CSCE 121

Introduction to Program Design & Concepts

A More Advanced Look at Functions

Dr. Tim McGuire

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

Why Do We Define Functions?

Modular Programming

Modular Programming

- Modular programming: breaking a program up into smaller, manageable functions or modules

- Function: a collection of statements to perform a task
- Motivation for modular programming:
  - Improves maintainability of programs
  - Simplifies the process of writing programs

This program has one long, complex function containing all of the statements necessary to solve a problem.

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

In this program the problem has been divided into smaller problems, each of which is handled by a separate function.

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

main function

```
void function2()
{
   statement;
   statement;
   statement;
}
```

function 2

```
void function3()
{
   statement;
   statement;
   statement;
}
```

function 3

```
void function4()
{
   statement;
   statement;
   statement;
}
```

function 4

Static Local Variables

- Local variables only exist while the function is executing. When the function terminates, the contents of local variables are lost.

- `static` local variables retain their contents between function calls.

- `static` local variables are defined and initialized only the first time the function is executed. `0` is the default initialization value.

Using a Static Variable

```
5  void showStatic(); // Function prototype
6
7  int main()
8  {
9     // Call the showStatic function five times.
10    for (int count = 0; count < 5; count++)
11       showStatic();
12    return 0;
13 }
14
15 //*********************************************************
16 // Definition of function showStatic.                     *
17 // statNum is a static local variable. Its value is displayed  *
18 // and then incremented just before the function returns.      *
19 //*********************************************************
20
21 void showStatic()
22 {
23    static int statNum;
24
25    cout << "statNum is " << statNum << endl;
26    statNum++;
27 }
```

statNum is automatically initialized to zero. Notice that it retains its value between function calls.

Program Output

```
Mon Feb 24 mcguire Functions-2 $ ./a.out
statNum is 0
statNum is 1
statNum is 2
statNum is 3
statNum is 4
Mon Feb 24 mcguire Functions-2 $
```

Default Arguments

A ***Default argument*** is an argument that is passed automatically to a parameter if the argument is missing on the function call.

- Must be a constant declared in prototype:

```
void evenOrOdd(int = 0);
```

- Can be declared in header if no prototype

- Multi-parameter functions may have default arguments for some or all of them:

```
int getSum(int, int=0, int=0);
```

Default Arguments Example

Default arguments specified in the prototype

```
1  // This program demonstrates default function arguments.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype with default arguments
6  void displayStars(int cols = 10, int rows = 1);
7
8  int main()
9  {
10    displayStars();      // Use default values for cols and rows.
11    cout << endl;
12    displayStars(5);     // Use default value for rows.
13    cout << endl;
14    displayStars(7, 3);  // Use 7 for cols and 3 for rows.
15    return 0;
16 }
17
```

Default Arguments Example (cont'd)

```
18  //*********************************************************
19  // Definition of function displayStars.                   *
20  // The default argument for cols is 10 and for rows is 1.*
21  // This function displays a square made of asterisks.     *
22  //*********************************************************
23
24  void displayStars(int cols, int rows)
25  {
26     // Nested loop. The outer loop controls the rows
27     // and the inner loop controls the columns.
28     for (int down = 0; down < rows; down++)
29     {
30        for (int across = 0; across < cols; across++)
31           cout << "*";
32        cout << endl;
33     }
34  }
```

mcguire@CSE-MCGUIRE-NB1: /mnt/c/Windows/System32

```
Mon Feb 24 mcguire Functions-2 $ g++ default-args.cpp
Mon Feb 24 mcguire Functions-2 $ ./a.out
**********

*****

*******
*******
*******
Mon Feb 24 mcguire Functions-2 $
```

Default Arguments

- If not all parameters to a function have default values, the defaultless ones are declared first in the parameter list:

```
int getSum(int, int=0, int=0);// OK
int getSum(int, int=0, int);  // NO
```

- When an argument is omitted from a function call, all arguments after it must also be omitted:

```
sum = getSum(num1, num2);     // OK
sum = getSum(num1, , num3);   // NO
```

Overloading Functions

Recall

- Declaration
  - `return_type name (formal arguments);`
  - `int sum(const int[] vals);`
- Note, the declaration does NOT need names for the formal arguments.
  - `int sum(const int[]); // acceptable in declaration`
- Why?
  - Compiler only needs this information to find the function definition that corresponds to it.

Overloading Functions

- ***Overloaded functions*** have the same name but different parameter lists
- Can be used to create functions that perform the same task but take different parameter types or different number of parameters
- Compiler will determine which version of function to call by argument and parameter lists

Function Signature

- Unique combination of
  - Name and parameter types and parameter order.
  - Frequently equated with the declaration.
    - However, it is possible to have different declarations that are considered the same signature.

- `int mean(int a, int b);`
- `int mean(int first, int last);`

- `double mean(int, int);`

Not allowed: Same signature, i.e. mean with two int parameters.

Not allowed: cannot differ by return type alone

Compiler

- The compiler tries to match function calls to a signature that matches a declaration it has already seen.
  - Note: The compiler starts at the top of the file containing main() and works its way down.
  - This is why functions must be declared above where they are called in a program.
  - #include essentially puts declarations at the top of the file.

Function Overloading Examples

Using these overloaded functions,

```
void getDimensions(int);            // 1
void getDimensions(int, int);       // 2
void getDimensions(int, double);    // 3
void getDimensions(double, double);// 4
```

the compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);              // 1
getDimensions(length, width);       // 2
getDimensions(length, height);      // 3
getDimensions(height, base);        // 4
```

Function Overloading Example

```
1  // This program uses overloaded functions.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  // Function prototypes
7  int square(int);
8  double square(double);
9
10 int main()
11 {
12    int userInt;
13    double userFloat;
14
15    // Get an int and a double.
16    cout << fixed << showpoint << setprecision(2);
17    cout << "Enter an integer and a floating-point value: ";
18    cin >> userInt >> userFloat;
19
20    // Display their squares.
21    cout << "Here are their squares: ";
22    cout << square(userInt) << " and " << square(userFloat);
23    return 0;
24 }
```

The overloaded functions have different parameter lists

Passing a double

Passing an int

(Program Continues)

Function Overloading

```
26  //*********************************************************
27  // Definition of overloaded function square.              *
28  // This function uses an int parameter, number. It returns the *
29  // square of number as an int.                            *
30  //*********************************************************
31
32  int square(int number)
33  {
34     return number * number;
35  }
36
37  //*********************************************************
38  // Definition of overloaded function square.              *
39  // This function uses a double parameter, number. It returns    *
40  // the square of number as a double.                      *
41  //*********************************************************
42
43  double square(double number)
44  {
45     return number * number;
46  }
```

mcguire@CSE-MCGUIRE-NB1: /mnt/c/Windows/System32

```
Mon Feb 24 mcguire Functions-2 $ g++ overloading-fcns.cpp
Mon Feb 24 mcguire Functions-2 $ ./a.out
Enter an integer and a floating-point value: 12 3.16
Here are their squares: 144 and 9.99
Mon Feb 24 mcguire Functions-2 $
```

Overloading

- Essentially creating multiple functions with
  - The same name
  - Different parameter configurations
    - **Number** of parameters
    - **Types** of parameters
    - **Order** of parameter *types*

- Only overload if the functions do very similar things!

Overloading Guidelines

If you want functions that:

- Have completely different logic → Use different function names
- Have similar logic, and can work for different datatypes → Use function overloading

Memory Diagrams
Stack Frames

Stack Frames and Function Calls

1. Function Called (including main)
2. New area of memory (stack frame) is set up on top of the stack.
   - Stack frame has an entry for
     - each formal parameter
     - return address – where in the code to return to after function exits
     - frame pointer – location of previous frame
     - each local variable
3. Body of function executes
4. Function finishes and its stack frame goes away
   - i.e. memory is recycled for later use

- Potential source of programming bugs if you don't understand how this works!

Stack Frame

- Contains
  - Parameters
    - Value of actual arguments are **copied** into corresponding formal arguments (parameters) in stack frame
    - In the same order as the parameters: 1st argument to 1st parameter, 2nd to 2nd, etc.
    - The function computes using the formal arguments (parameters)
    - No change is made to the actual arguments
  - Local variables
  - Information for returning to the calling function
    - Return address: address in code of calling function to go back to
    - Frame pointer: location of previous frame

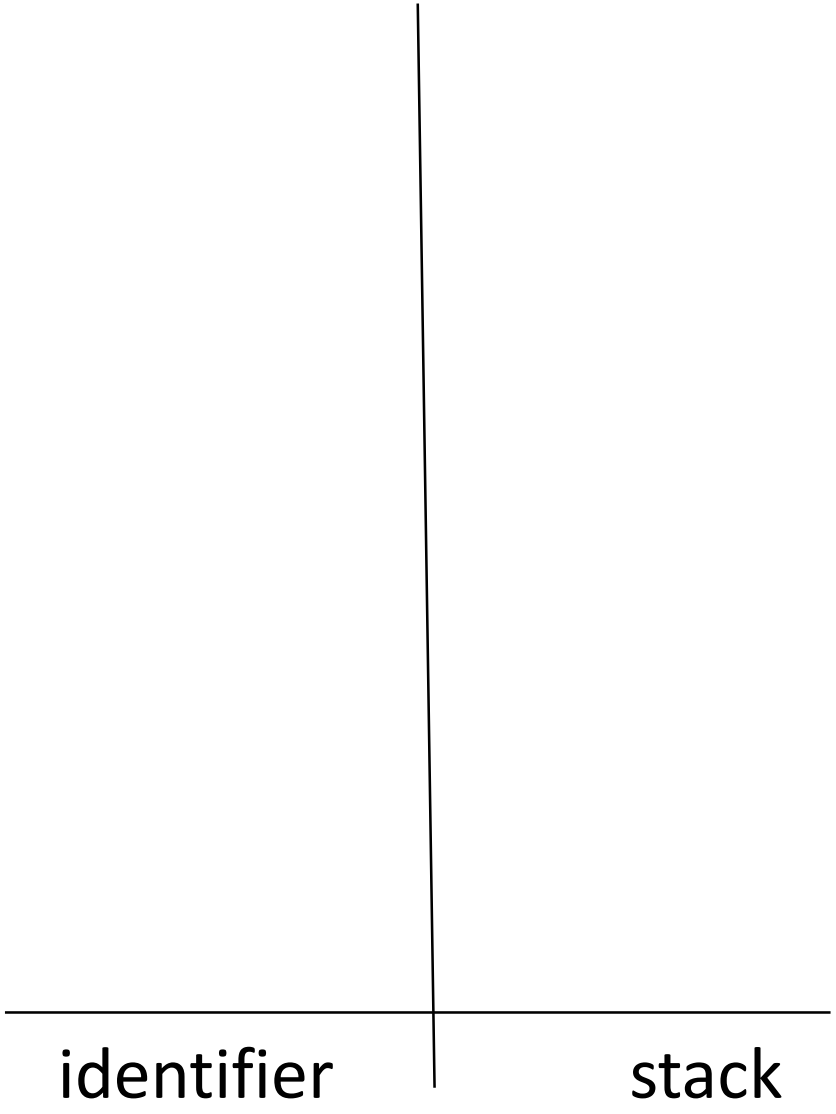
Memory Diagram

- Helps us visualize the call stack
- zyBooks starts at top since addressing is considered to start at zero and diagrams are frequently drawn with zero at the bottom.
  - This is the standard convention: The stack grows down. Low addresses are down, high addresses are up.
- However, when talking about stacking, we usually think: put things on **top**!
  - So we will start at the **bottom** (high addresses) so that when we put a stack frame on top (lower address), it looks like it is going on top: The stack grows up.
  - Either way: the stack grows from high addresses → low addresses
- When we draw a memory diagram, we'll include information about
  - Parameters
  - Local variables
  - Name of function

output

Program

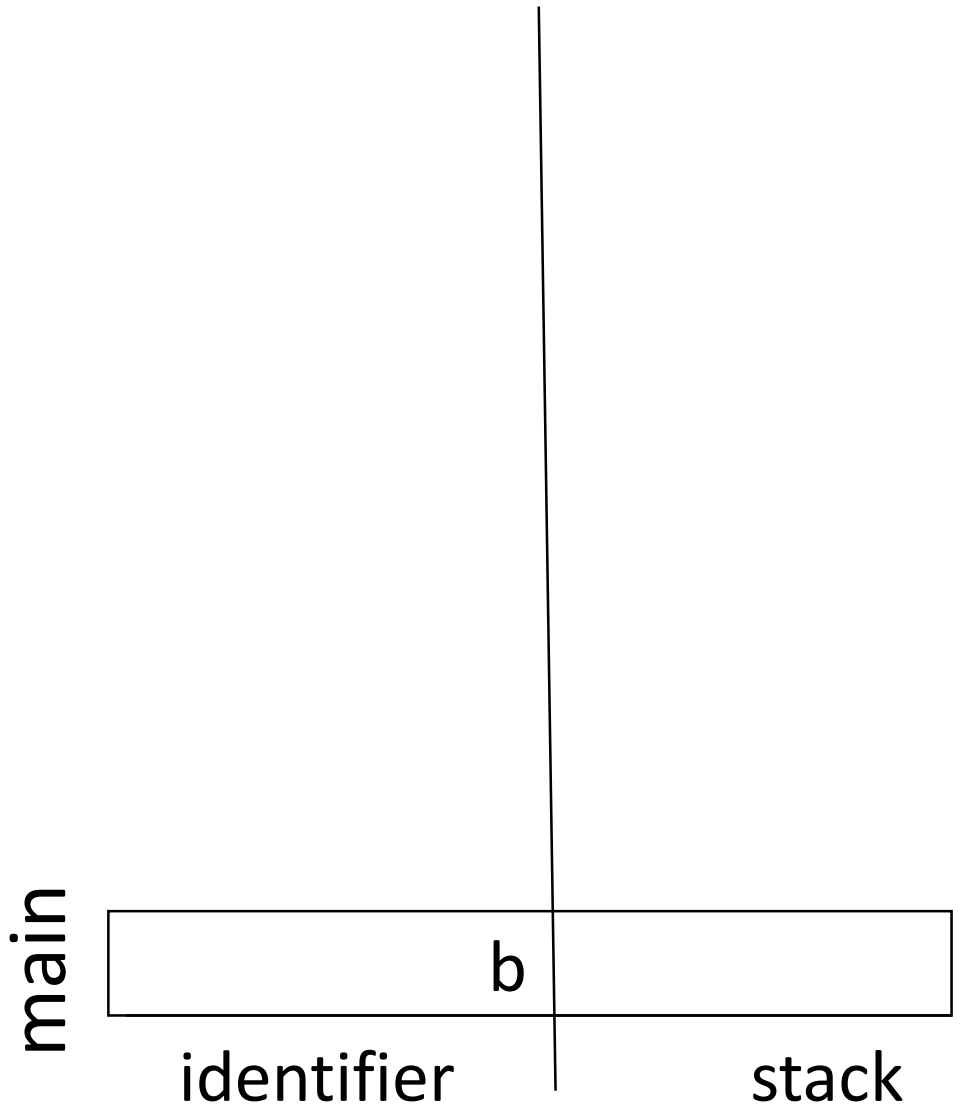
```
int em(int a, int b) {
    int k = 1000;
    int whoop = a + b + k;
    return whoop;
}

int gig(int rev) {
    int howdy = 22;
    return em(rev, howdy);
}

int main() {
    int b = gig(em(7, -10));
    cout << "b: " << b << endl;
    return 0;
}
```



output

Program

```
int em(int a, int b) {
    int k = 1000;
    int whoop = a + b + k;
    return whoop;
}

int gig(int rev) {
    int howdy = 22;
    return em(rev, howdy);
}

int main() {
    int b = gig(em(7, -10));
    cout << "b: " << b << endl;
    return 0;
}
```

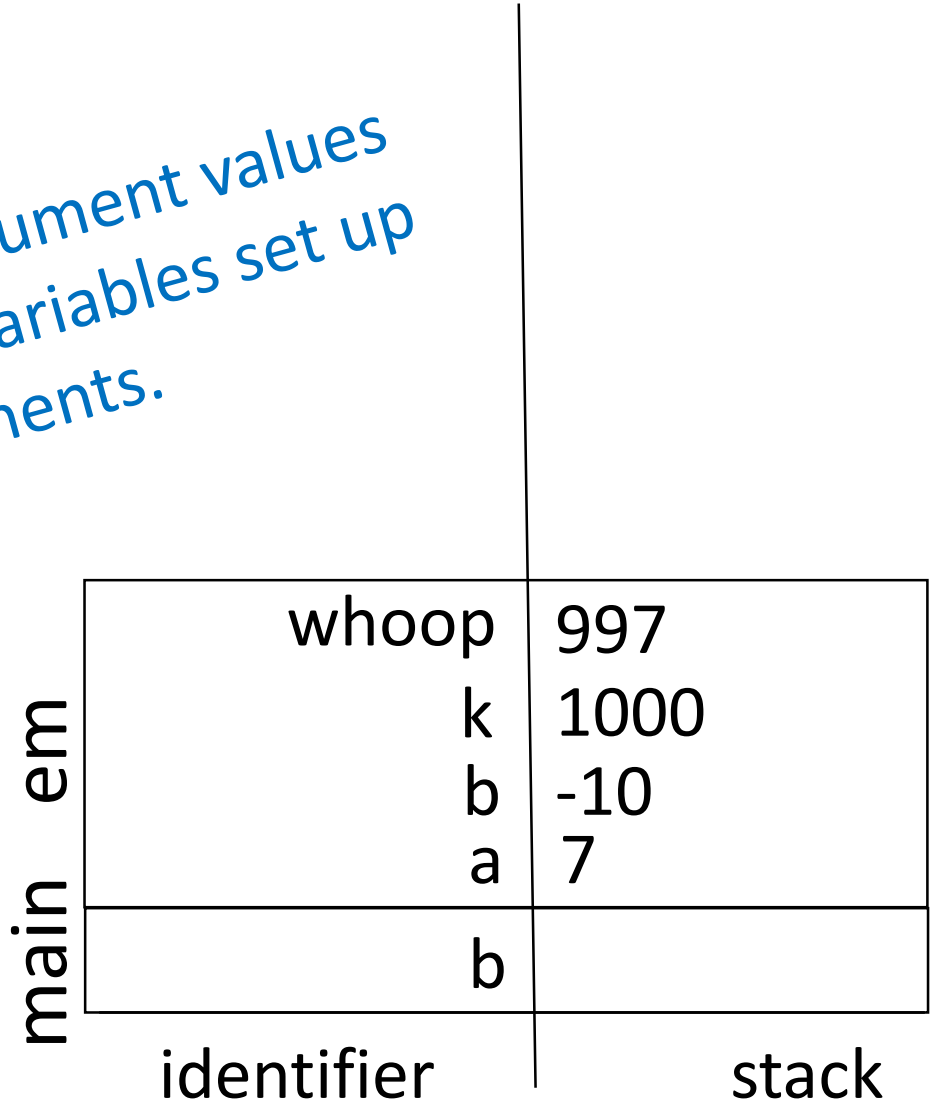


output

Program

```
int em(int a, int b) {
    int k = 1000;
    int whoop = a + b + k;
    return whoop;
}

int gig(int rev) {
    int howdy = 22;
    return em(rev, howdy);
}

int main() {
    int b = gig(em(7, -10));
    cout << "b: " << b << endl;
    return 0;
}
```

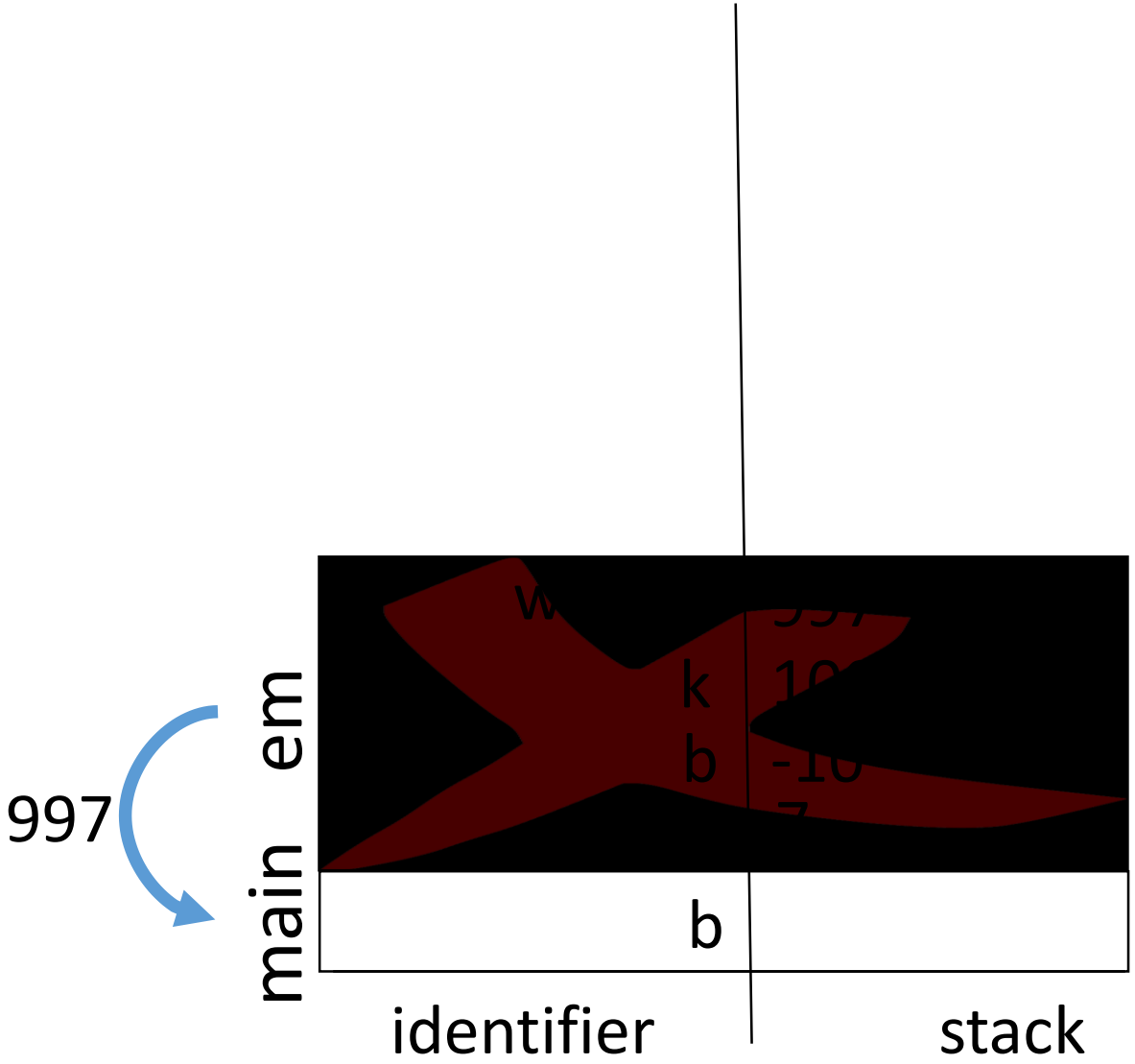
Notice that actual argument values are **copied** into the variables set up for the formal arguments.



output

Program

```
int em(int a, int b) {
    int k = 1000;
    int whoop = a + b + k;
    return whoop;
}

int gig(int rev) {
    int howdy = 22;
    return em(rev, howdy);
}

int main() {
    int b = gig(em(7, -10));
    cout << "b: " << b << endl;
    return 0;
}
```

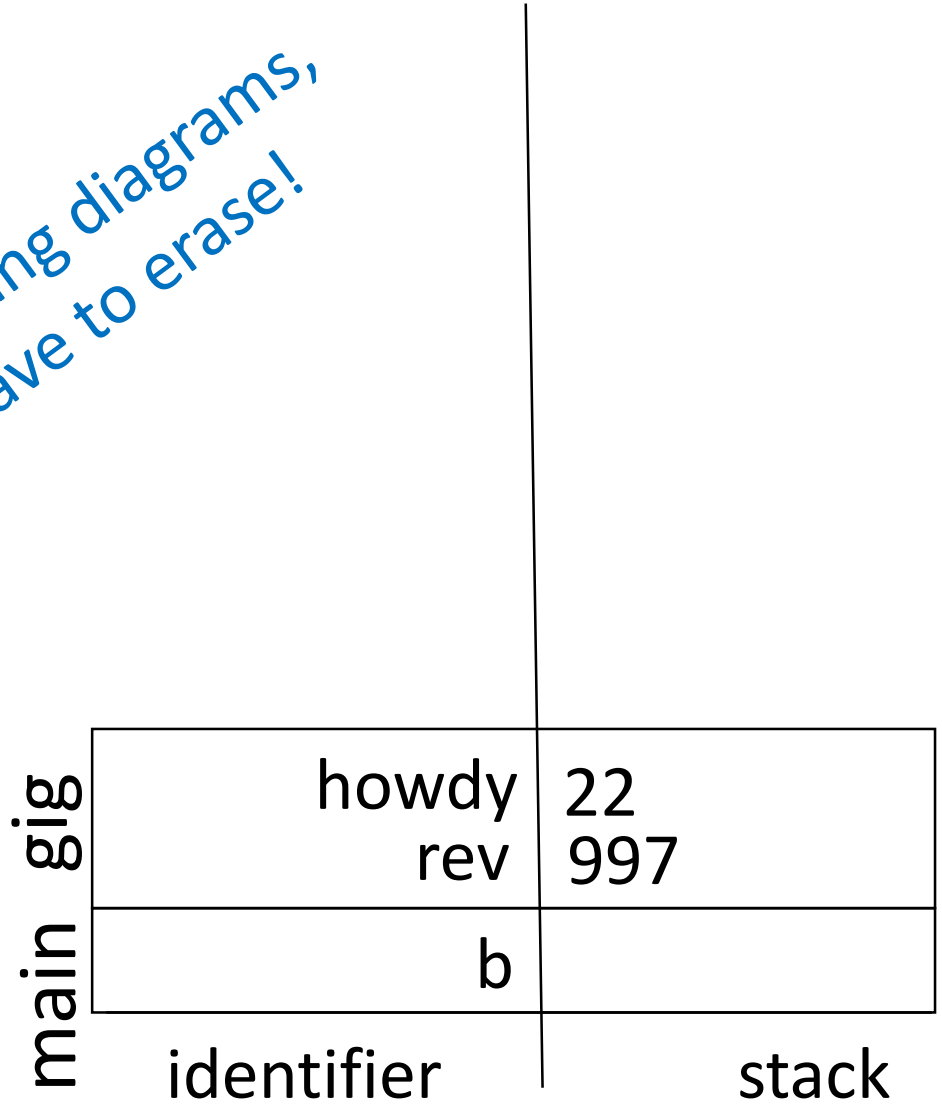


output

Program

```
int em(int a, int b) {
    int k = 1000;
    int whoop = a + b + k;
    return whoop;
}

int gig(int rev) {
    int howdy = 22;
    return em(rev, howdy);
}

int main() {
    int b = gig(em(7, -10));
    cout << "b: " << b << endl;
    return 0;
}
```

However, when sketching diagrams, we do not want to have to erase!

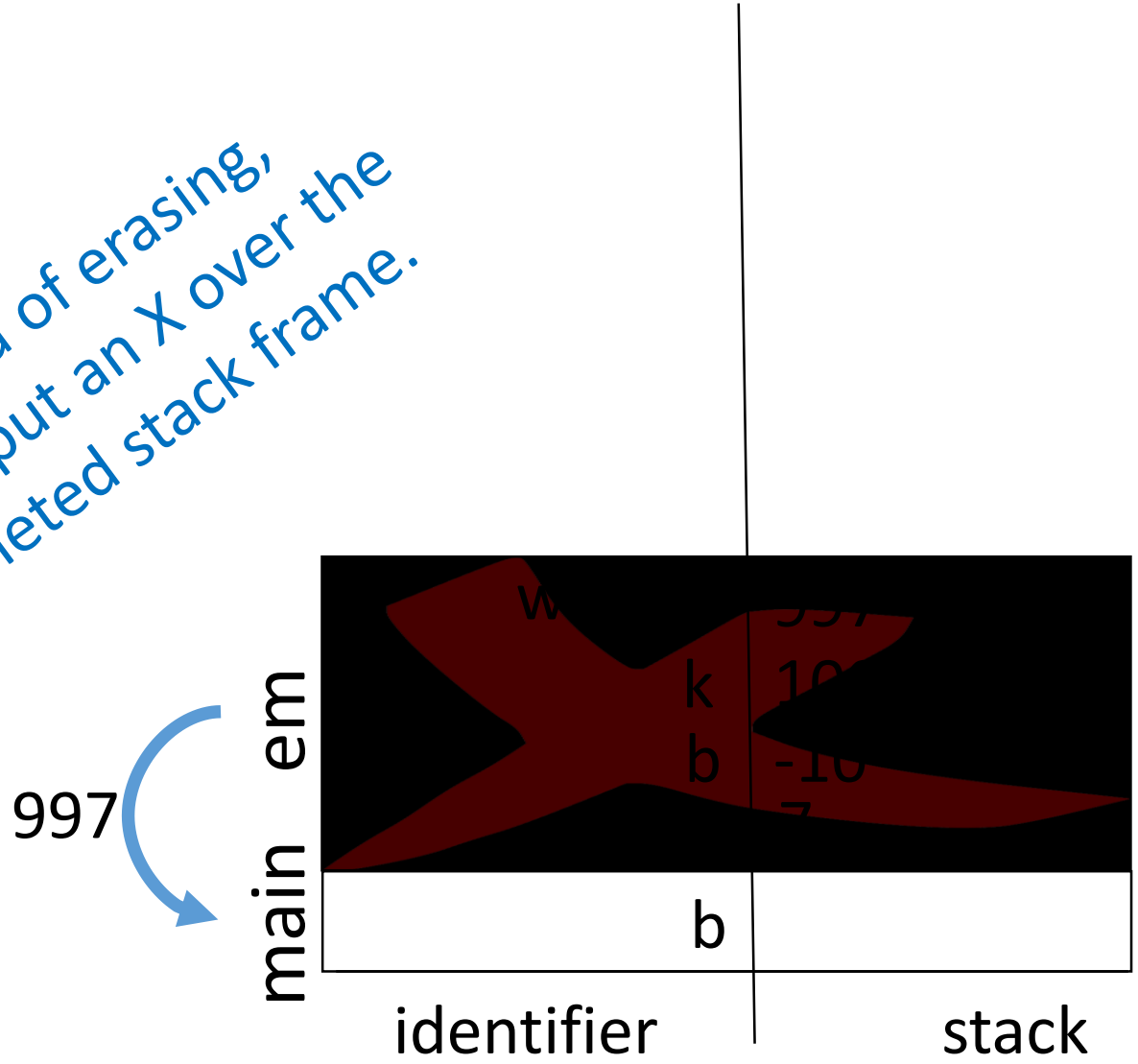


output

Program

```
int em(int a, int b) {
    int k = 1000;
    int whoop = a + b + k;
    return whoop;
}

int gig(int rev) {
    int howdy = 22;
    return em(rev, howdy);
}

int main() {
    int b = gig(em(7, -10));
    cout << "b: " << b << endl;
    return 0;
}
```

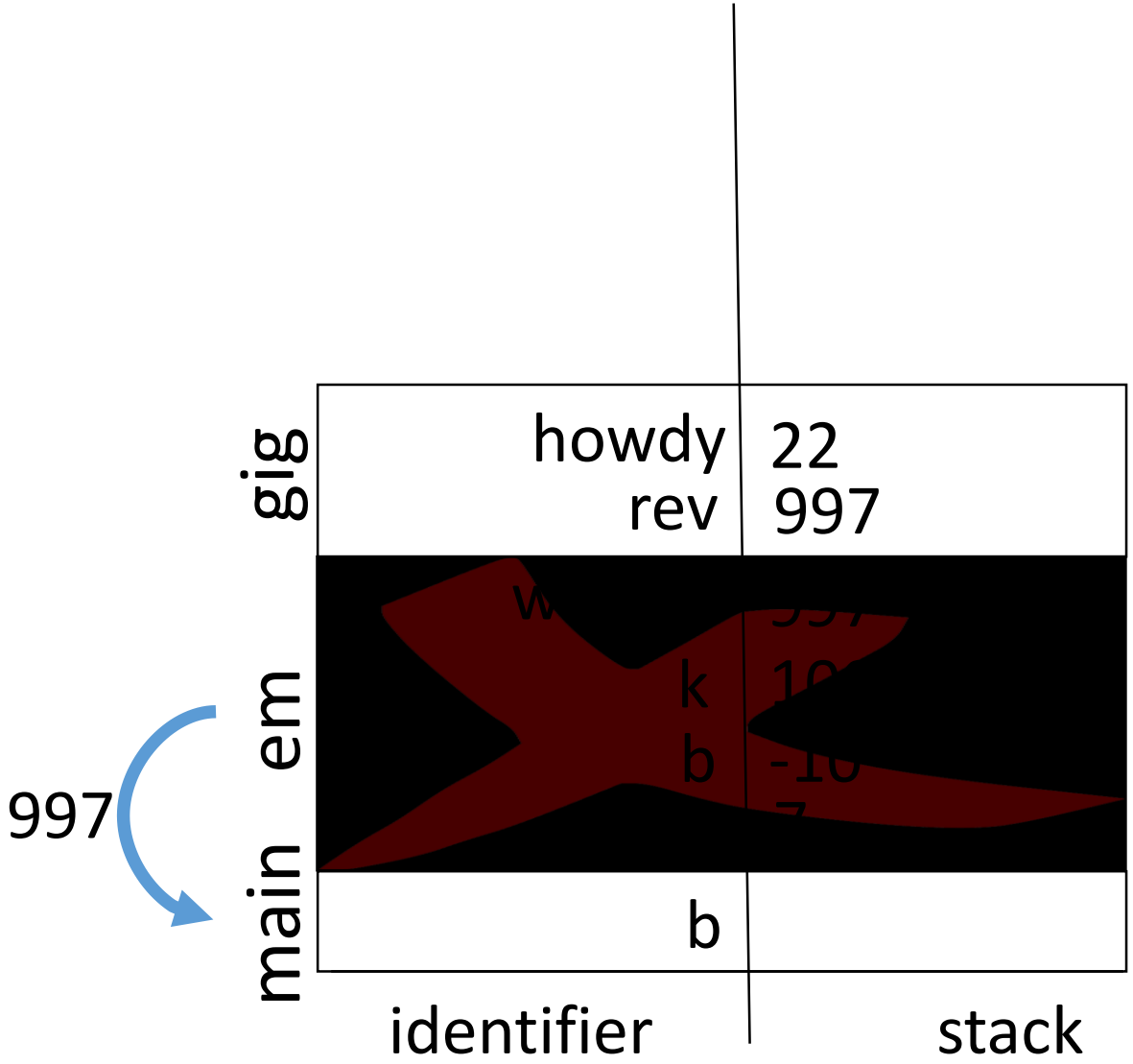
Instead of erasing, just put an X over the deleted stack frame.



output

Program

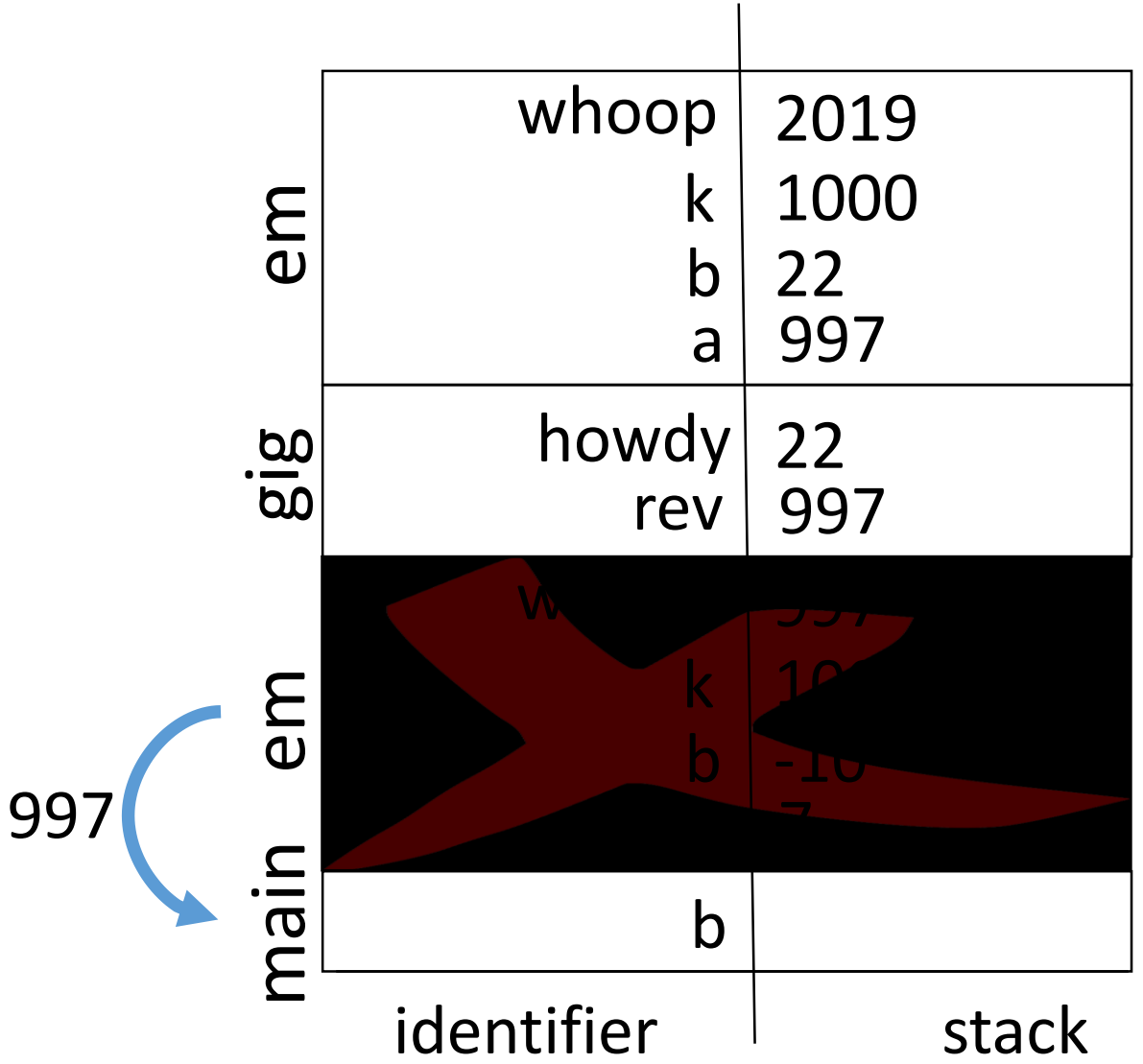
```
int em(int a, int b) {
    int k = 1000;
    int whoop = a + b + k;
    return whoop;
}

int gig(int rev) {
    int howdy = 22;
    return em(rev, howdy);
}

int main() {
    int b = gig(em(7, -10));
    cout << "b: " << b << endl;
    return 0;
}
```



output

Program

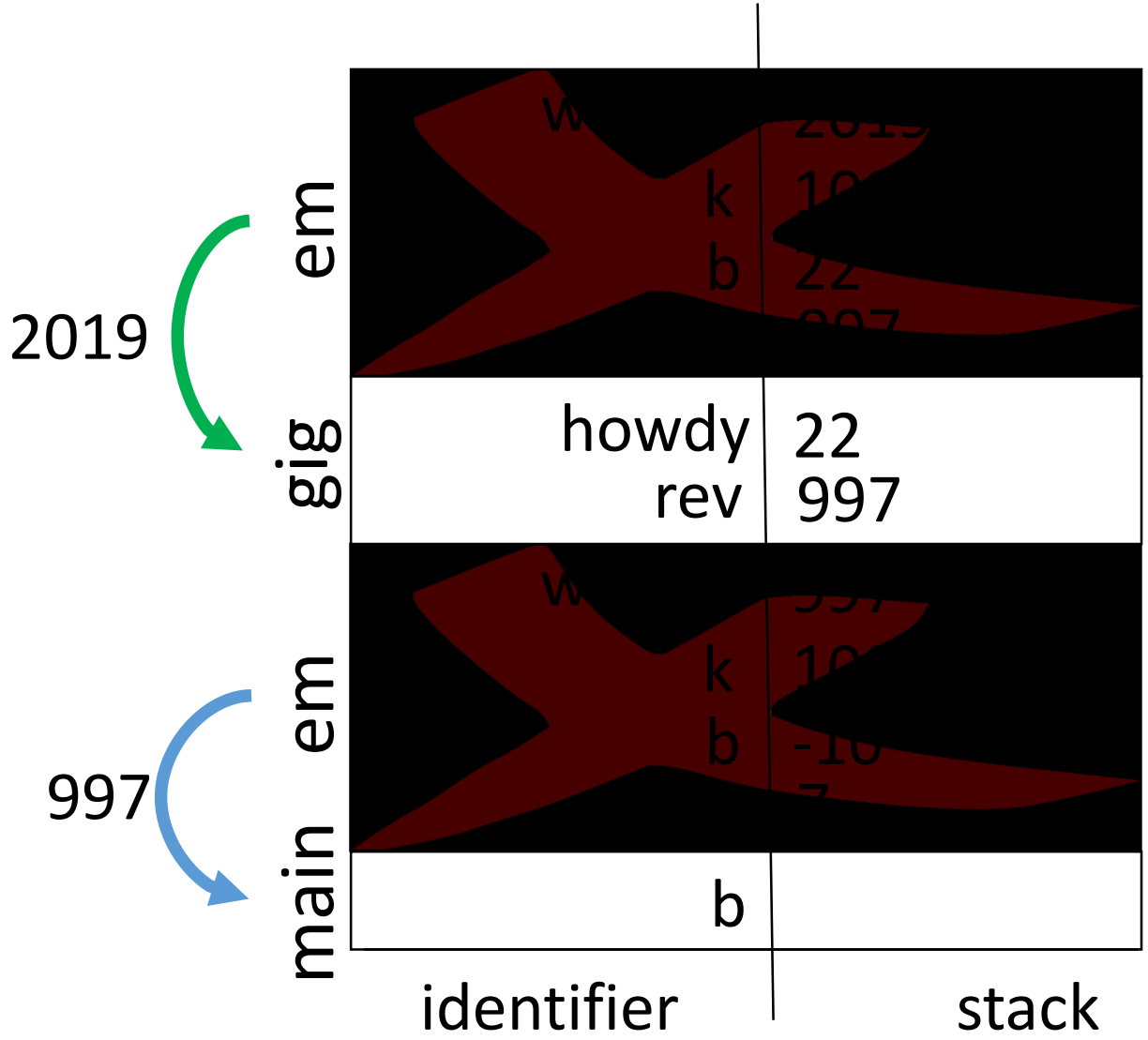
```
int em(int a, int b) {
    int k = 1000;
    int whoop = a + b + k;
    return whoop;
}

int gig(int rev) {
    int howdy = 22;
    return em(rev, howdy);
}

int main() {
    int b = gig(em(7, -10));
    cout << "b: " << b << endl;
    return 0;
}
```



output

Program

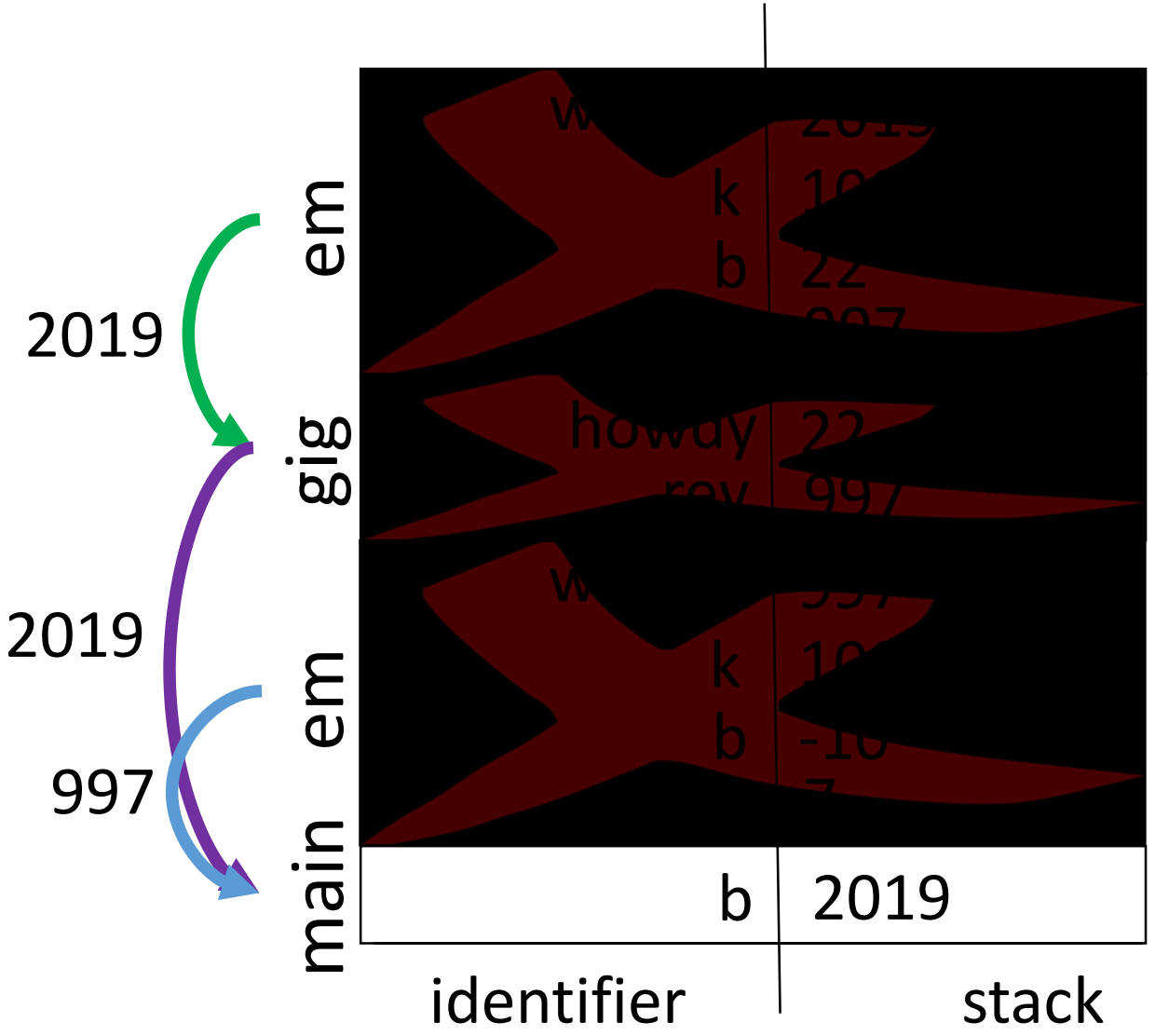
```
int em(int a, int b) {
    int k = 1000;
    int whoop = a + b + k;
    return whoop;
}

int gig(int rev) {
    int howdy = 22;
    return em(rev, howdy);
}

int main() {
    int b = gig(em(7, -10));
    cout << "b: " << b << endl;
    return 0;
}
```



output

Program

```
int em(int a, int b) {
    int k = 1000;
    int whoop = a + b + k;
    return whoop;
}

int gig(int rev) {
    int howdy = 22;
    return em(rev, howdy);
}

int main() {
    int b = gig(em(7, -10));
    cout << "b: " << b << endl;
    return 0;
}
```



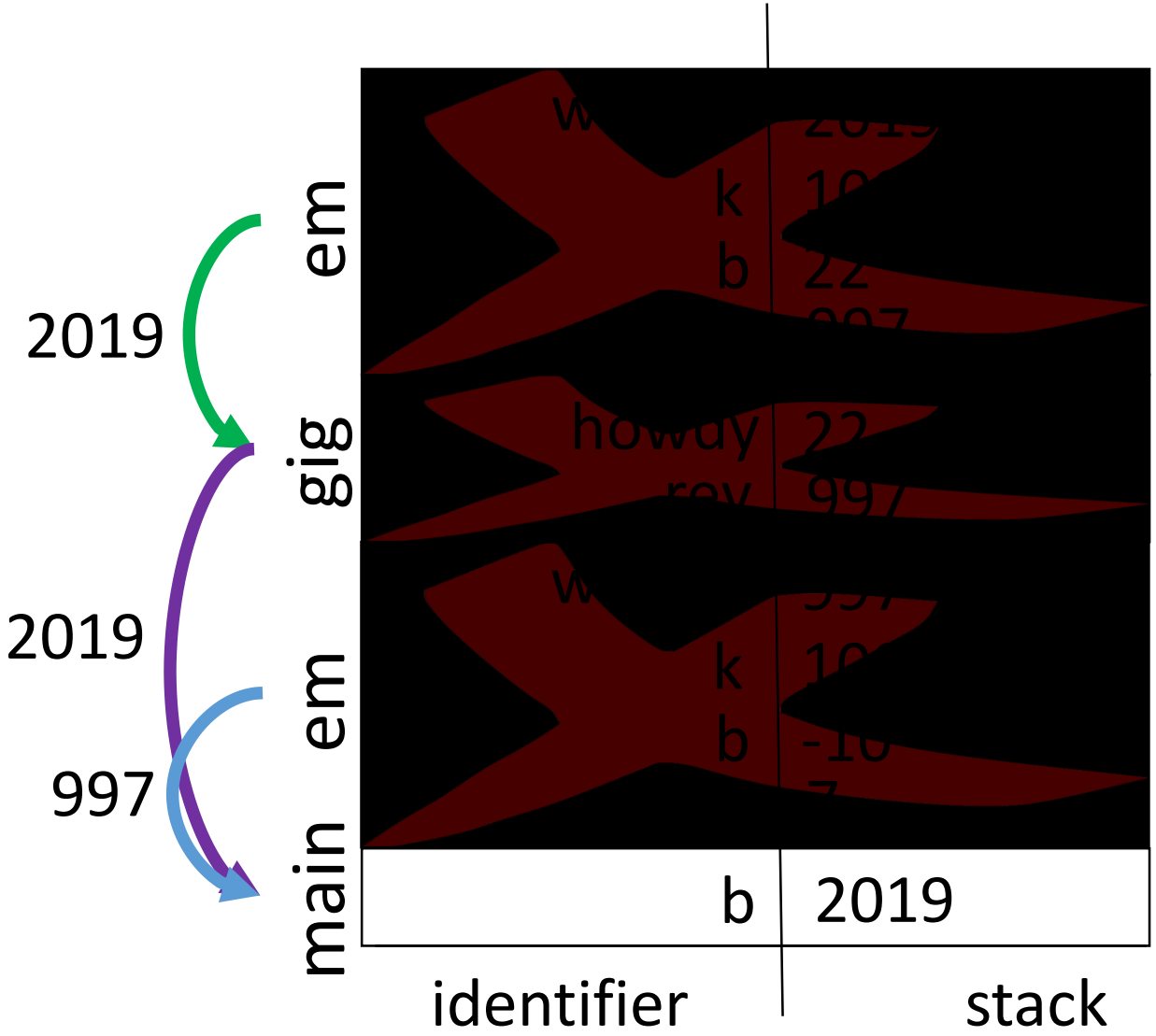
output

b: 2019

Program

We will expand on this model as the semester progresses.

You will see memory diagrams on exams



Memory Diagrams 2
Pointers and References

Memory Diagram

- How do pointers and references affect the memory diagram?
- Pointers store an address.
  - Rather than worry about a particular number:
    - Use a small circle to indicate the value is an address.
    - Use an arrow from the circle that points to the location/address.
- References are similar to pointers
  - Some refer to them as “safe pointers”
  - Technically, references are an alias to a memory location
  - We’ll draw it like a pointer, even though it’s technically not

Program

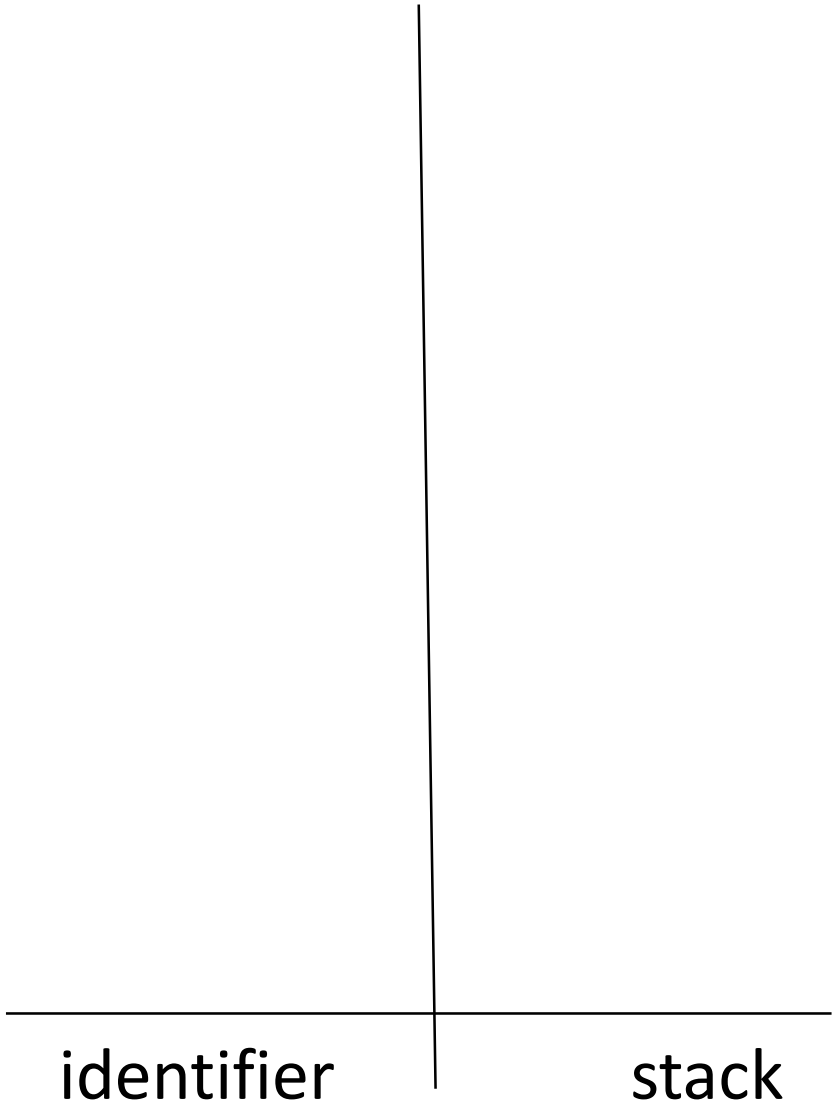
```
int mi(int j) {
  int i = 5;
  return j % i;
}

int re(int& s, int p)
{
  s = 12;
  return mi(s*p);
}
```

```
int doe(int w) {
  int k = 2;
  int q = w+3;
  cout << "k: " << k << endl;
  int z = re(k, q);
  cout << "k: " << k << endl;
  return z + w;
}

int main() {
  int b = doe(11);
  cout << "b: " << b << endl;
}
```

output

main

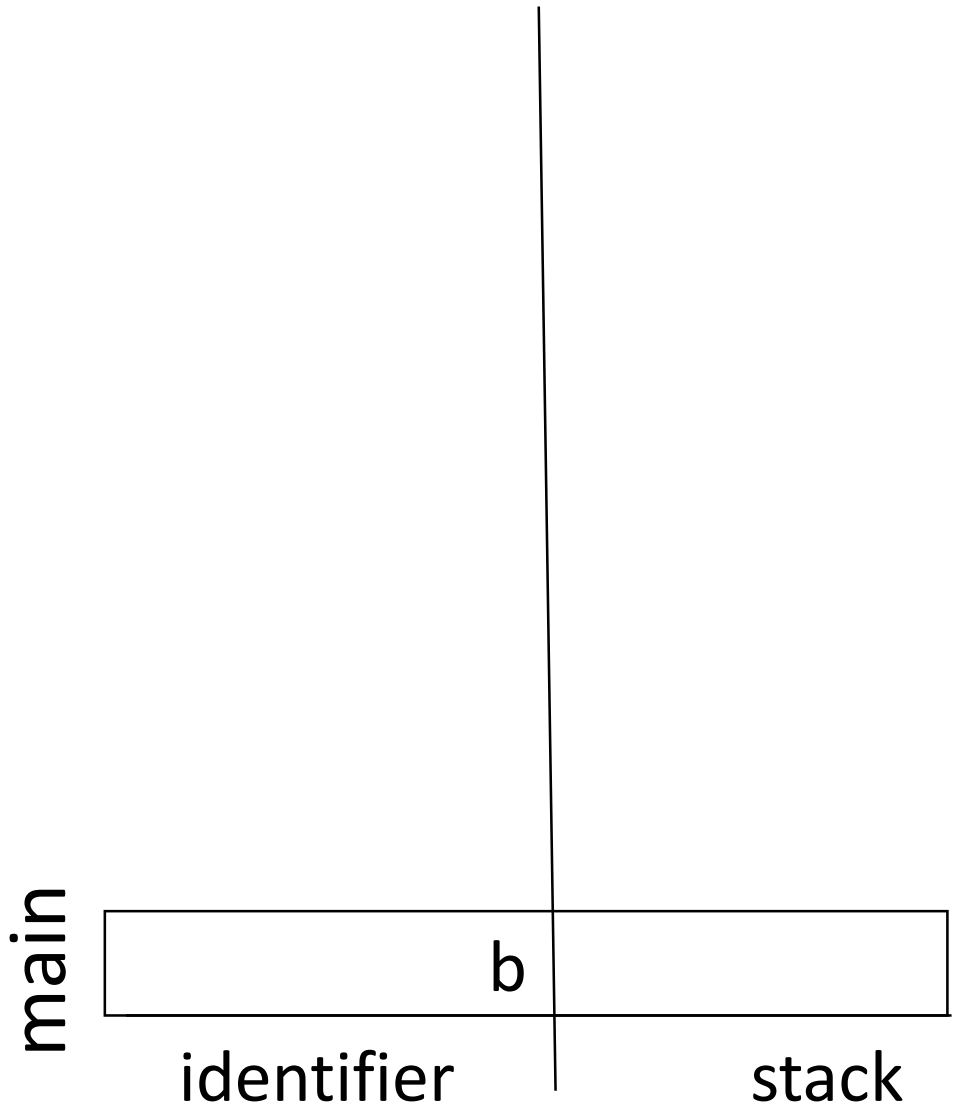
```
int main() {
  int b = doe(11);
  cout << "b: " << b << endl;
}
```



output

main

```
int main() {
  int b = doe(11);
  cout << "b: " << b << endl;
}
```

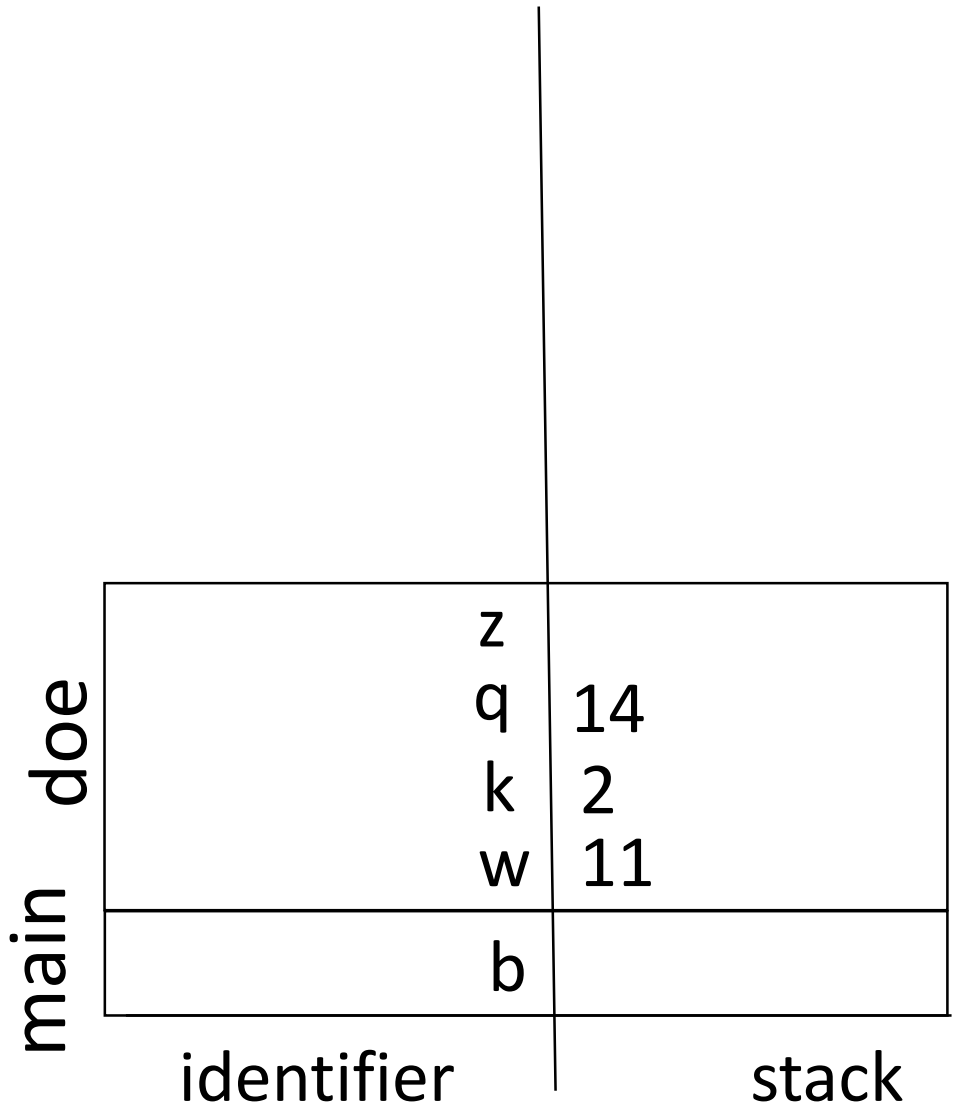


doe

output

k: 2

```
int doe(int w) {
  int k = 2;
  int q = w+3;
  cout << "k: " << k << endl;
  int z = re(k, q);
  cout << "k: " << k << endl;
  return z + w;
}
```

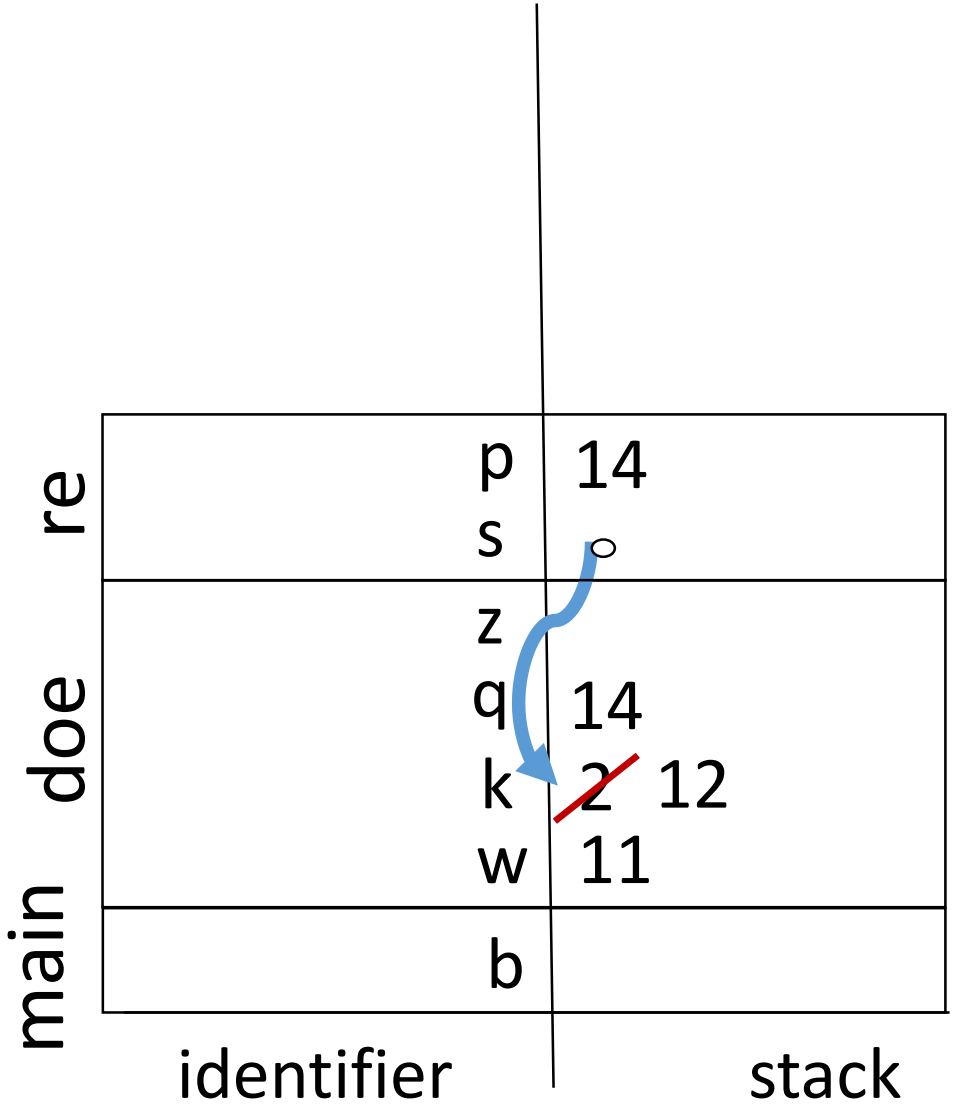


output

re

k: 2

```
int re(int& s, int p) {
  s = 12;
  return mi(s*p);
}
```

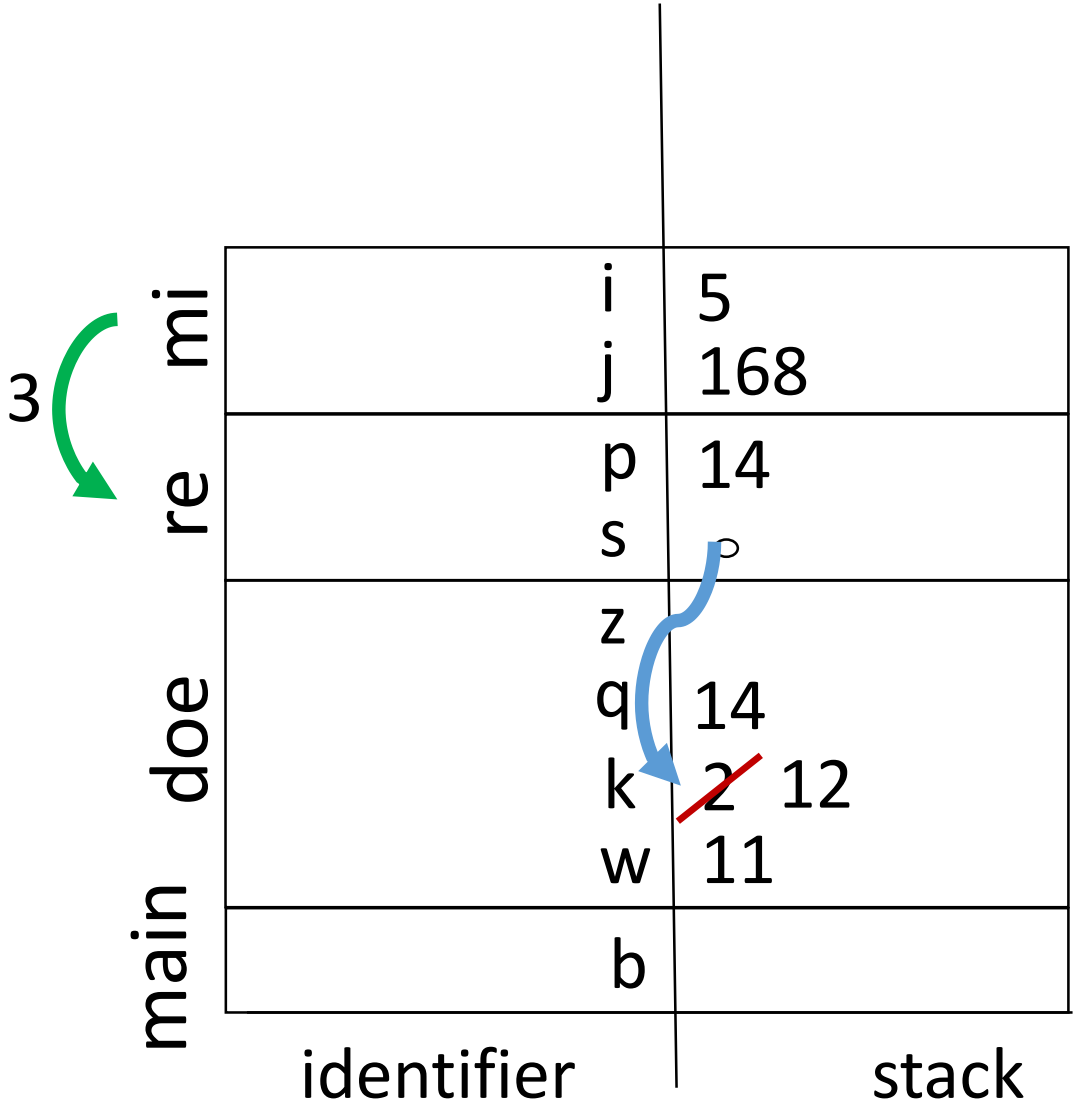


output

k: 2

mi

```
int mi(int j) {
  int i = 5;
  return j % i;
}
```

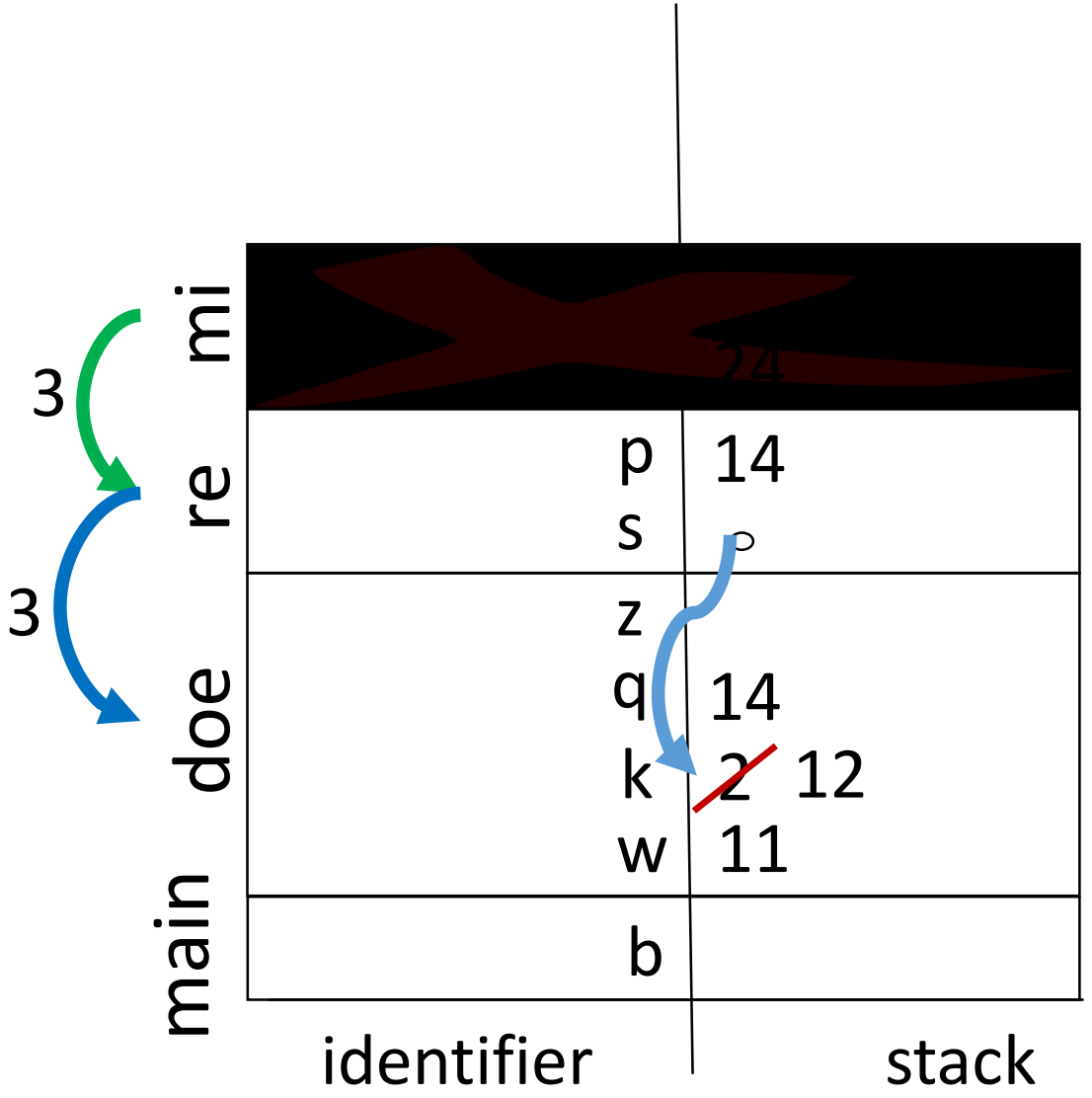


output

re

k: 2

```
int re(int& s, int p) {
  s = 12;
  return mi(s*p);
}
```



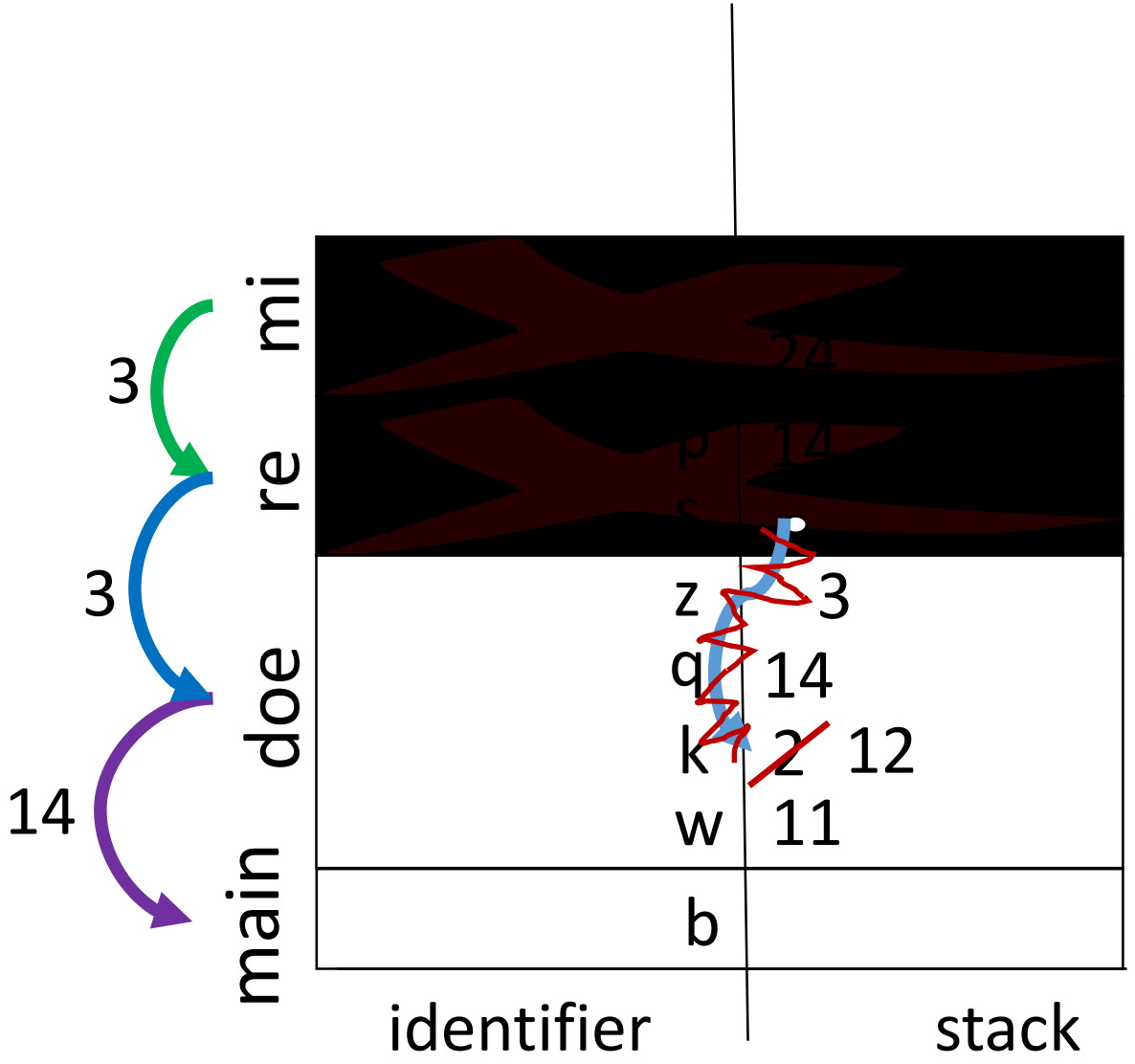
doe

output

k: 2

k: 12

```
int doe(int w) {
   int k = 2;
   int q = w+3;
   cout << "k: " << k << endl;
   int z = re(k, q);
   cout << "k: " << k << endl;
   return z + w;
}
```



main

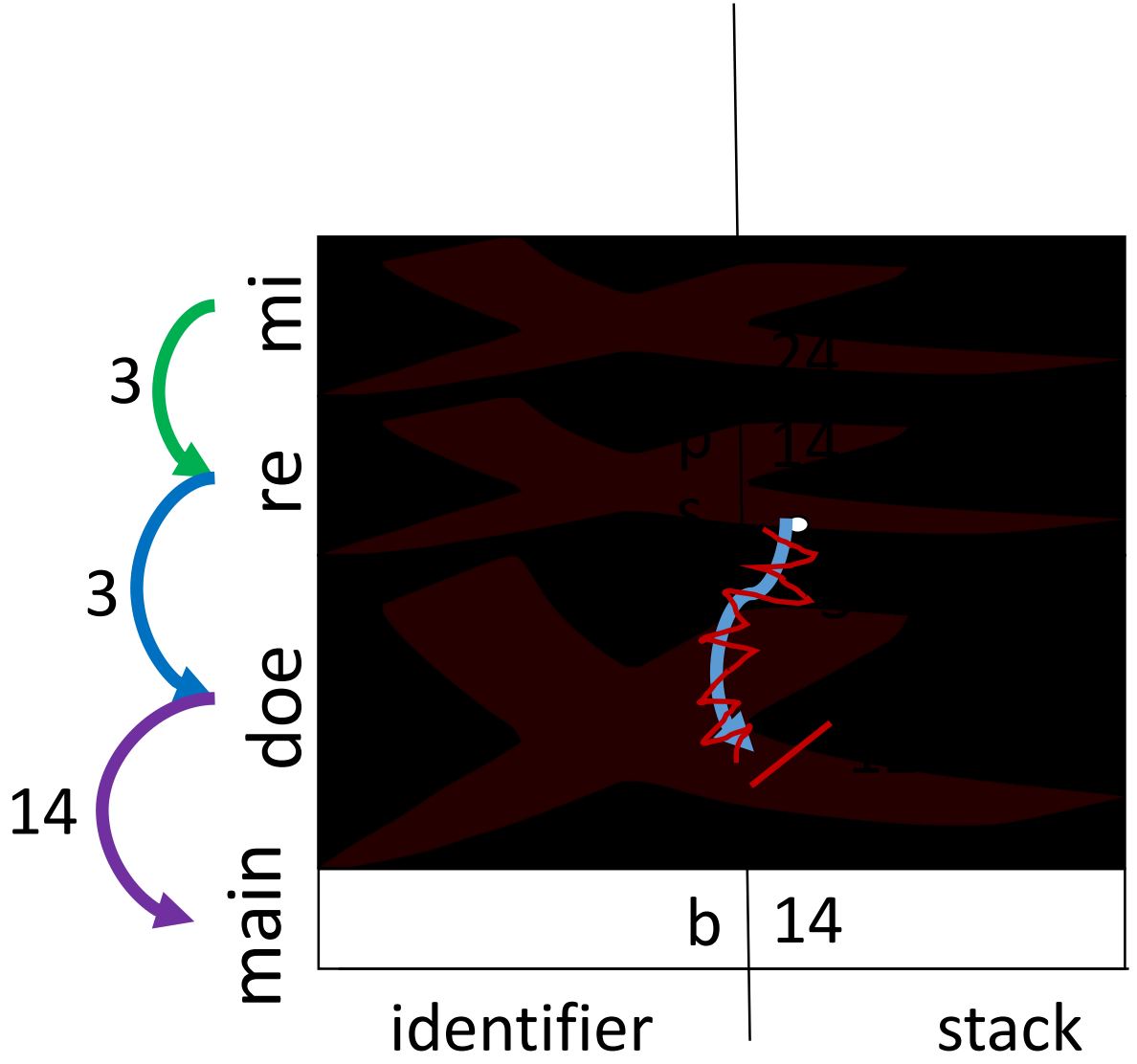
output

k: 2

k: 12

b: 14

```
int main() {
   int b = doe(11);
   cout << "b: " << b << endl;
}
```



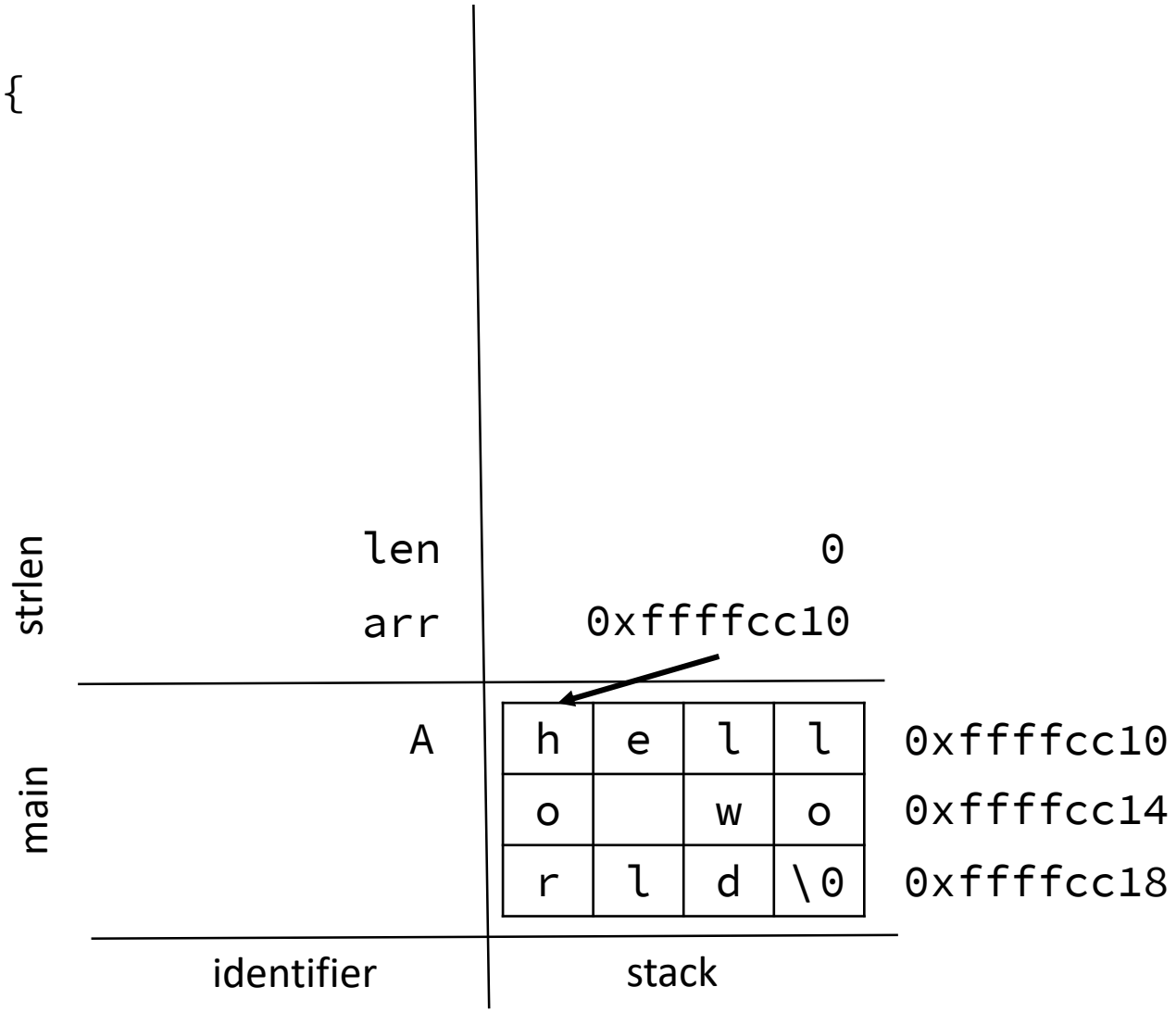
Memory Diagrams 3
Arrays & Structs

How are arrays represented in the memory diagram?

- “An array is just a pointer”
  - Not the whole story…
- The space allocated for an array goes on the stack or the heap
  - Stack: working memory for functions
  - Heap: free store, dynamically allocated memory
- An array on the heap is stored as a pointer on the stack to a contiguous block of memory on the heap
- An array on the stack is stored as a contiguous block of memory on the stack
  - No need to store the address

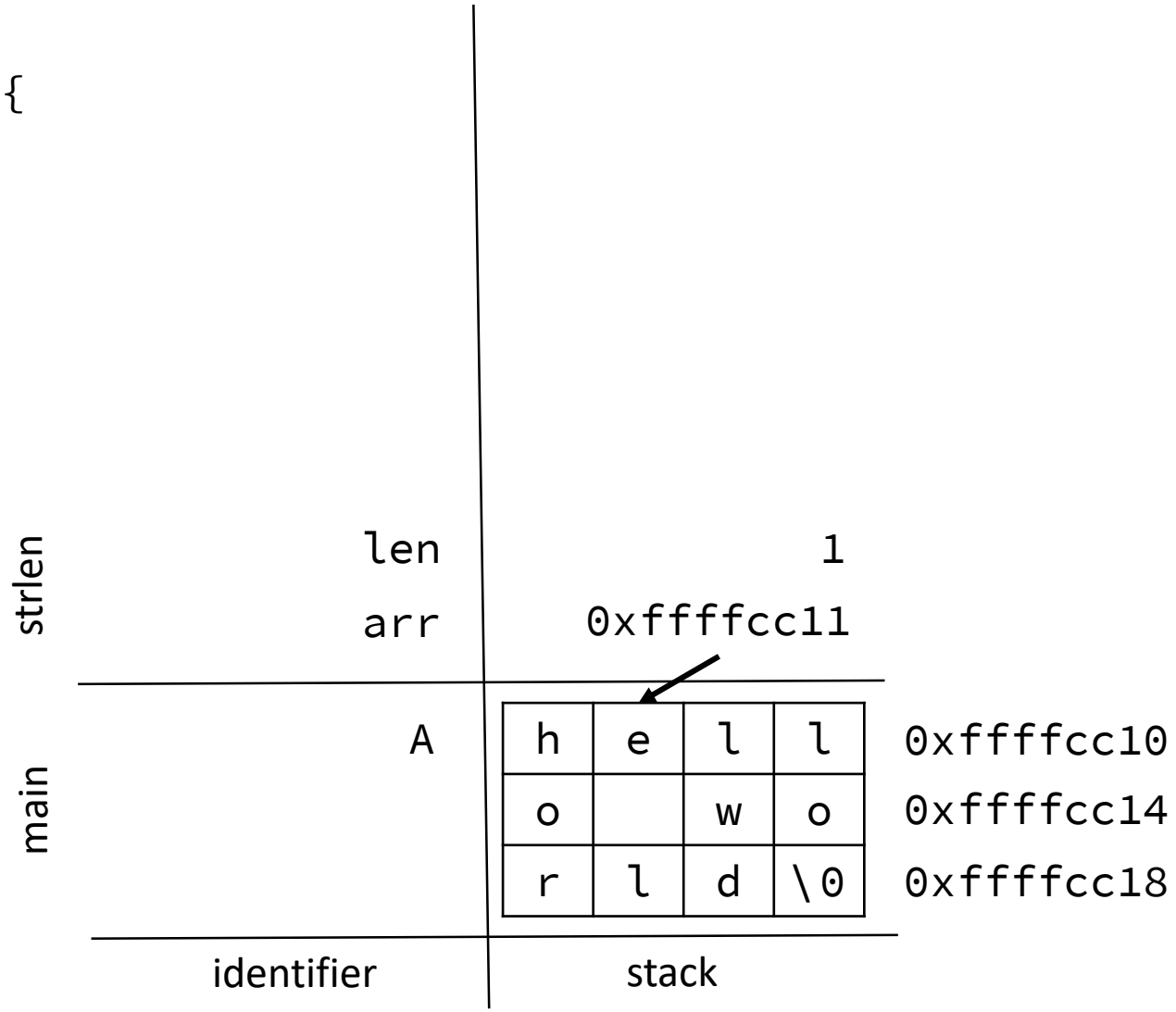
A memory diagram with an array on the stack

```
unsigned int strlen(const char* arr) {
    unsigned int len = 0;
    while (*arr) {
        len++
        arr++;
    }
    return len;
}

int main() {
    char A[] = “hello world”;
    strlen(A);
    return 0;
}
```



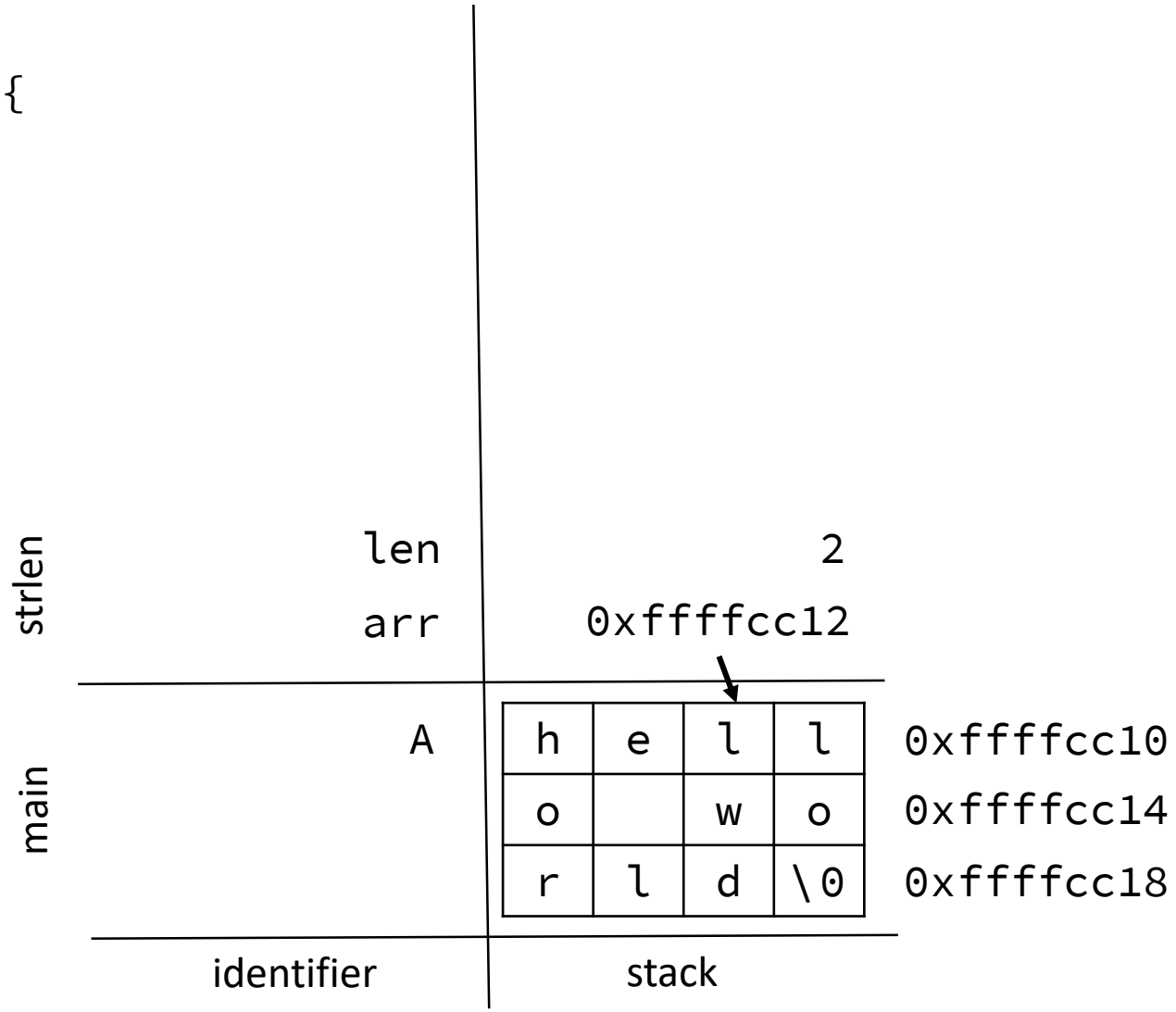
A memory diagram with an array on the stack

```
unsigned int strlen(const char* arr) {
    unsigned int len = 0;
    while (*arr) {
        len++
        arr++;
    }
    return len;
}

int main() {
    char A[] = “hello world”;
    strlen(A);
    return 0;
}
```



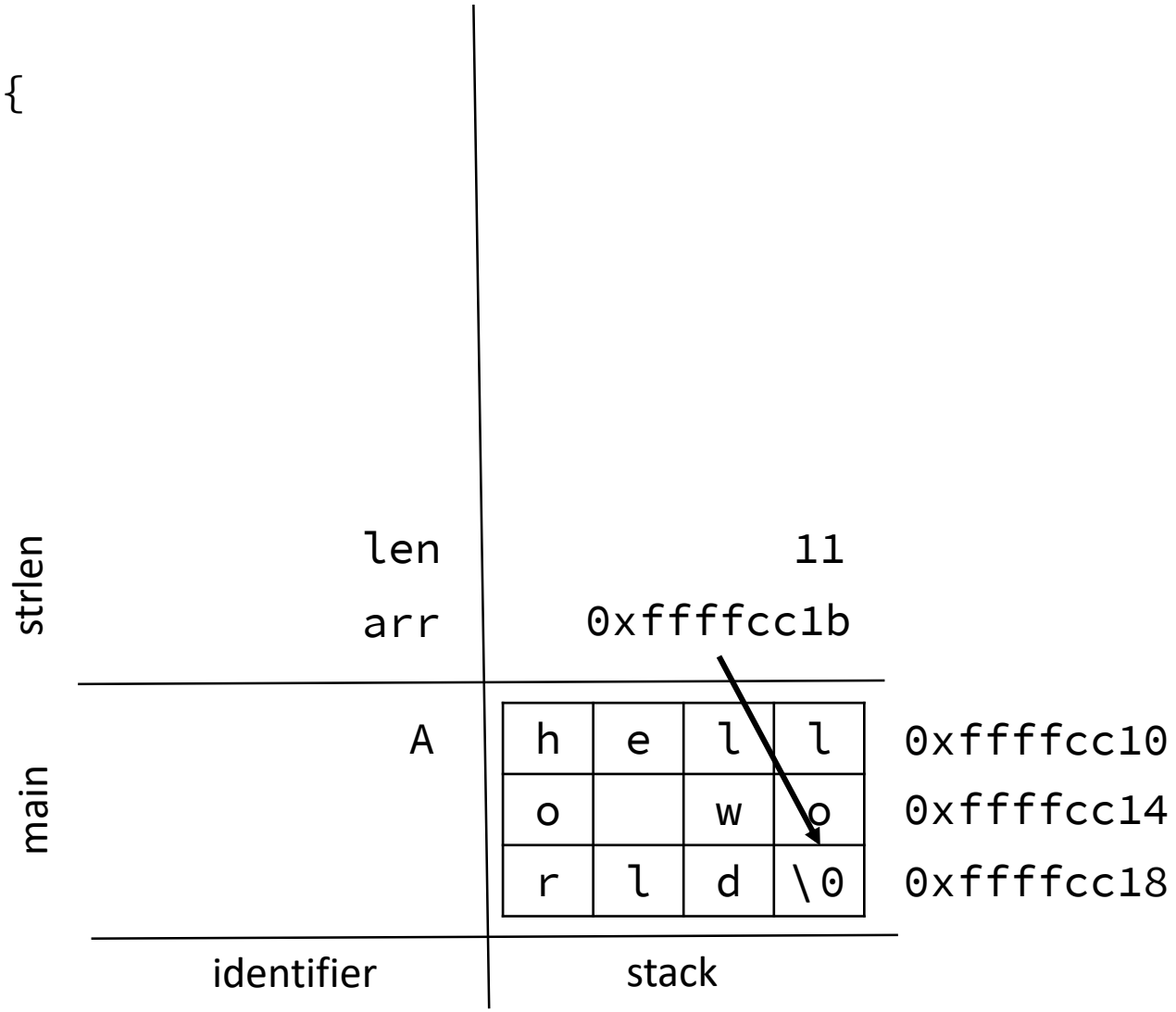
A memory diagram with an array on the stack

```
unsigned int strlen(const char* arr) {
    unsigned int len = 0;
    while (*arr) {
        len++
        arr++;
    }
    return len;
}

int main() {
    char A[] = “hello world”;
    strlen(A);
    return 0;
}
```



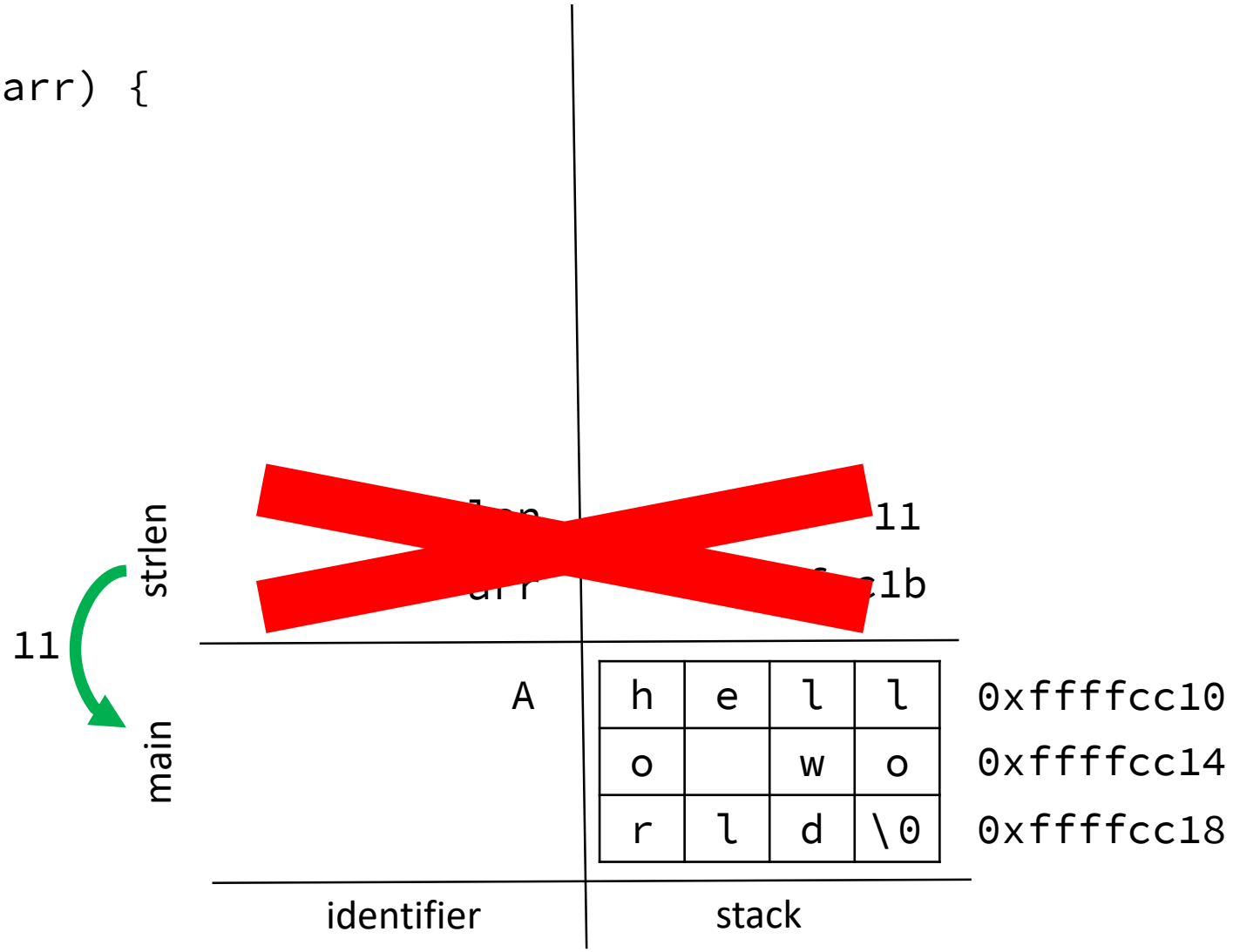
A memory diagram with an array on the stack

```
unsigned int strlen(const char* arr) {
    unsigned int len = 0;
    while (*arr) {
        len++
        arr++;
    }
    return len;
}

int main() {
    char A[] = “hello world”;
    strlen(A);
    return 0;
}
```



A memory diagram with an array on the stack

```
unsigned int strlen(const char* arr) {
    unsigned int len = 0;
    while (*arr) {
        len++
        arr++;
    }
    return len;
}

int main() {
    char A[] = “hello world”;
    strlen(A);
    return 0;
}
```



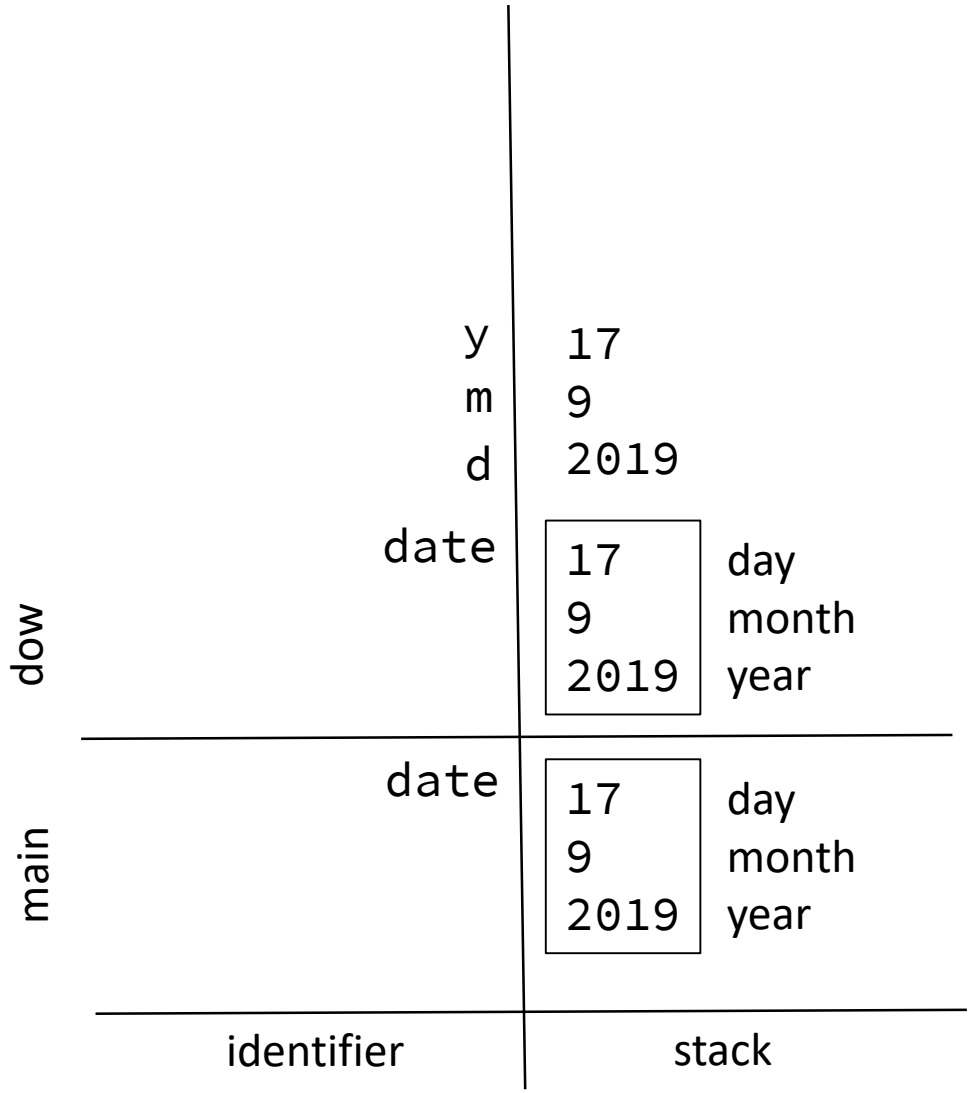
How are structs represented in the memory diagram?

- Structs are bundles of data
  - Can also have functions… struct + functions = class
- Data in a struct is stored in contiguous memory
  - Space is allocated for each field (member variable / attribute) of the struct

```
typedef struct Date {
    short day;
    short month;
    short year;
} Date;
```

A memory diagram with a struct on the stack

```
short dow(const Date date) {
    short d = date.day;
    short m = date.month;
    short y = date.year;
    y -= m<3;
    return (y+y/4-y/100+y/400+
        "-bed=pen+mad."[m]+d)%7;
}

int main() {
    Date date = {17, 9, 2019};
    cout << dow(date) << endl;
    return 0;
}
```



A memory diagram with a struct on the stack

```
short dow(const Date date) {
    short d = date.day;
    short m = date.month;
    short y = date.year;
    y -= m<3;
    return (y+y/4-y/100+y/400+
        "-bed=pen+mad."[m]+d)%7;
}

int main() {
    Date date = {17, 9, 2019};
    cout << dow(date) << endl;
    return 0;
}
```