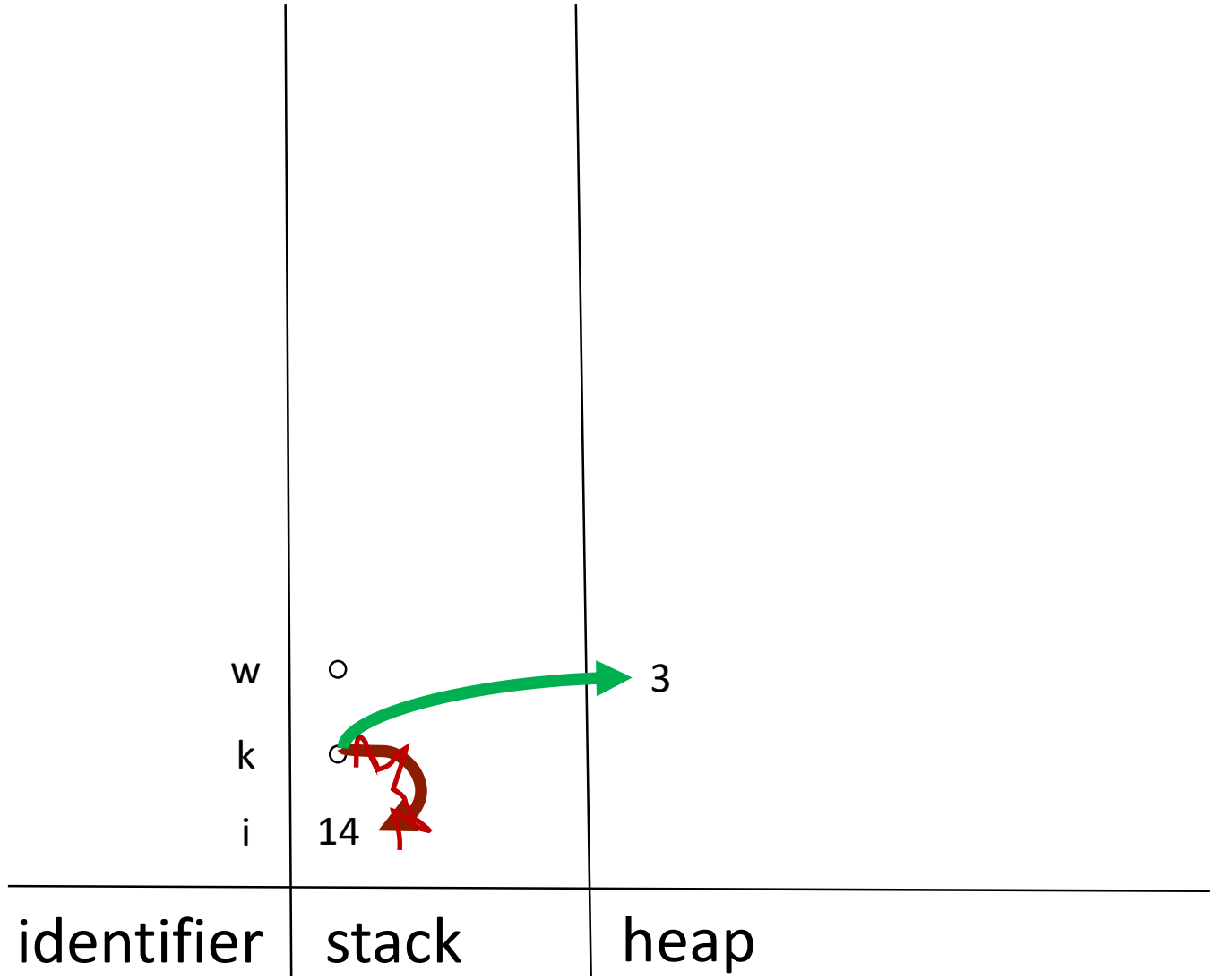
output

```
int main() {
    int i = 14;

    int* k = &i;

    k = new int(3);

    delete k;

    Date* w = new Date(7, 7, 2015);

    delete w;

    w = nullptr;
}
```



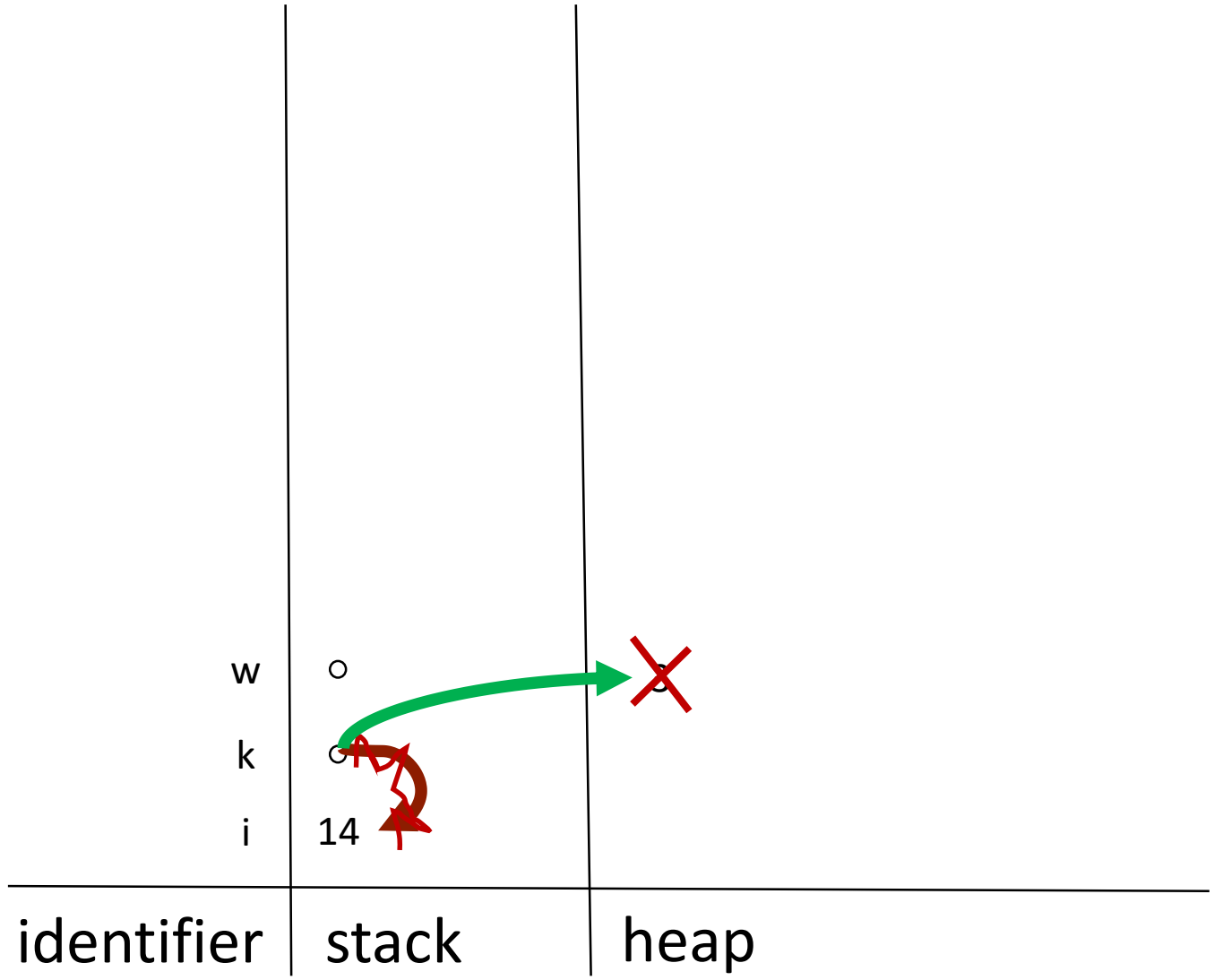
output

```
int main() {
    int i = 14;

    int* k = &i;

    k = new int(3);

    delete k;

    Date* w = new Date(7, 7, 2015);

    delete w;

    w = nullptr;
}
```



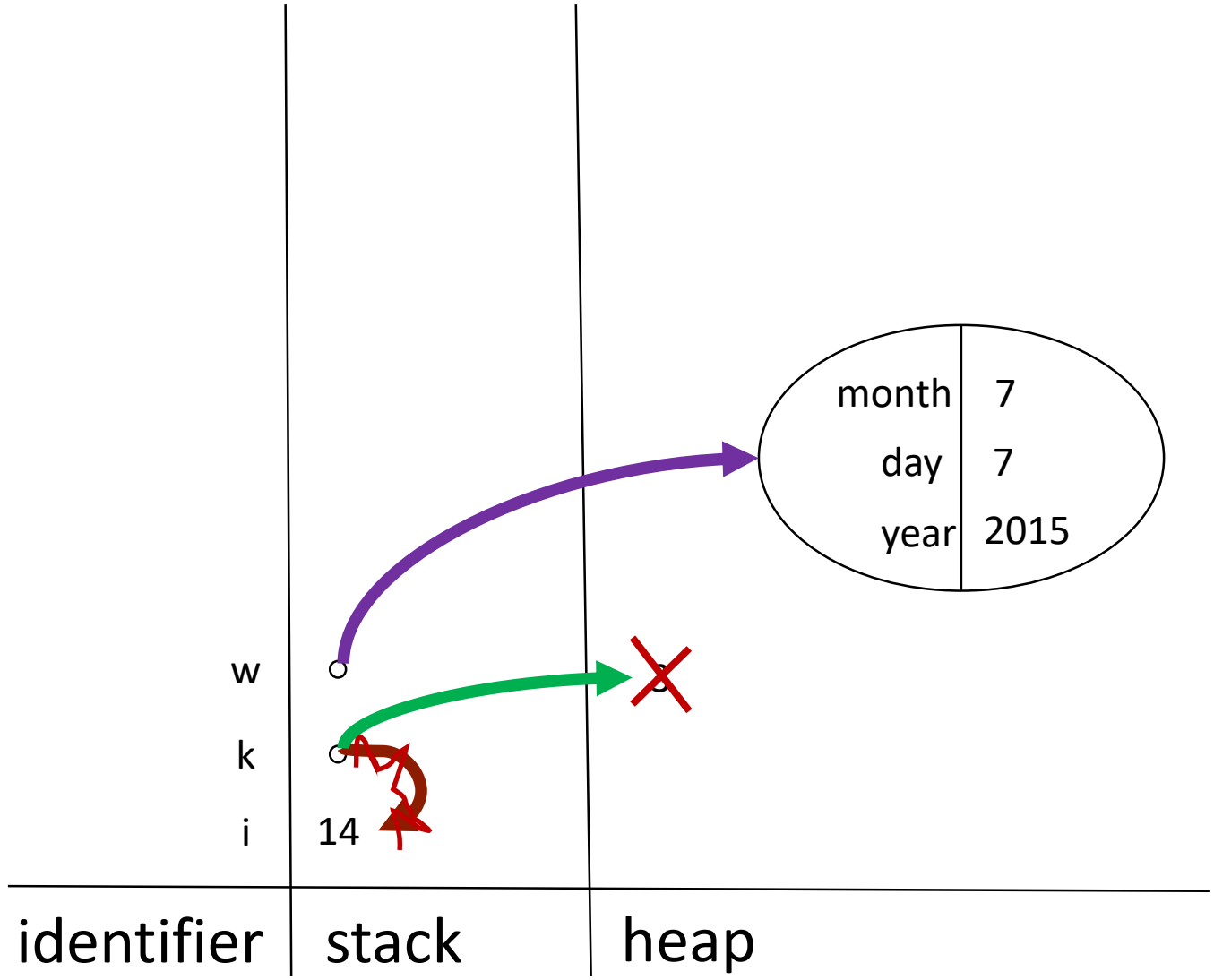
output

```
int main() {
    int i = 14;

    int* k = &i;

    k = new int(3);

    delete k;

    Date* w = new Date(7, 7, 2015);

    delete w;

    w = nullptr;
}
```



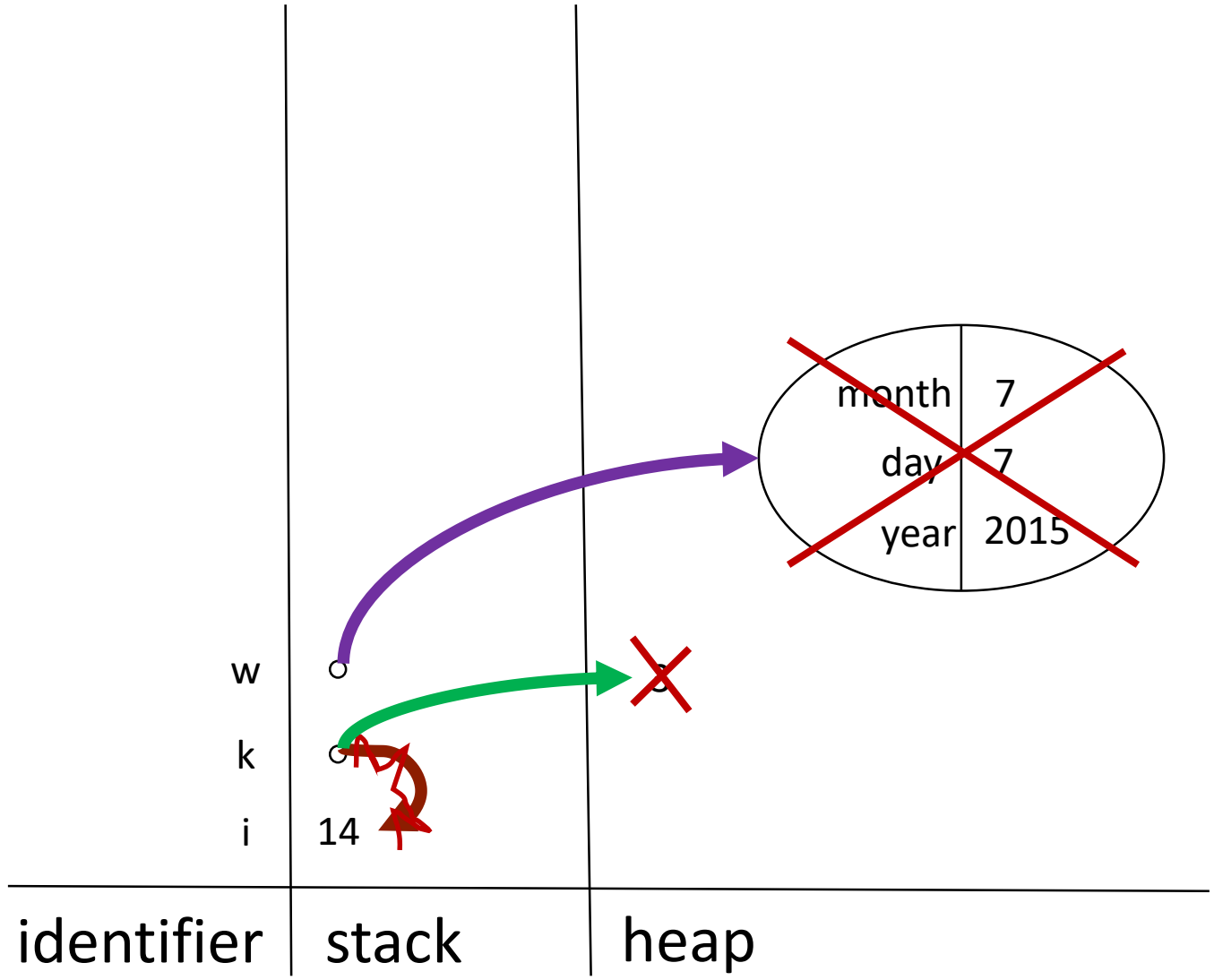
output

```
int main() {
    int i = 14;

    int* k = &i;

    k = new int(3);

    delete k;

    Date* w = new Date(7, 7, 2015);

    delete w;

    w = nullptr;
}
```



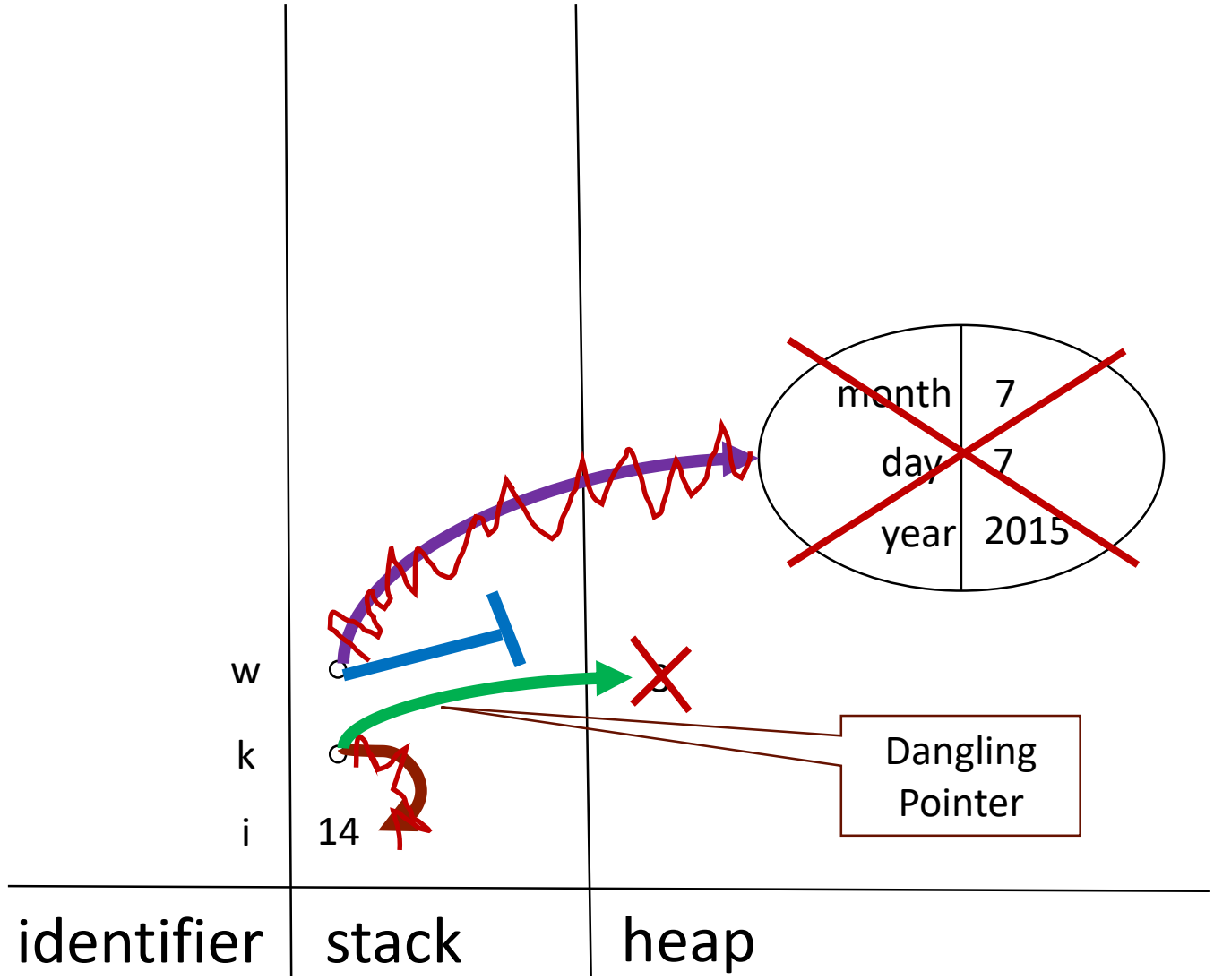
output

```
int main() {
    int i = 14;

    int* k = &i;

    k = new int(3);

    delete k;

    Date* w = new Date(7, 7, 2015);

    delete w;

    w = nullptr;
}
```



output

```
int main() {
    int i = 14;

    int* k = &i;

    k = new int(3);

    delete k;

    Date* w = new Date(7, 7, 2015);

    delete w;

    w = nullptr;
}
```



CSCE 121
Introduction to Program Design & Concepts

Dynamic Memory Management

Dr. Tim McGuire

Grateful acknowledgment to Dr. Philip Ritchey and Dr. Michael Moore for some of the material on which these slides are based.

Memory Leaks

- Lose access to memory allocated on the heap.
  - Easy to lose when dealing with pointers
- We want to avoid!!!
- Sometimes conflated with poor memory management.

Managing Memory

- Programmer gets memory from the heap
- Programmer must free it when done
  - Commonly referred to as **garbage collection**.

- Good exam question
  - What are specific scenarios that can result in a memory leak?
  - Think about the definition of a memory leak as we continue to discuss dynamic memory management.

CSCE 121

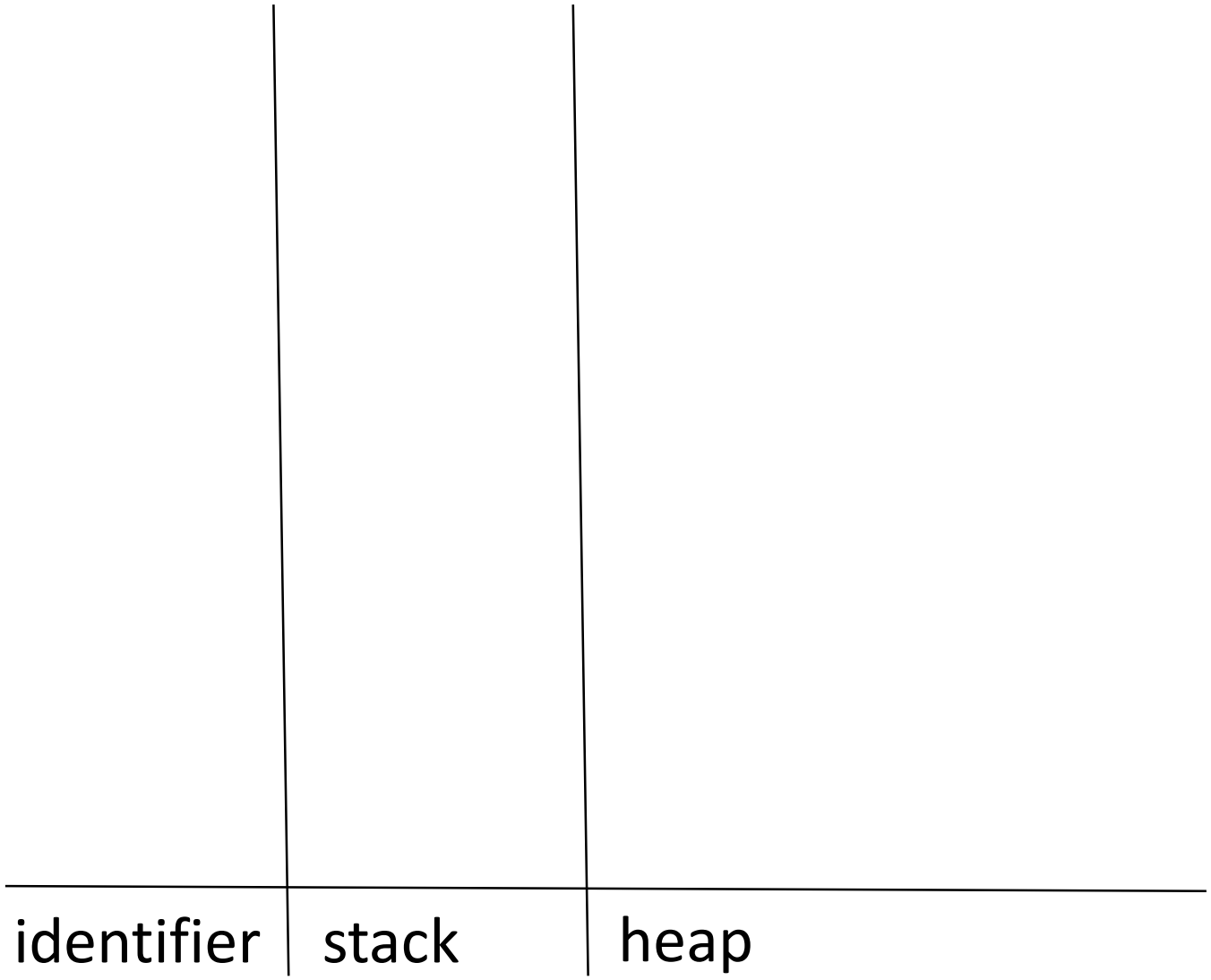
Introduction to Program Design & Concepts

How Memory Leaks Work with Memory Diagram

Dr. Tim McGuire

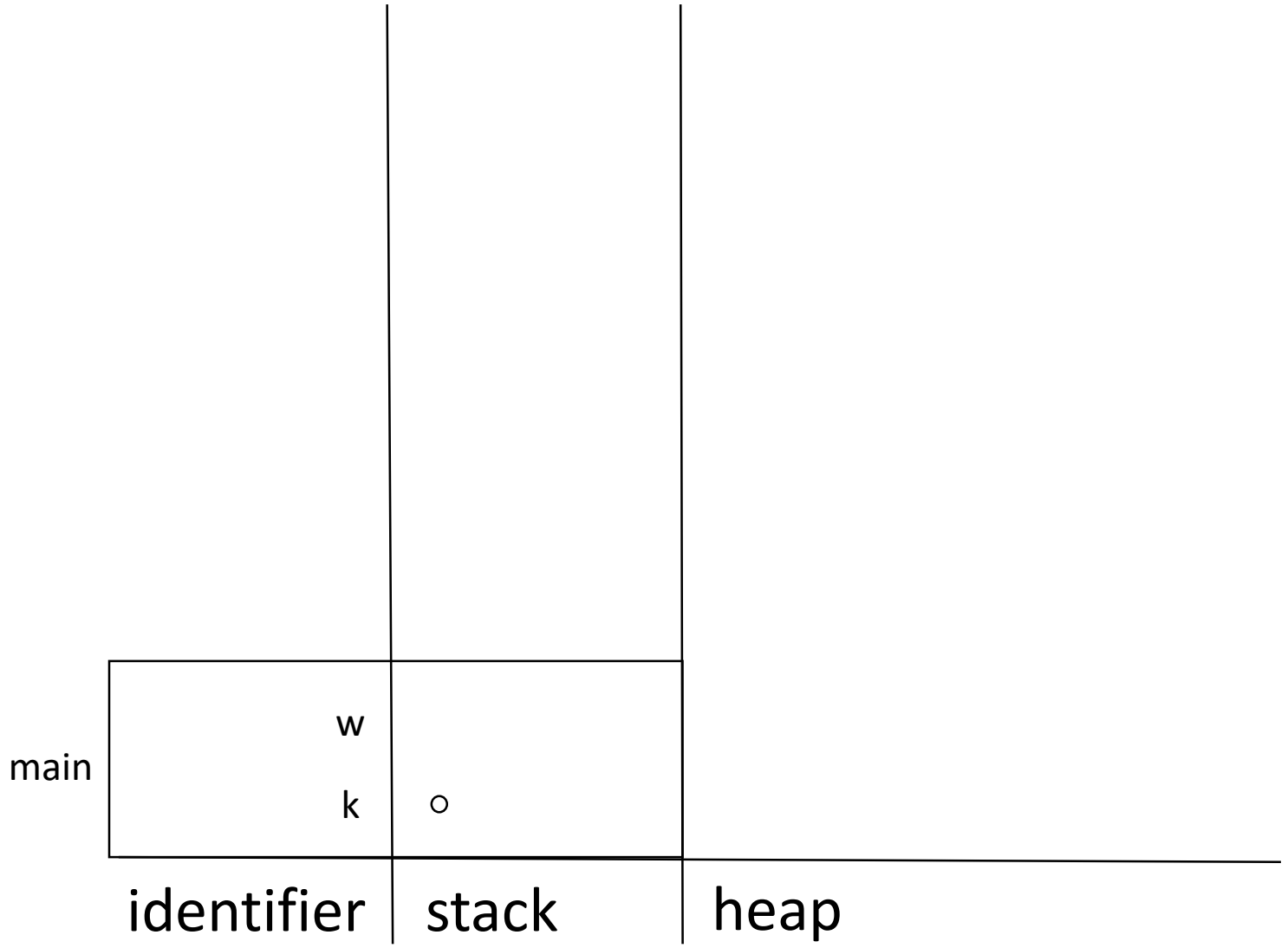
Grateful acknowledgment to Dr. Philip Ritchey and Dr. Michael Moore for some of the material on which these slides are based.

output



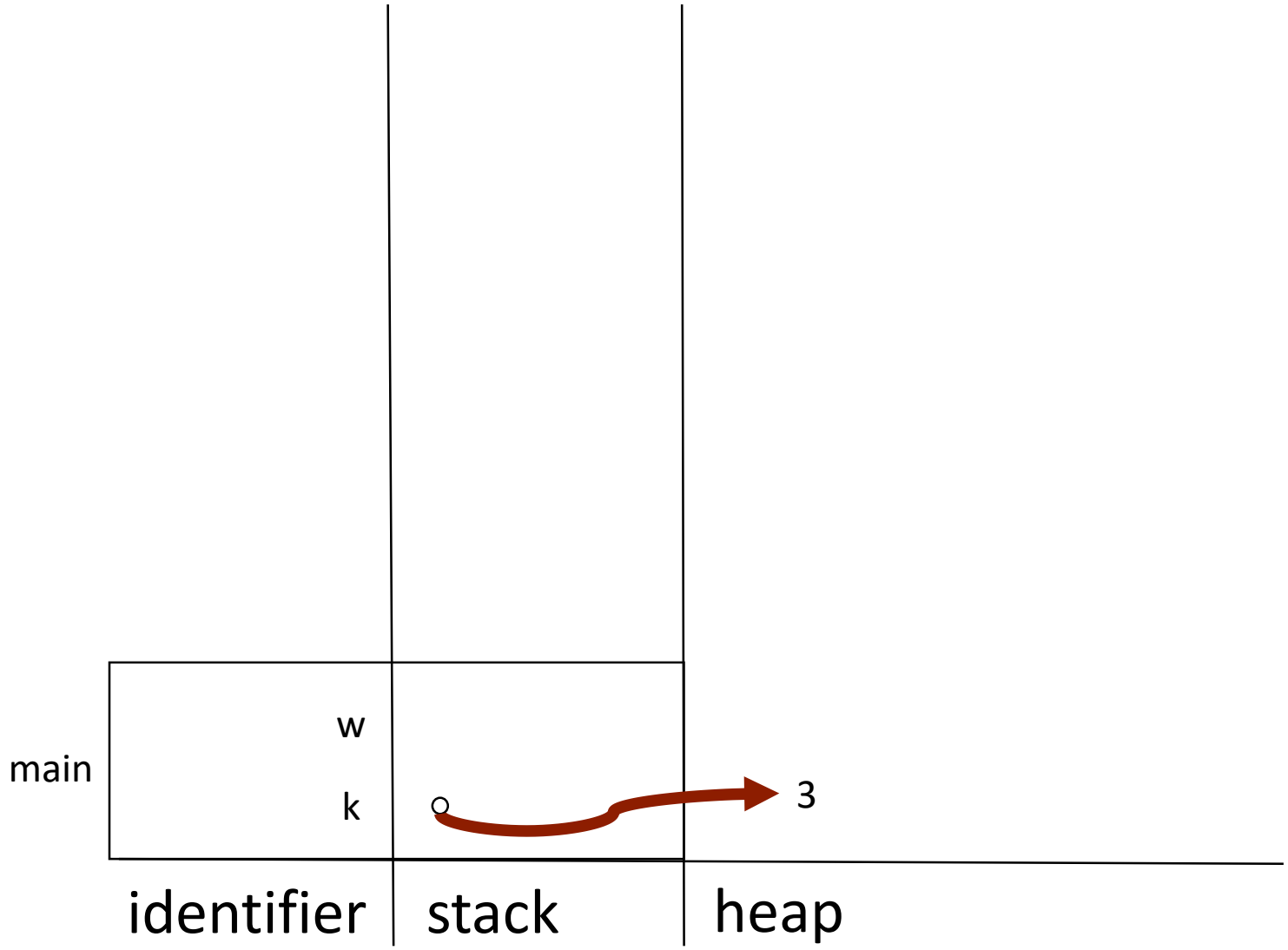
output

```
int getANumber() {
    int* z = new int(15);
    return *z;
}
int main() {
    int* k = new int(3);
    k = new int (7);
    int w = getANumber();
}
```



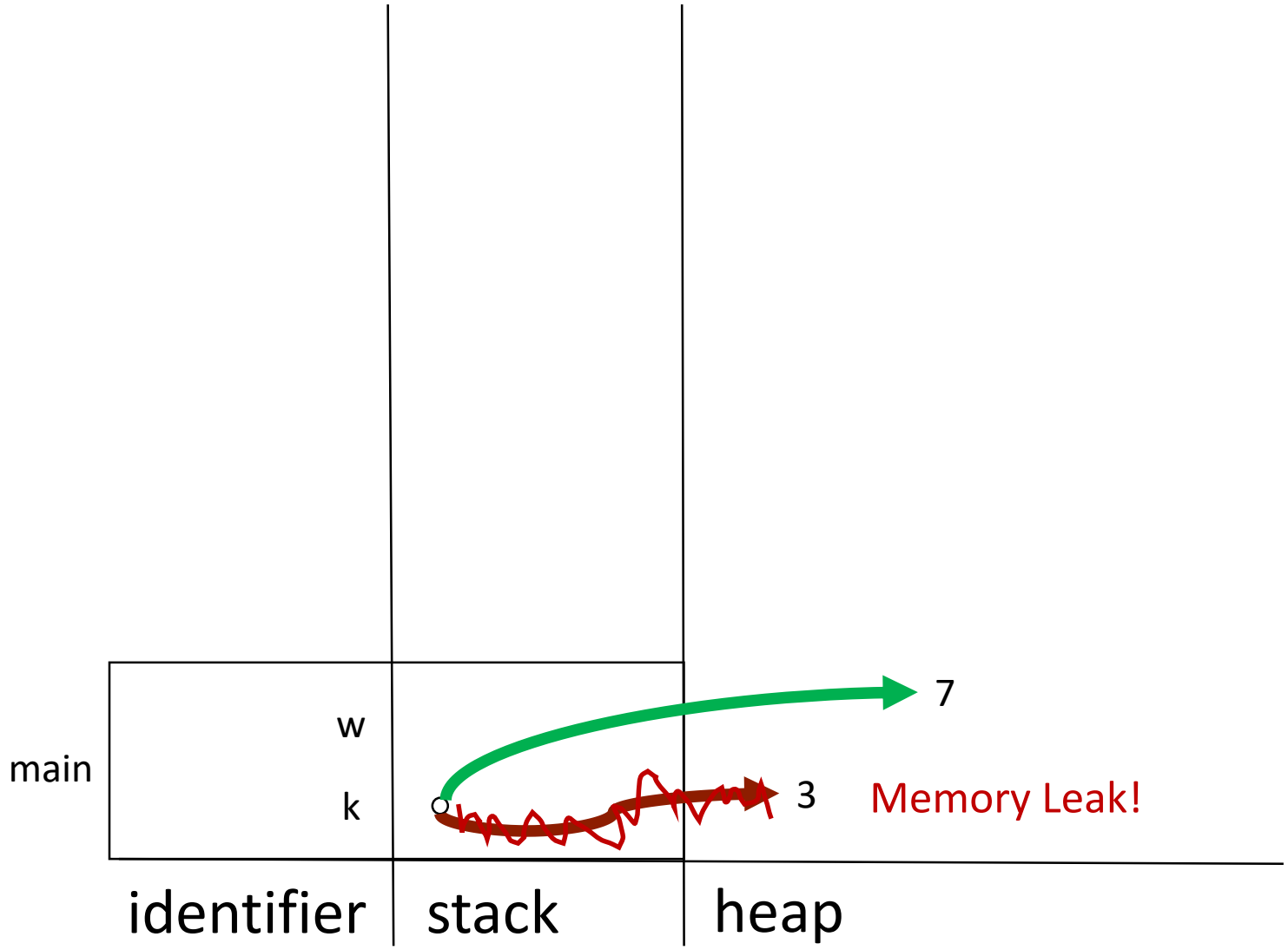
output

```
int getANumber() {
    int* z = new int(15);
    return *z;
}

int main() {
    int* k = new int(3);
    k = new int (7);
    int w = getANumber();
}
```



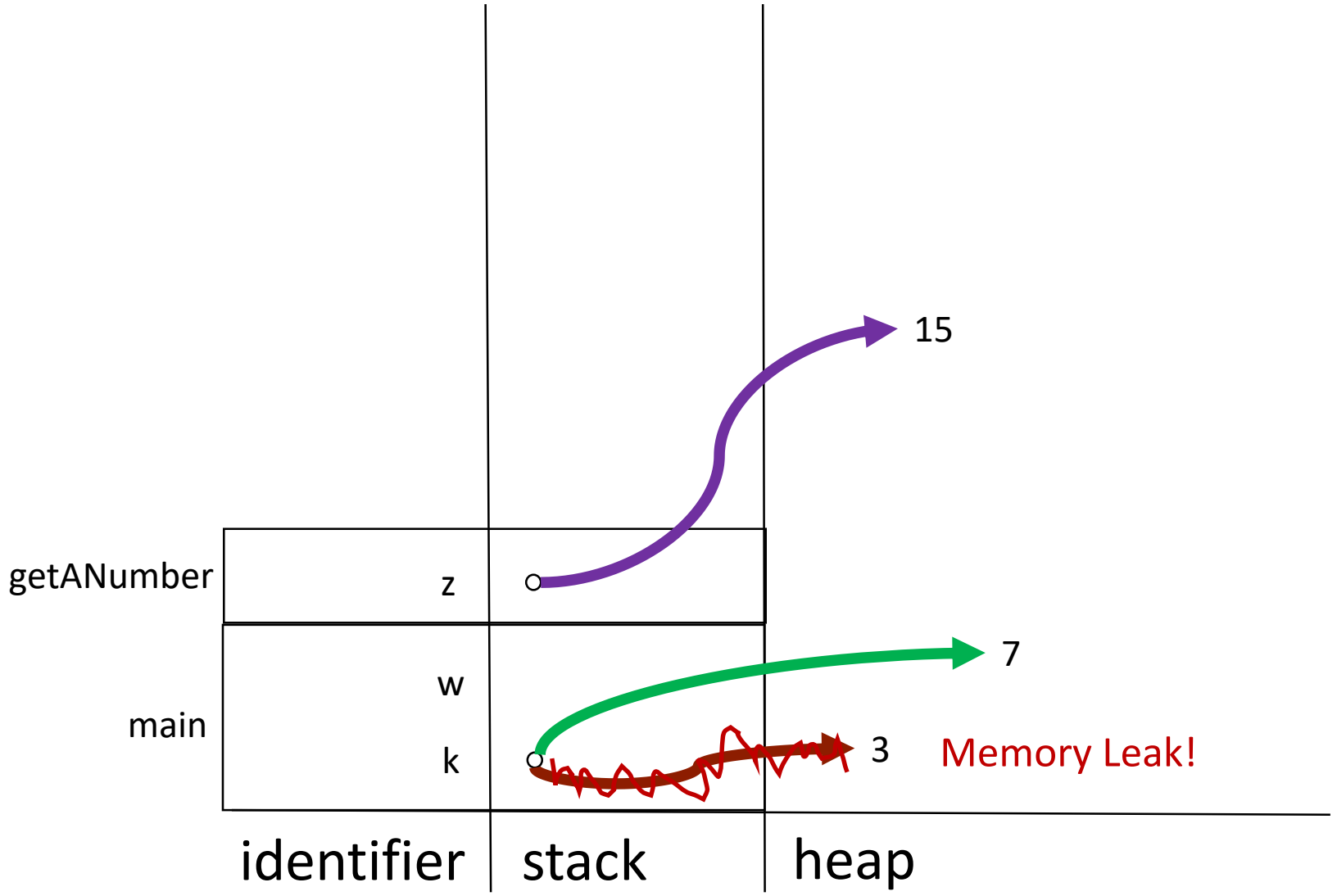
output

```
int getANumber() {
    int* z = new int(15);
    return *z;
}

int main() {
    int* k = new int(3);
    k = new int (7);
    int w = getANumber();
}
```



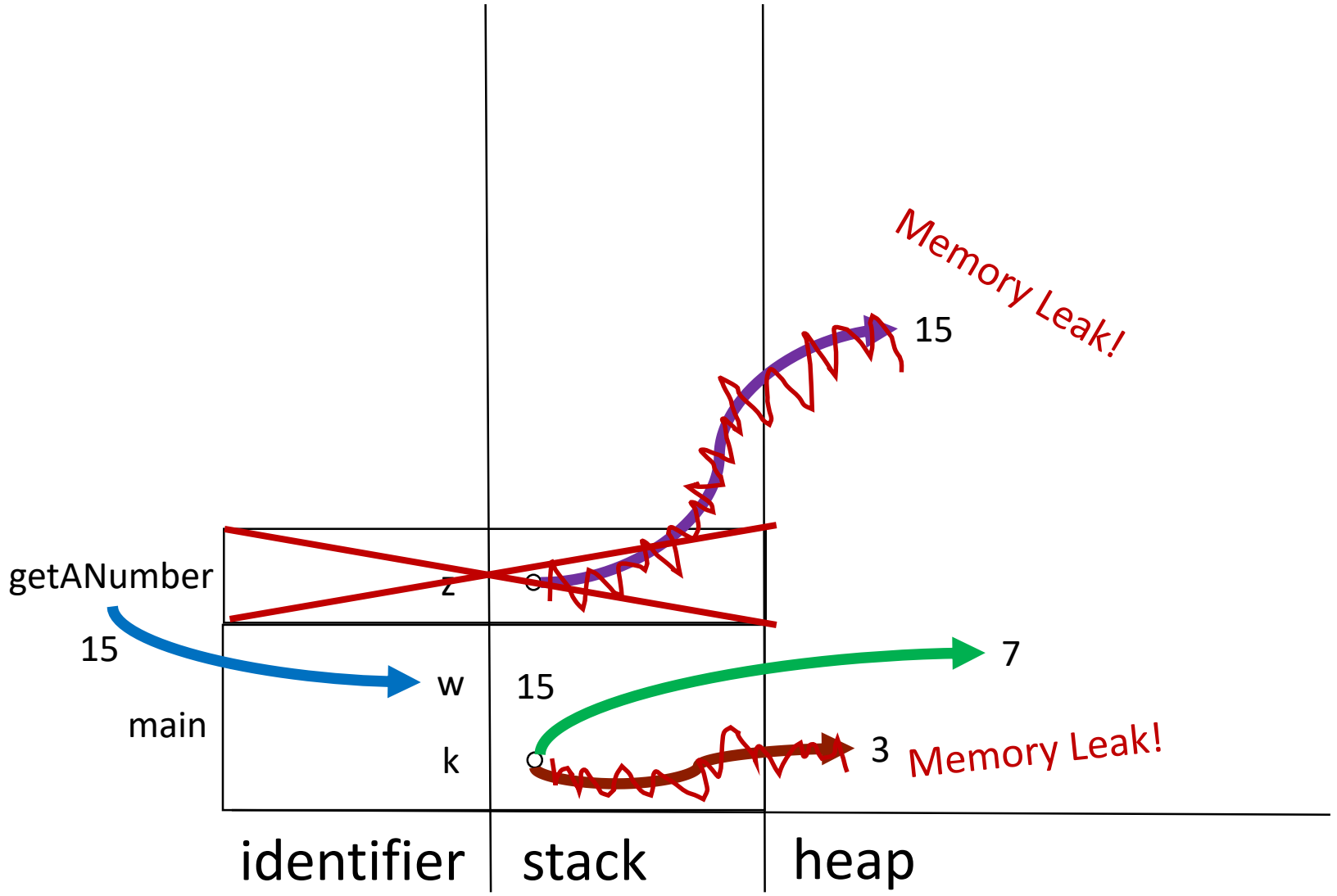
output

```
int getANumber() {
    int* z = new int(15);
    return *z;
}
int main() {
    int* k = new int(3);
    k = new int (7);
    int w = getANumber();
}
```



output

```
int getANumber() {
    int* z = new int(15);
    return *z;
}
int main() {
    int* k = new int(3);
    k = new int (7);
    int w = getANumber();
}
```



Resizing Arrays

For convenience, let’s put this in a struct

```
struct ResizableArray {
   int size;      // number of actual elements in the array
   int capacity; // the current capacity of the array
   int * data;   // data points to the array containing the data
};
```

We can initialize our resizeable array thus:

```
ResizableArray a;
a.size = 0;
a.capacity = INITIAL_SIZE;
a.data = new int[a.capacity];
```

Now, let's write code to add data to the array:

```
 if(a.size < a.capacity) {
    a.data[a.size] = val;
    a.size++;
}
//    but note that we have to handle the case if the array
//    becomes full
```

What do we do if the array becomes full?

- We have to resize it.
- That means allocate a new array and copy the original into it.

```
int *temp = new int[2 * a.capacity];
// copy the current values into the new array
for(int index = 0; index < a.size; index++)
   temp[index] = a.data[index];
// free the original array
delete [] a.data;
// make a.data point to the new array and update the capacity
a.data = temp;
a.capacity = 2 * a.capacity;
```

resize-array.cpp