Program Tracing

This handout is based on the ideas presented in An Explicit Strategy to Scaffold Novice Program Tracing (https://dl.acm.org/citation.cfm?id=3159527) *by Benjamin Xie, Greg L. Nelson, and Andrew J. Ko.*

Program tracing is the process of executing program code by hand, with concrete inputs. Similar to how it is important for children to have the basic skills of reading before they can start writing their own sentences, it is important to be able to execute existing code by hand, before writing own code. This handout explains how to trace C++ programs with pen and paper, and check that traces are correct with C++ (http://pythontutor.com/cpp.html#mode=edit) Tutor (http://pythontutor.com/cpp.html#mode=edit) (a version of Python Tutor (http://pythontutor.com/) for C++).

In the first section we start with tracing the most basic programs. Then in each following section we extend our approach to handle more and more complicated language features.

Basic

In this section we will learn how to execute the most basic C++ programs, such as the following one, by hand:

```
#include <iostream>
int main() {
  int a;
  a = 5;
  int b;
  b = 3;
  a = a + b;
  std::cout << a << std::endl;
  return 0;
}
```

One of the hard aspects of program tracing is precisely tracking how the program state changes during execution. The program state of simple programs such as the one above is the values stored in the variables. We can represent this state by using a simple table:

Variable	Value
...	...

Now we can execute the program, statement by statement, and update the table:

1. The first statement in the program is `int a;`. We execute this statement by adding a new row with variable `a` and no value to our table:

Variable	Value
a	

   Why do we leave the *Value* column empty? Because variable `a` is *uninitialised*, and if our executions reaches a program point where we read an uninitialised variable, we know that our program has a bug because reading an uninitialised variable is an illegal operation in C++ (in C++ jargon: the execution has *undefined behaviour*).

2. The second statement is `a = 5;`. We execute it by finding the row with variable `a` and adding `5` to its *Value* column:

Variable	Value
a	5

   If the table did not have a row with a variable `a`, that would indicate a bug in the program.

3. The next two statements, `int b;` and `b = 3;`, are executed analogously:

Variable	Value
a	5
b	3

4. The next statement is `a = a + b`. First we evaluate the assignment's right hand side, i.e. the expression `a + b`. To do that, we look up the values of `a` and `b` in the table and add them: `a + b` evaluates to `5 + 3`, and then to `8`. Since the computed value is assigned to (stored in) variable `a`, we cross out the variable's old value `5` and write down its new value `8`. The updated table looks as follows:

Variable	Value
a	~~5~~, 8
b	3

5. The last interesting statement is `std::cout << a << std::endl;`, which writes the current value of `a`, followed by a line break (`std::endl`), to standard output (`std::cout`). Since a program can only create additional output, but cannot change what has already been output, we can track the program output by always appending it to a dedicated table row, like this:

Program Output
8

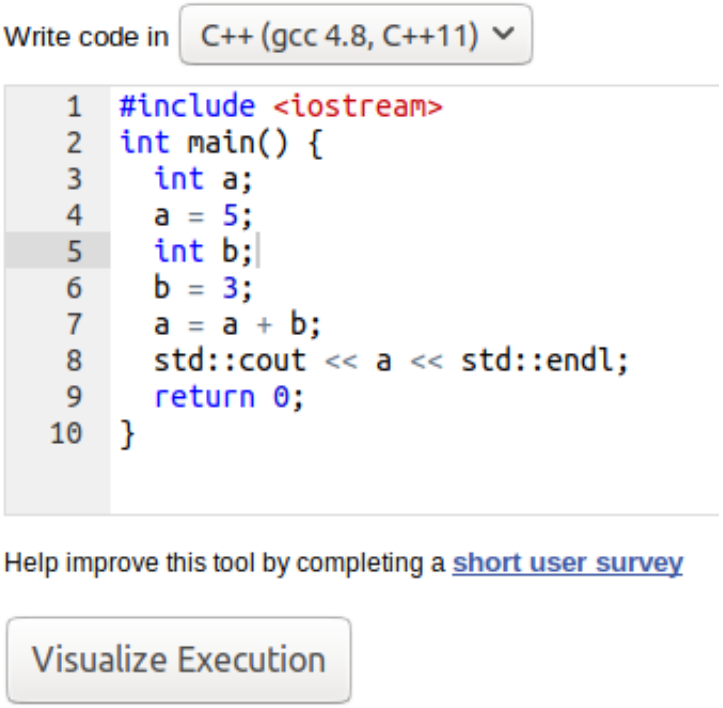
6. The final statement in the program is `return 0;`: it exits the `main` function and terminates the program (and we can ignore the returned `0` in this case).

Checking Program Traces with C++ Tutor

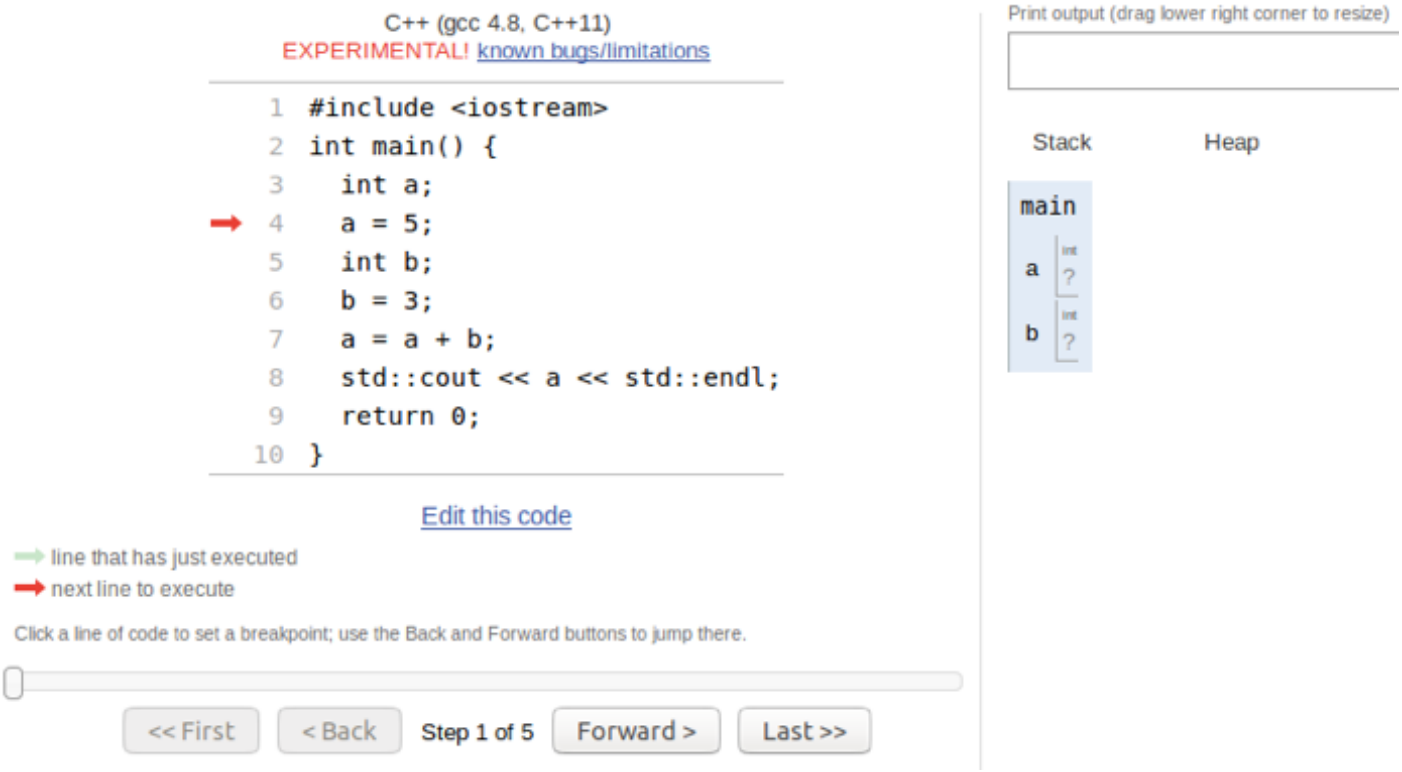
To check whether your program trace is correct, you can use C++ Tutor:
(http://pythontutor.com/cpp.html#mode=edit)

1. Open the C++ Tutor website http://pythontutor.com/cpp.html#mode=edit
(http://pythontutor.com/cpp.html#mode=edit)

2. Enter your code into the text field:



3. Click on *Visualize Execution*. The following view should be displayed:



4. You can step through the program execution by using the *"Forward >"* button. The values displayed in the blue *Stack* table should match the not crossed-out values in our table above.

Exercises

In exams, typical questions related to program tracing are "What is the output of this program?" and "What value will variable `x` have at a specific program point?".

Task 1

What values will variables `a` and `b` have at the end of this program:

```
int main() {
  int a;
  a = 5;
  int b;
  b = 3;
  int c;
  c = a;
  a = b;
  b = c;
  return 0;
}
```

Execute the program on paper, and then check your solution using C++ Tutor.

Scopes

Note: Scopes are going to be introduced in the third week of the course.

In C++, it is possible to *shadow* a variable by declaring a new variable with the same name in a nested scope. For example, in the following program, variable `a` of type `int` is shadowed by variable `a` of type `double`:

```
#include<iostream>
int main() {
  int a = 1;
  {
    double a;
    a = 5;
    std::cout << a << std::endl;
  }
  std::cout << a << std::endl;
  return 0;
}
```

If we execute this program, it will output the numbers `5` and `1`. However, if we traced the program as described in the previous section, we would conclude that it outputs `5` and `5`. Therefore, we need to extend our tracing approach from the previous section to handle scopes. We can do this as follows:

1. We execute the first statement `int a = 1;` as before. Since it declares a variable *and* initialises it with a value, we do the following:
    1. Add a new row for variable `a` at the **bottom** of the program state table.
    2. Mark that `1` was assigned to `a` by updating the *Value* column to 1.
    Our table now looks as follows:

Variable	Value
a	1

2. As a next step, the execution will enter a new block (whose start is marked by `{`), and thereby a new scope. We can reflect this by appending a new row that contains `{` to the **bottom** of our table:

Variable	Value
a	1
{	

3. The next statement in our program is `double a;`. Since it is a variable declaration, we add a new row for variable `a` at the **bottom** of our table:

Variable	Value
a	1
{	
a	

4. We execute `a = 5;` by updating the *Value* column of the **lowest not crossed-out** row that contains variable `a`:

Variable	Value
a	1
{	
a	5.0

5. Similarly to the previous step, to look up the value of `a` we take the lowest not crossed-out row that contains variable `a`. Therefore, `std::cout << a << std::endl;` will output `5`:

Program Output
5

6. As a next step, the execution will exit the block (whose end is marked by `}`), and thereby the corresponding scope. We reflect this in our state table as follows:

    1. We find the lowest row that contains a not crossed-out `{`.
    2. We cross out that row and all rows that are below it starting from the lowest one. The updated table looks as follows:

Variable	Value
a	1
~~b~~	
~~a~~	~~5.0~~

7. The next statement is `std::cout << a << std::endl;`, which writes the value of variable `a` to standard output. To find the value of `a`, we take the lowest not crossed-out row, which, in this case, contains value `1`. Therefore, the statement will output `1` to standard output and the final output of the program will be:

Program Output
5 1

8. The last statement is again `return 0;`, which ends the program execution.

Note: Unfortunately, C++ Tutor visualises the programs that shadow variables incorrectly. Therefore, you cannot use it to check whether your traces are correct.

Function Calls

Note: Function calls are going to be introduced in the fifth week of the course.

In C++, one of the constructs for structuring code is functions. For example, we can *define* a function that finds a larger of two integers as follows:

```
int max(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

The `int max(int a, int b)` is called the function *signature*. The word before the parenthesis (`max`)is the function *name*. `int` before the function name is the function *return type*. Between the parenthesis we have a declaration of function *parameters*, in this case, the function has two parameters of type `int` : `a` and `b` .
After we declared the function, we can call it to compute the result for us. For example, in the following `main` function we call function `max` with arguments `x` and `y` :

```
#include<iostream>
int max(int a, int b) { /* code from above */ }
int main() {
  int x = 4;
  int y = 8;
  int larger;
  larger = max(x, y);
  std::cout << larger << std::endl;
  return 0;
}
```

We can trace this program as follows:

1. We create a table as before, just now we additionally write the function signature with argument names omitted as the title of the table. For `main` , the initial table would look like this:

int main()	
Variable	**Value**

2. We execute the first statement `int x = 4;` as before and get the following table:

int main()	
Variable	**Value**
x	4

3. Similarly for the second `int y = 8;` and third `int larger;` statements. The table we have so far is:

int main()	
Variable	**Value**
x	4
y	8
larger	

4. Now, the next statement is `larger = max(x, y);` that is composed of a function call `max(x, y)` and assignment of the function result to the variable `larger`. First, we execute the function call. We do that as follows:

    1. We create a **new** table for the target function `max`:

int main()	
Variable	**Value**
x	4
y	8
larger	

int max(int, int)	
Variable	**Value**

    2. Then, we create a local variable for each parameter of the function `max`:

int main()	
Variable	**Value**
x	4
y	8
larger	

int max(int, int)	
Variable	**Value**
a	
b	

3. Then, we evaluate the arguments passed to the function and match them with the newly allocated variables. For example, the first argument to the function `max` is `x`, which evaluates to `4`. Since the first parameter is `a`, we write `4` as the value of `a`. Similarly, we compute the value of `b` and the updated table looks like this:

int main()	
Variable	**Value**
x	4
y	8
larger	

int max(int, int)	
Variable	**Value**
a	4
b	8

4. Once we initialized all the parameters, we continue tracing the program as before, just now we use the newly allocated table.

5. To decide which branch we need to take, we need to evaluate the expression `a > b`. We lookup the values of `a` and `b` in the **latest** table and then evaluate the expression. Since `4 > 8` is false, we take the `else` branch.

6. The statement on the else branch is `return b;`. We evaluate the expression `b` and write the computed value next to the function signature. Then, we cross out the table for the call of the `max` function and return back to evaluating in the `main` function.

   - **Note:** if the function return type is `void` (that is, the function does not return anything), we omit the computation of its return value.
   - **Note:** if we reach end of the non-`void` function body without executing the return statement, that indicates the we found a bug in our program and we have to stop tracing.
     The updated tables look as follows:

int main()	
Variable	**Value**
x	4
y	8
larger	

int max(int, int)	result: 8
Variable	**Value**
a	4
b	8

7. To finish evaluating statement `larger = max(x, y);`, we still need to assign the function result to variable `larger`. We do it as before. Both tables now look like this:

int main()	
Variable	**Value**
x	4
y	8
larger	8

int max(int, int)	result: 8
Variable	**Value**
a	4
b	8

5. The next statement in our `main` function is `std::cout << larger << std::endl;`. We execute it as usual and get the following output:

Program Output
8

6. The final statement of the program is `return 0;`. We execute it in the same way as the previous `return` statement: we write `0` next to the function signature and cross out the table. Since this was the last table, the program terminates. The final state looks as follows:

int main()	
Variable	**Value**
x	4
y	8
larger	8

int max(int, int)	result: 8
Variable	**Value**
a	4
b	8

Note: C++ Tutor does support function calls, so you can use it to check your traces.

Exercises

Task 1

What values are going to be printed by the following program:

```
#include<iostream>
unsigned int f(unsigned int n) {
    if (n == 0) {
        return 0;
    } else {
        if (n == 1) {
            return 1;
        } else {
            return f(n - 2) + f(n - 1);
        }
    }
}

int main() {
    unsigned int x = f(5);
    std::cout << x << std::endl;
    return 0;
}
```

References

Note: References are going to be introduced in the sixth week of the course.

In C++ we can also access variables using references. This is especially useful when we need to access the same variable from several scopes (e.g. two different functions). Let's see why:

Imagine we want to create a function that increments the value of a variable. Consider the following code using the function called `increment`.

```
#include <iostream>

void increment(int x) {
    x = x + 1;
}

int main() {
    int a = 3;
    increment(a);
    std::cout << a << std::endl;
    return 0;
}
```

Before we call the `increment` function, our table looks like this:

int main()	
Variable	**Value**
a	3

Once we enter the `increment` function, we create its table and copy the value of `a` to `x`. This means we now have two variables with the same value of `3`.

int main()	
Variable	**Value**
a	3

void increment()	
Variable	**Value**
x	3

However, the `x = x + 1` statement will only affect the variable `x`.

int main()	
Variable	**Value**
a	3

void increment()	
Variable	**Value**
x	4

This means that the variable `a` will not change when we finish executing the `increment` function, as we do not return the variable `x` and do not assign a new value to variable `a`. Therefore, the statement `std::cout << a << std::endl` will print `3`.

int main()	
Variable	**Value**
a	3

void increment()	
Variable	**Value**
x	1

Obviously, this is not how we want `increment` to work. Instead, we want the `increment` and `main` functions to both use the same variable.

This is where references come into play...

References are nothing else than an alias to another variable. Basically, we are telling the C++ compiler *"I want to continue using this variable, but with a different name and/or in different scope"*. In order to indicate that a variable is actually a reference, we can just add a `&` symbol to the end of its type.

Consider the following code:

```
#include <iostream>

int main() {
    int a = 3;
    int& b = a;
    b = 7;
    std::cout << a << std::endl;
    return 0;
}
```

The statement `int a = 3` has the same effect as in previous examples:

int main()	
Variable	**Value**
a	3

However, `int& b = a` now creates a reference of `a` called `b`. In our table the variable `b` should therefore not have its own value. Instead we just indicate what variable it references.

int main()

Variable	Value
a	3
b	&

We use the “&” symbol to indicate that `b` is a reference. Then we draw an arrow to the the variable that it references. In this case it is `a`. Using the “&” symbol is not mandatory, but it helps us remember that `b` is a reference.

The next statement `b = 7` would usually change the value of `b` in our table. But as `b` is only a reference we now use our table to look up which value we actually need to change. We just need to follow the arrow to the the value of variable `a`. So let’s change the value of `a` to `7`.

int main()

Variable	Value
a	~~3~~ 7
b	&

Thanks to references we were able to change the value of a variable without needing to access it directly. Therefore the statement `std::cout << a << std::endl` will now print `7`.

Now let’s get back to our `increment` function...

Let’s change the signature of the function just slightly: Instead of `void increment(int x)` we write `void increment(int& x)`. So `x` is no longer and independent variable, but a reference instead.

```
#include <iostream>

void increment(int& x) {
    x = x + 1;
}

int main() {
    int a = 3;
    increment(a);
    std::cout << a << std::endl;
    return 0;
}
```

Tracing the statement `int a = 3` gives us this table once again:

int main()	
Variable	**Value**
a	3

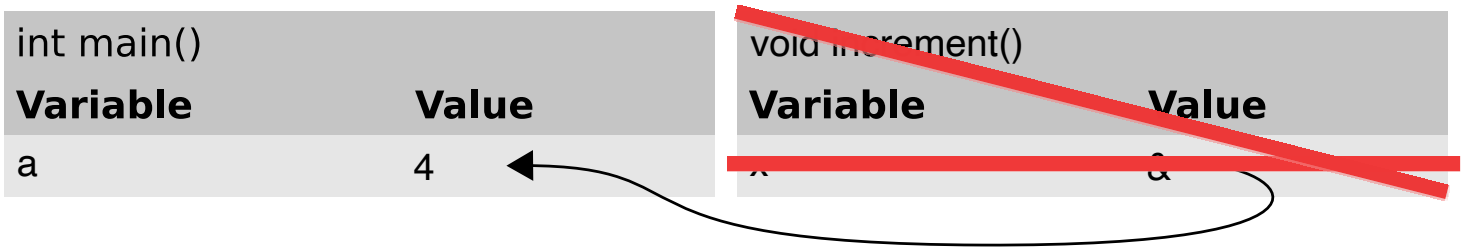
However, once we enter the `increment` function things get interesting. As `x` is now a reference, its value will no longer just get a copy of the value of `a`. Instead it will now reference `a`.

int main()		void increment()	
Variable	**Value**	**Variable**	**Value**
a	3	x	&

The statement `x = x + 1` will therefore effect the value of `a`.

int main()		void increment()	
Variable	**Value**	**Variable**	**Value**
a	4	x	&

By finishing and leaving the `increment` function, we destroy the reference `x`. However, our change to `a` is not affected.



Our `increment` function is therefore working correctly. And the statement `std::cout << a << std::endl` will print `4`.

Exercises

Task 1

What values are going to be printed by the following program:

```
#include <iostream>

int main() {
    int a = 2;
    int b = 5;
    int& c = a;
    b = a + c;
    c = a + b;
    std::cout << b << std::endl;
    a = 7;
    std::cout << c << std::endl;
    return 0;
}
```

Task 2

What value is going to be printed by the following program:

```
#include <iostream>

int something(int& n) {
    if (n == 17) {
        return 5;
    } else {
        return 8;
    }
}

int main() {
    int h = 42;
    int g = 17;
    int& n = g;
    h = g + something(h);
    std::cout << h << std::endl;
    return 0;
}
```

Task 3

What values are going to be printed by the following program:

```
#include <iostream>

int t(int x, int& y) {
    x = y + 4;
    y = x + 2;
    return x + y;
}

int main() {
    int a = 3;
    int b = t(a, a);
    std::cout << a << std::endl;
    std::cout << b << std::endl;
    return 0;
}
```

You can use C++ Tutor (http://pythontutor.com/cpp.html#mode=edit) to check whether your solution is correct. However, please note that C++ tutor uses a slightly different terminology.

Pointers

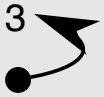
Note: Pointers are going to be introduced in the eleventh week of the course.

A pointer in C++ is a variable that stores the address of a memory location. We can get an address of a memory location by using `&` operator. For example, the following code snippet prints an address of the variable `a`:

```
#include <iostream>
int main() {
    int a = 3;
    int* a_ptr = &a;
    std::cout << a_ptr << std::endl;
    return 0;
}
```

If you run this program, it will print some large hex number such as 0x7fff8957ab1c. When we think about the program, we are typically interested not in the actual numeric value of the address, but what object is at that address. Therefore, when tracing pointers we will use a similar notation to references: an arrow that points to the memory the pointer is currently pointing to. To distinguish between references and pointers, we will use a dot at the beginning of the arrow instead of `&`. For example, the state after executing the statement `int* a_ptr = &a;` in the program above would be:

int main()

Variable	Value
a	3
a_ptr	

Let's trace the following program:

```
#include <iostream>
int main() {
    int a = 3;
    int b = 4;
    int* p1;
    int* p2;
    p1 = &a;
    p2 = p1;
    *p2 = 5;
    p2 = &b;
    std::cout << a << " " << b << std::endl;
    return 0;
}
```

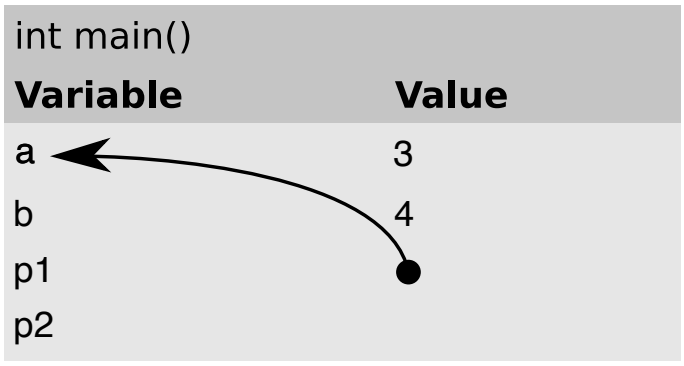
1. The statements `int a = 3;` and `int b = 4;` has the same effect as in the previous examples:

int main()	
Variable	**Value**
a	3
b	4

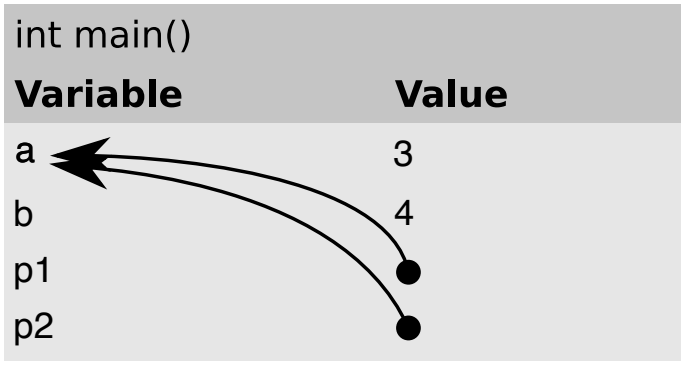
2. The statements `int* p1;` and `int* p2;` declare pointers `p1` and `p2` of type `int*`:

int main()	
Variable	**Value**
a	3
b	4
p1	
p2	

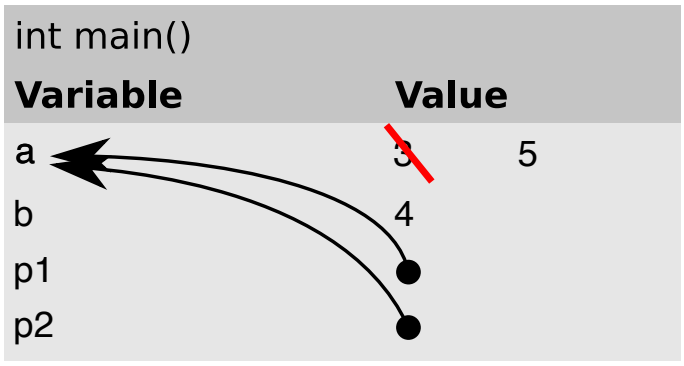
3. `p1 = &a;` stores the address of `a` into `p1`:



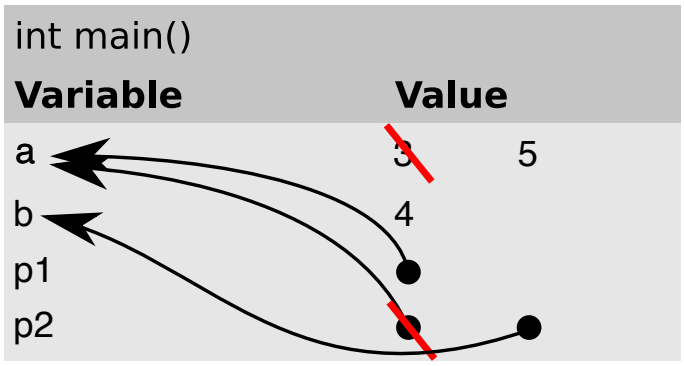
4. `p2 = p1;` assigns the value of `p1` (which is the address of `a`) to `p2`:



5. When we assign directly to the pointer (like in the previous statement), we change the address stored in it. If we want to store a new value to the memory location whose address is stored in the pointer, we need to dereference the pointer with the dereference operator `*`. That is exactly what the statement `*p2 = 5;` does; it stores 5 at the address pointed at by `p2`:



6. Unlike references, pointers can be reassigned. We track that by crossing the old dot and adding a new arrow:



7. The last two statements print the values stored in `a` and `b` to the standard output and return from the function terminating the program.

Exercises

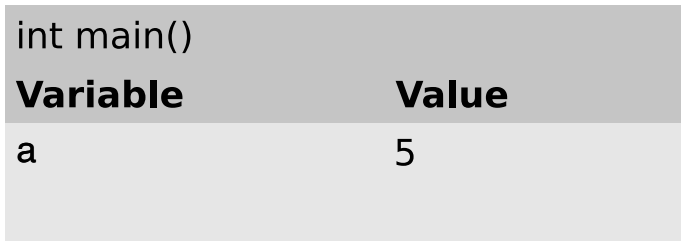
Task 1

Trace the following program:

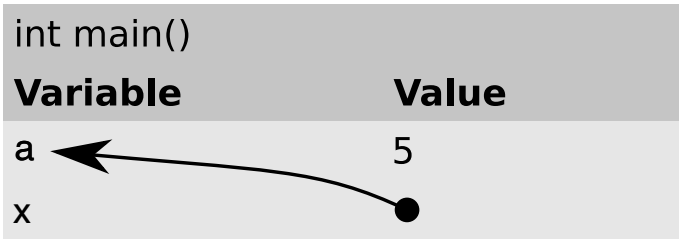
```
int main() {
    int a = 5;
    int* x = &a;
    *x = 6;
    return 0;
}
```

Solution

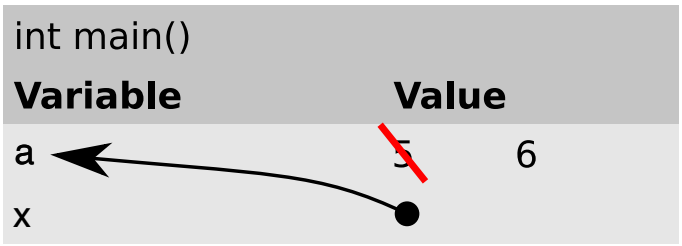
1. After `int a = 5;`:



2. After `int* x = &a;`:



3. After `*x = 6;` :



Dynamic Data Types

Note: Dynamic data types are going to be introduced in the eleventh week of the course.

Pointers can point not only to variables, but also to dynamically allocated memory. Let's trace the following program:

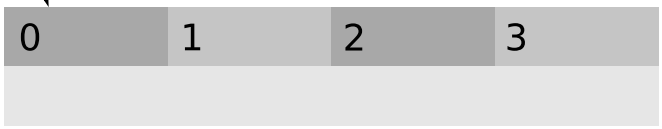
```
#include<iostream>
int main() {
    int* a;
    int* b;
    a = new int[4];
    b = a;
    b++;
    *b = 1;
    b += 2;
    *b = 3;
    delete [] a;
    return 0;
}
```

1. The first two statements `int* a;` and `int* b;` declare two pointers `a` and `b` :

int main()	
Variable	**Value**
a	
b	

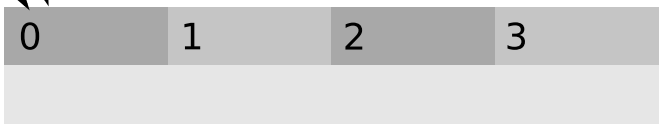
2. The statement `a = new int[4];` allocates a memory block of 4 integers and stores its address in the variable `a`. We represent this by drawing a new table with 4 columns enumerated from 0 to 3 and an arrow from `a` pointing into the first column:

int main()	
Variable	**Value**
a	
b	

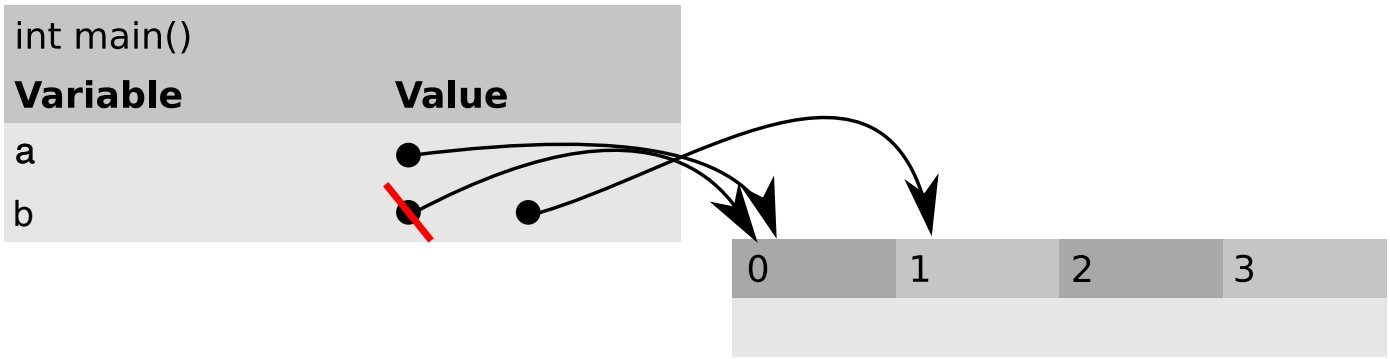


3. The statement `b = a;` assigns the value of `a` to `b`:

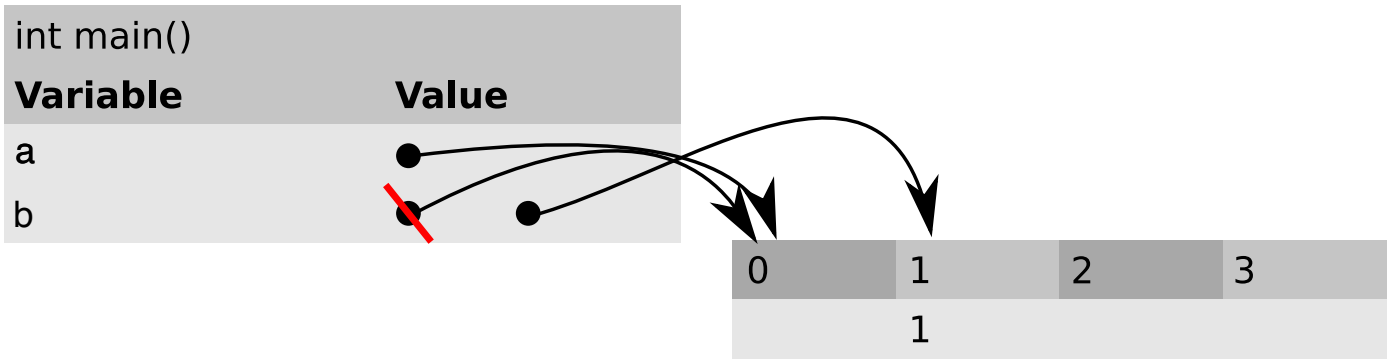
int main()	
Variable	**Value**
a	
b	



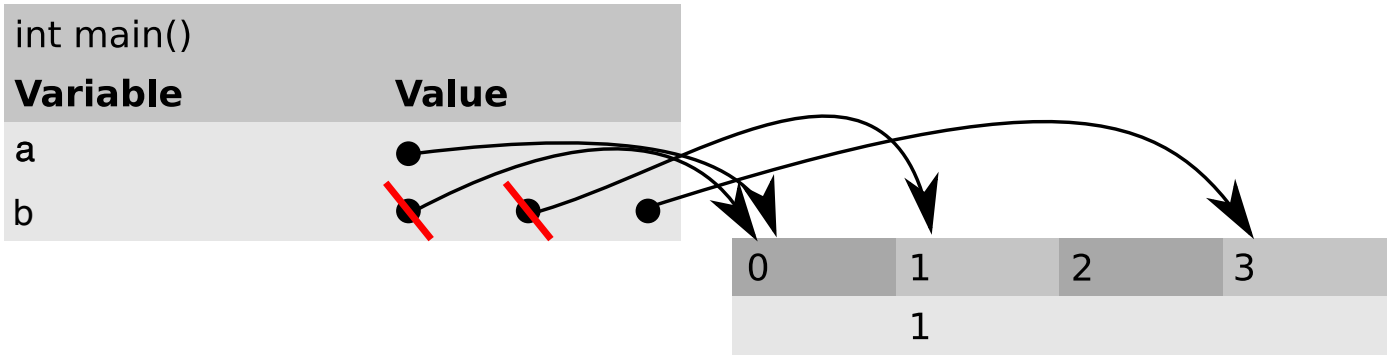
4. The statement `b++;` increments the pointer by 1, which means that now it points to the next element in the memory block:



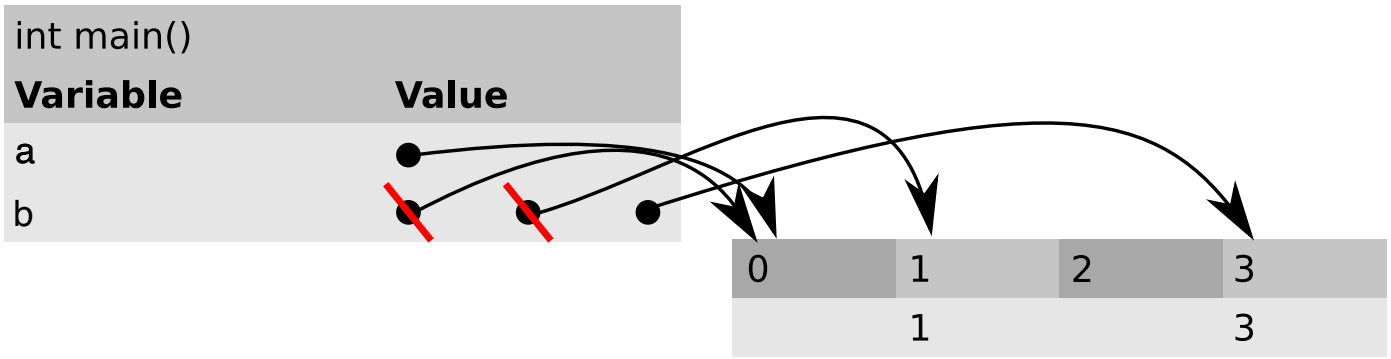
5. The statement `*b = 1;` stores 1 at the memory location whose address is stored in `b`:



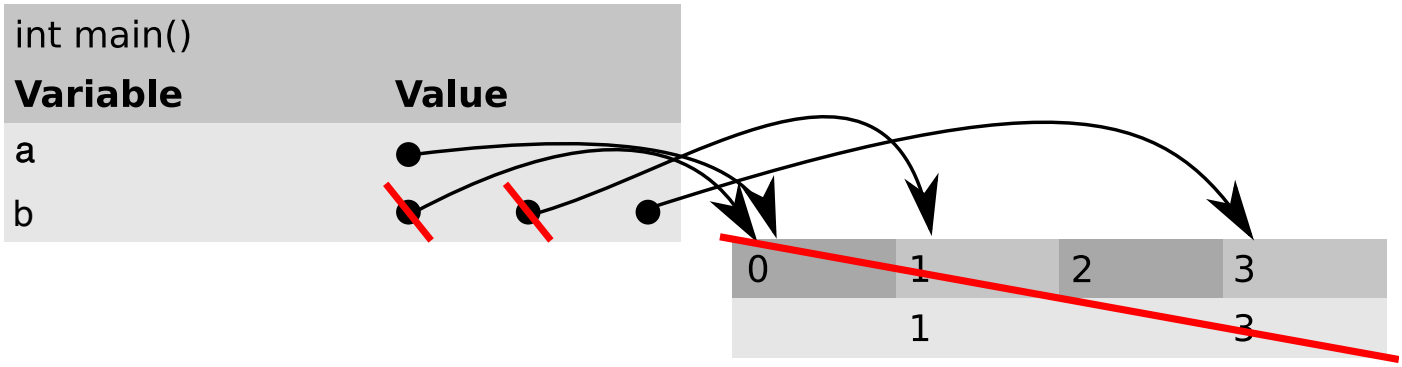
6. The statement `b += 2;` increments the pointer by 2, which means that now it points to the last element in the memory block:



7. The statement `*b = 3;` stores 3 at the memory location whose address is stored in `b`:



8. The statement `delete [] a;` deletes the memory block pointed at by `a`:



Note: the pointer must point to the beginning of an allocated memory block. If you reach a state where the `delete` statement tries to delete an already deleted block or tries to do that via a pointer that does not point to the beginning of the block, then you have reached an error state, which means that the program has a bug. Also, dereferencing a pointer that points to deallocated memory is an error, too.

9. The last statement returns from the function and terminates the program.

Exercises

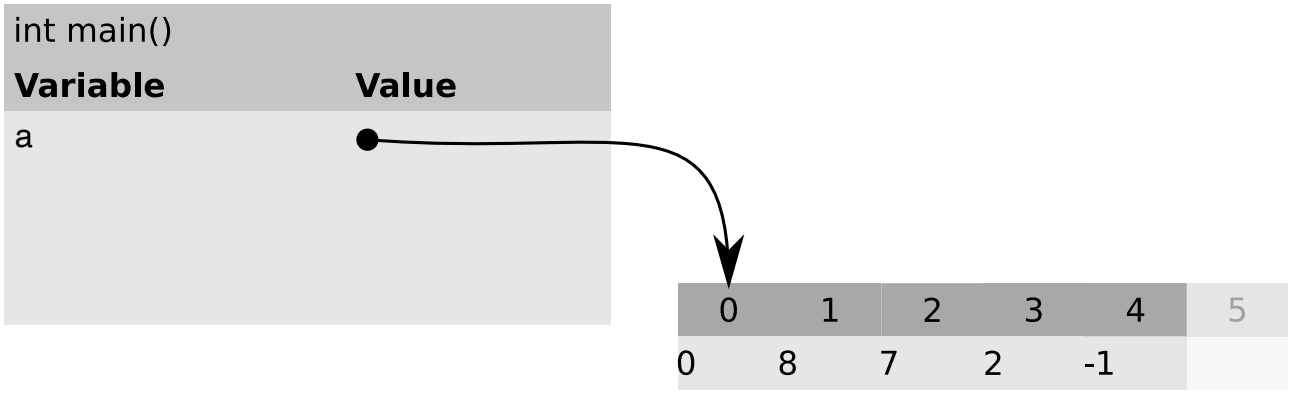
Task 1

Trace the following program:

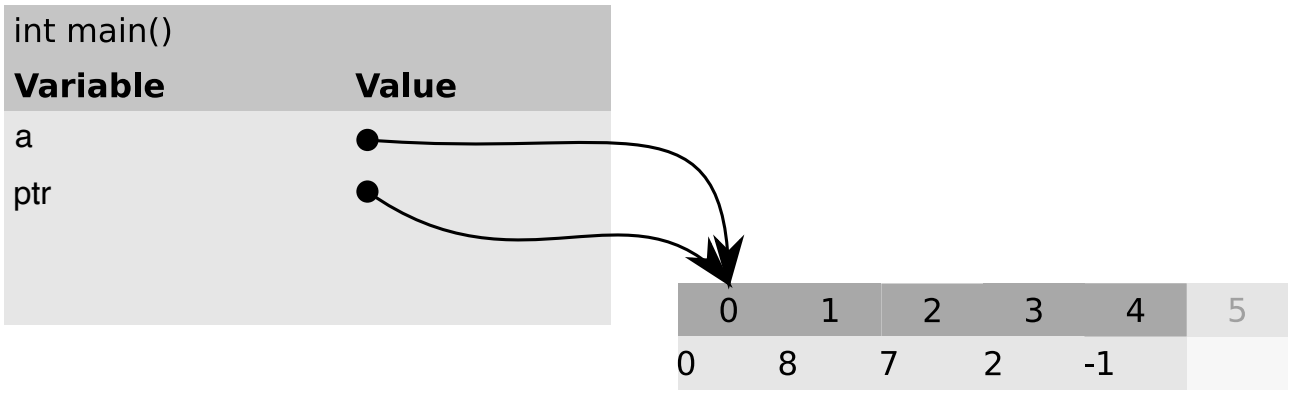
```
#include<iostream>
int main() {
    int* a = new int[5]{0, 8, 7, 2, -1};
    int* ptr = a;                              // pointer assignment
    ++ptr;                                     // shift to the right
    int my_int = *ptr;                         // read target
    ptr += 2;                                  // shift by 2 elements
    *ptr = 18;                                 // overwrite target
    int* past = a + 5;
    std::cout << (ptr < past) << "\n";         // compare pointers
    return 0;
}
```

Solution

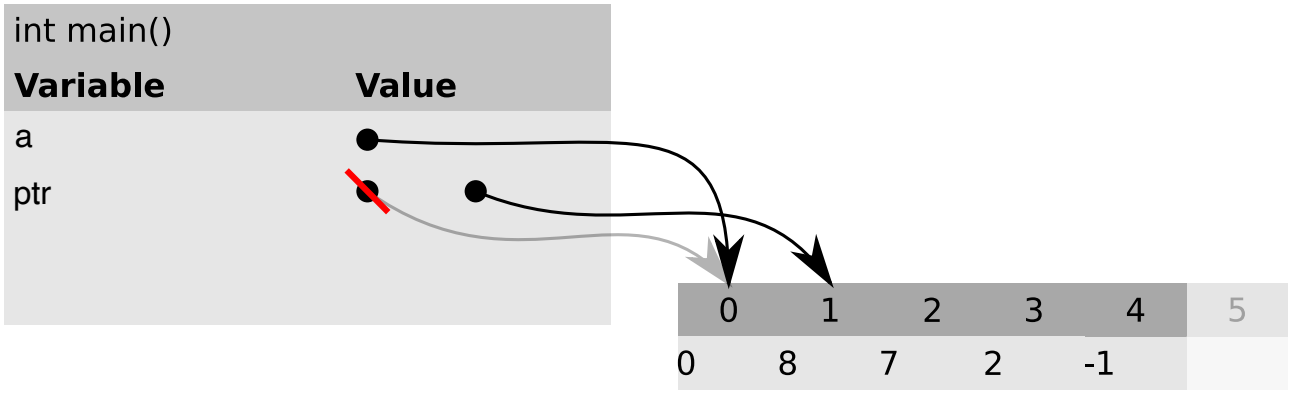
1. After `int* a = new int[5]{0, 8, 7, 2, -1};` :



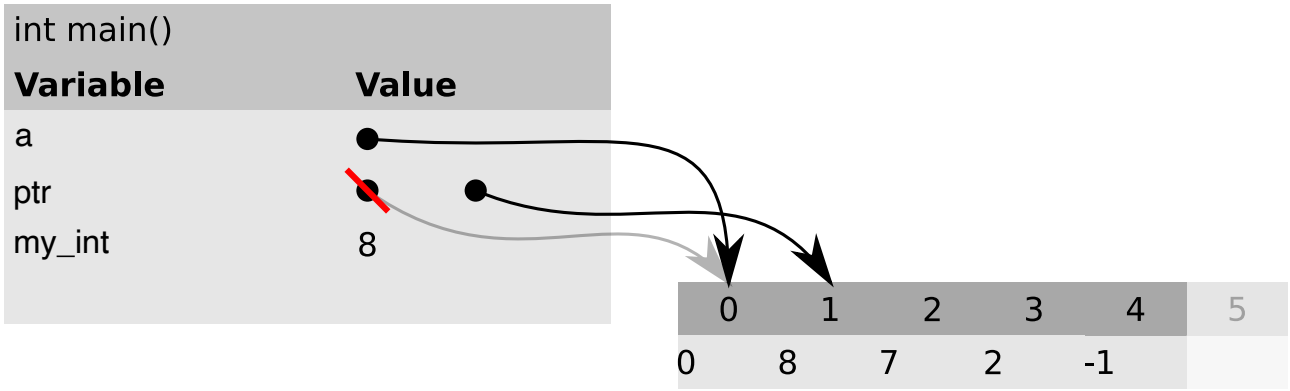
2. After `int* ptr = a;` :



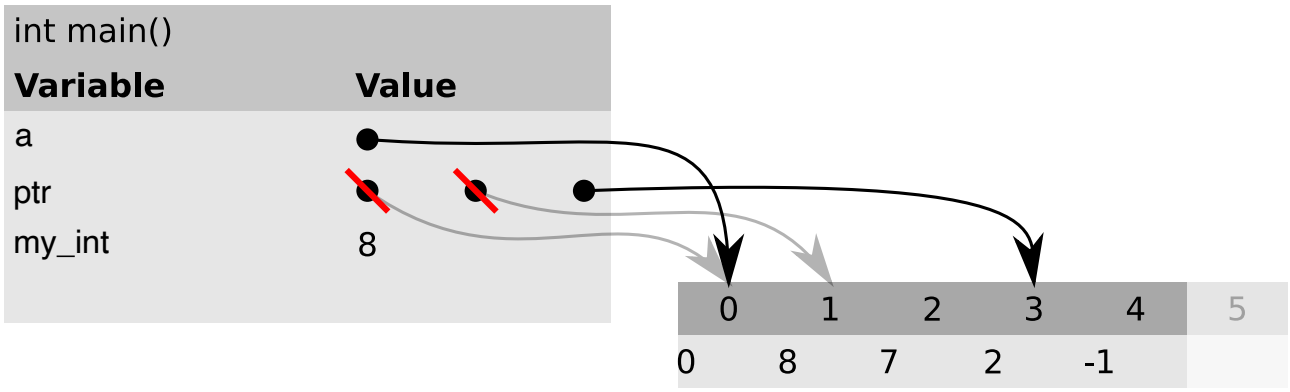
3. After `++ptr;` :



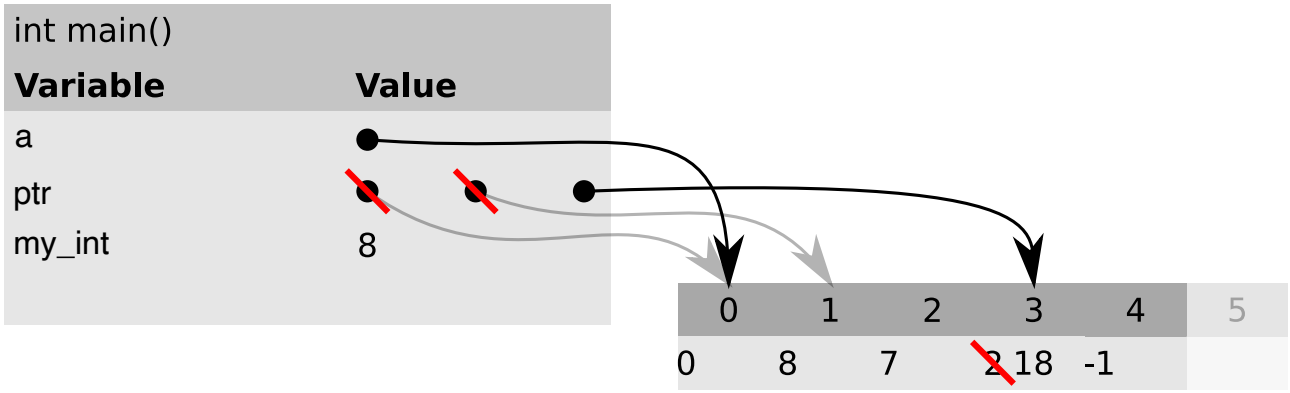
4. After `int my_int = *ptr;` :



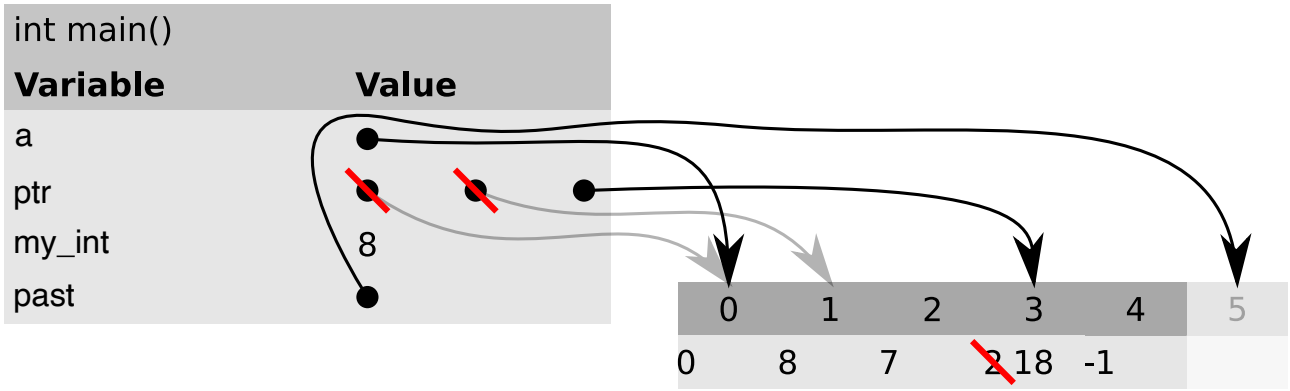
5. After `ptr += 2;` :



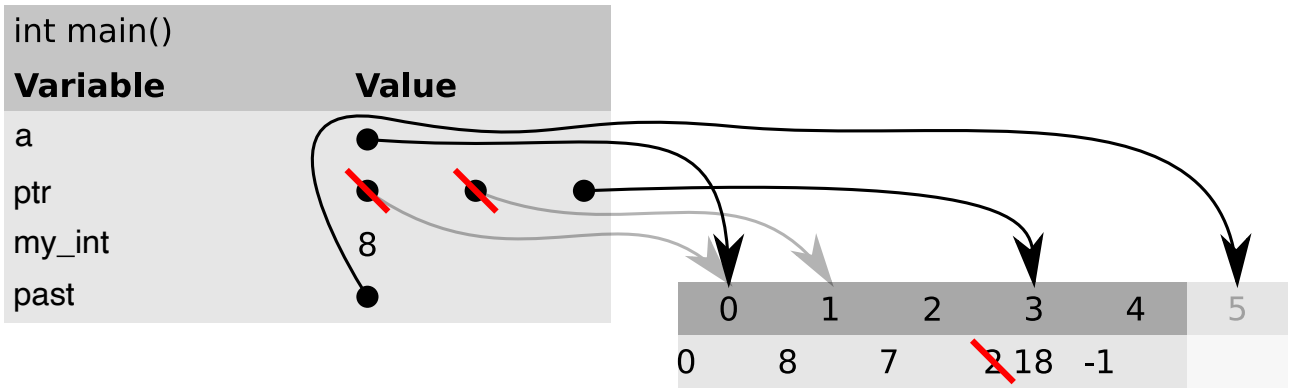
6. After `*ptr = 18;` :



7. After `int* past = a + 5;` :



8. After `std::cout << (ptr < past) << "\n";` :



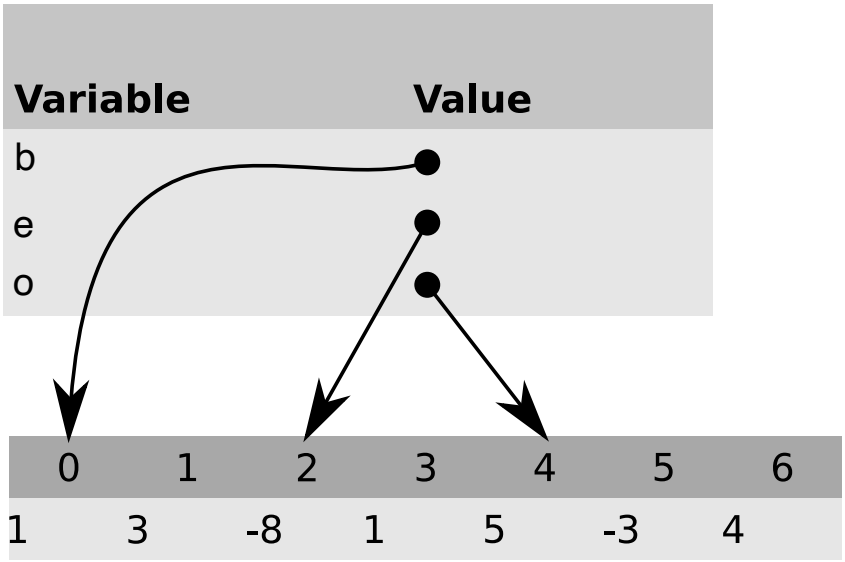
Program Output
1

Task 2

Trace the following function:

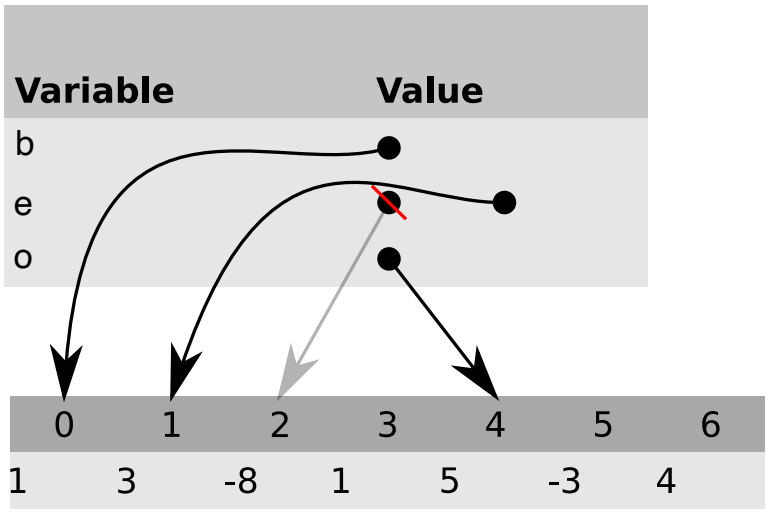
```
// PRE: [b, e) and [o, o+(e-b)) are disjoint valid ranges
void f(int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

assuming its initial state is:

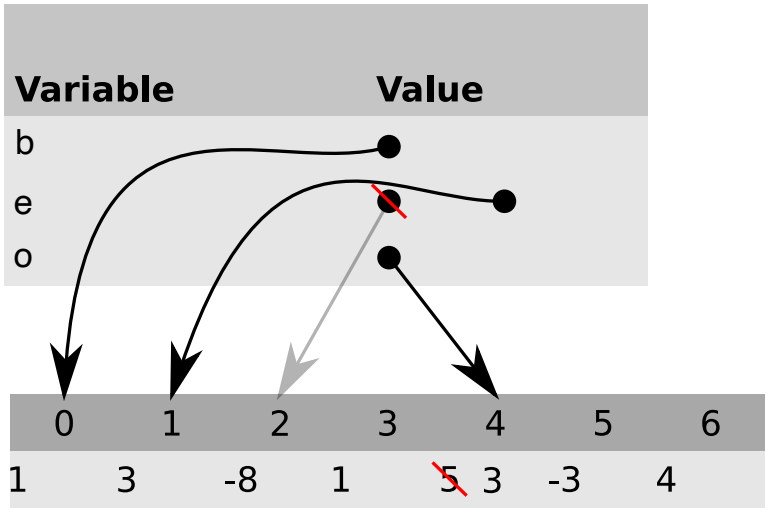


Solution

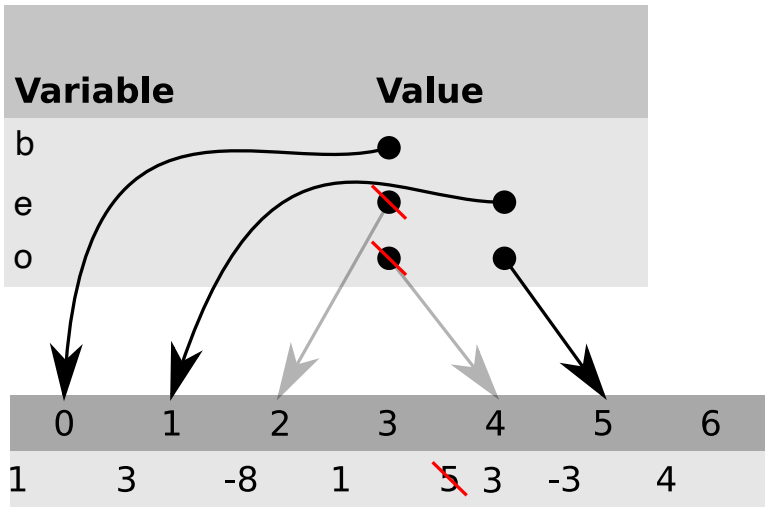
1. The loop condition `b != e` evaluates to `true`. Therefore, we enter the loop body:
    1. After `--e;`:



2. After `*o = *e;` :

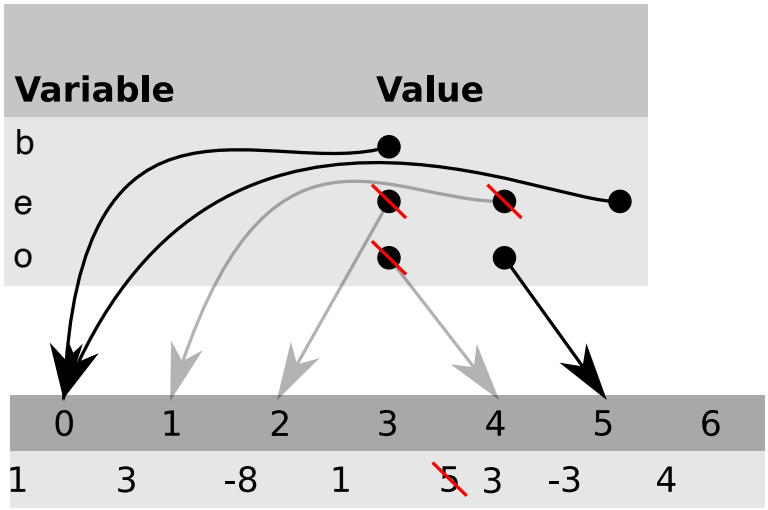


3. After `++o;` :

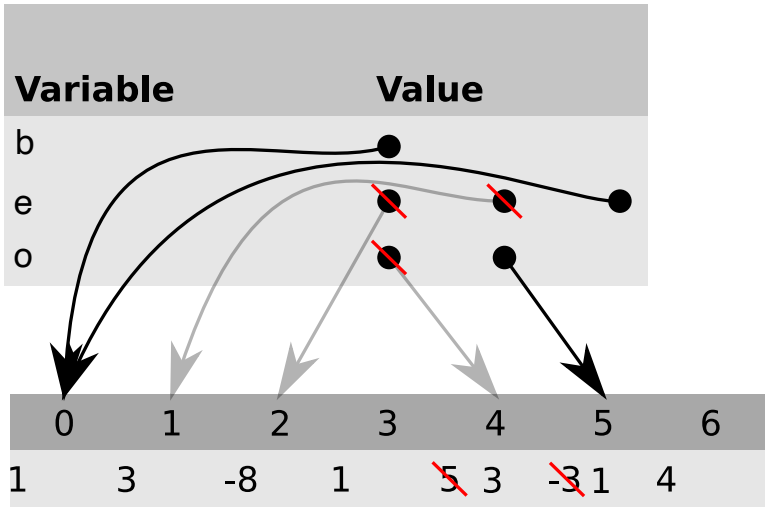


2. The loop condition `b != e` still evaluates to `true`. Therefore, we enter the loop body:

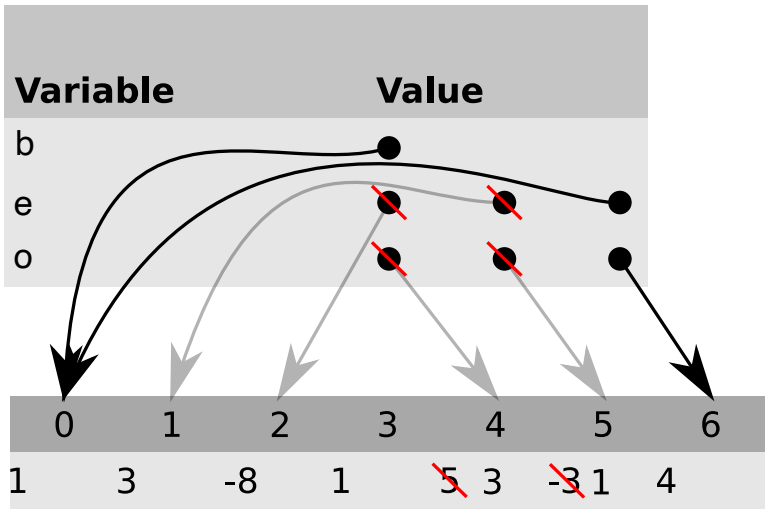
    1. After `--e;`:



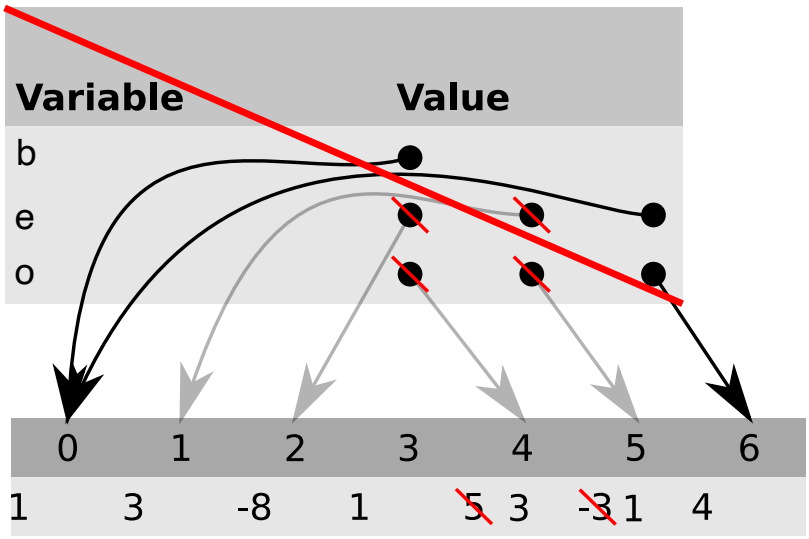
    2. After `*o = *e;`:



    3. After `++o;`:



3. The loop condition `b != e` evaluates to `false`. Therefore, we exit the loop. Since there are no statements after the loop, the function terminates. The final state immediately after exiting the function is:



Destructors

Note: Destructors are going to be introduced in the thirteenth week of the course.

In C++, some data structures (for example, `std::vector`) can be used like primitive values even though internally they are using dynamic memory. This is achieved by implementing a custom constructor, destructor, copy constructor, and assignment operator. By using an

example below, we will show how to trace a program that uses a class with a custom destructor. Other constructs can be traced in a similar way.

```
#include<iostream>
class Box {
  int* ptr;
public:
  Box(int value) {
    this->ptr = new int;
    *(this->ptr) = value;
  }
  ~Box() {
    delete this->ptr;
    this->ptr = nullptr;
  }
  int value() {
    return *(this->ptr);
  }
};
int main() {
    int a;
    {
        Box b(5);
        a = b.value();
    }
    std::cout << a << std::endl;
    return 0;
}
```

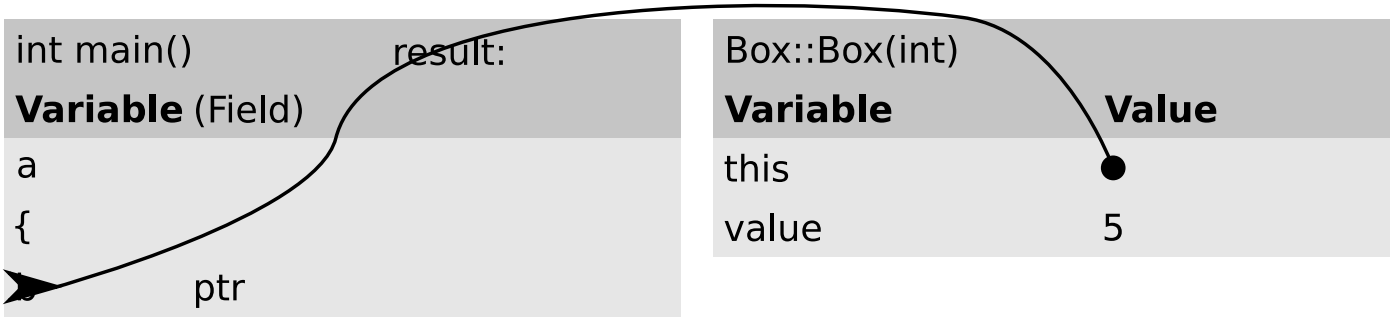
1. The first statement `int a;` declares an integer `a`:

int main()	result:
Variable (Field)	
a	

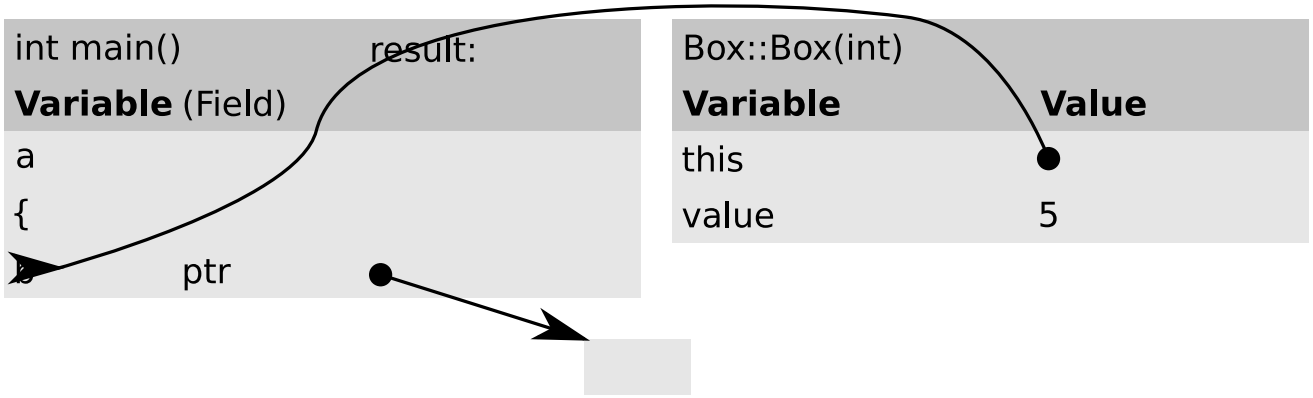
2. Then, the execution enters a new block and a new scope:

int main()	result:
Variable (Field)	
a	
{	

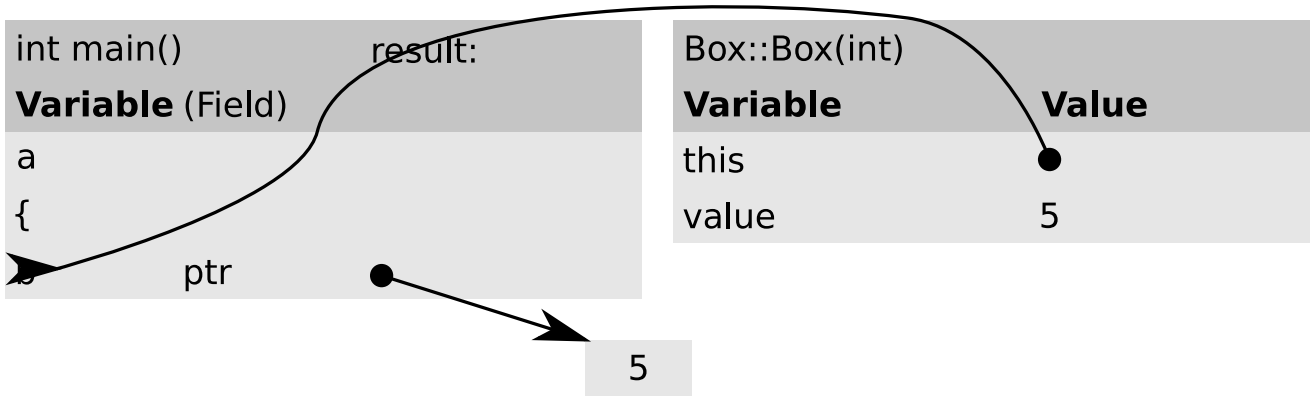
3. The statement `Box b(5);` declares a `Box` b and calls its constructor with an argument 5:



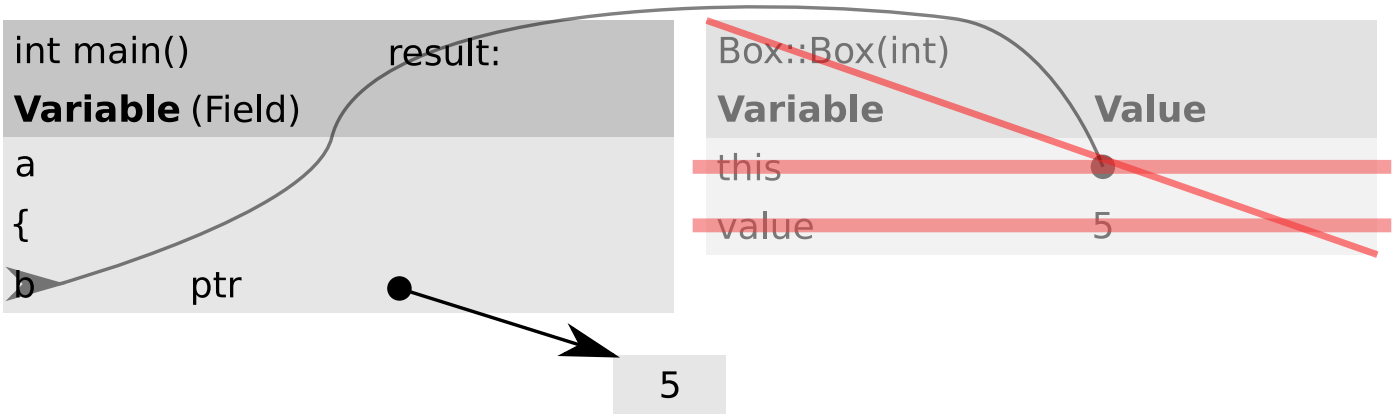
1. The statement `this->ptr = new int;` in the constructor allocates a new integer:



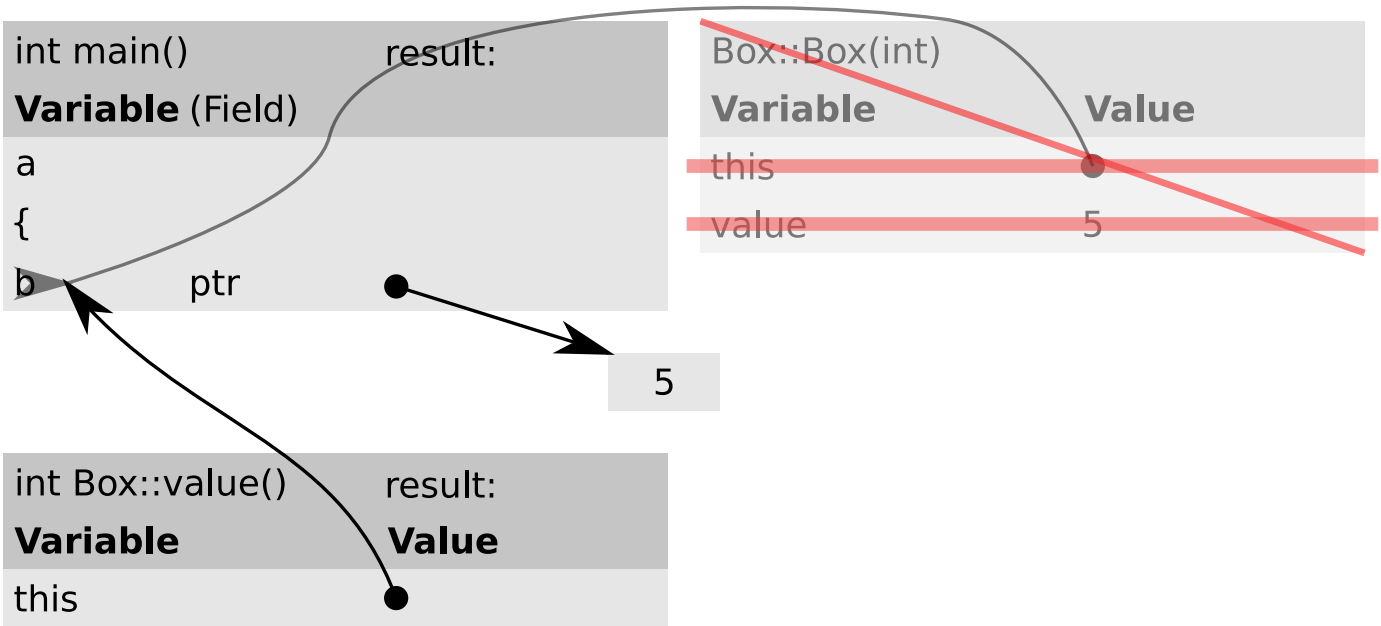
2. The statement `*(this->ptr) = value;` assigns the passed in value to the newly allocated integer:



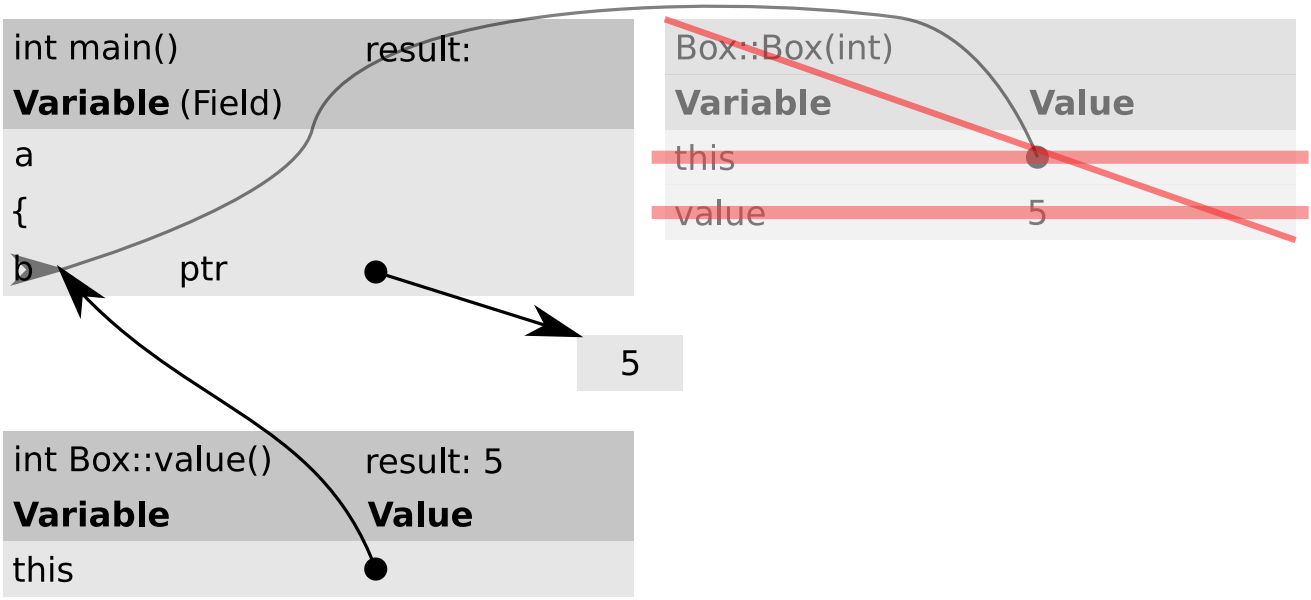
The state after the constructor returns:



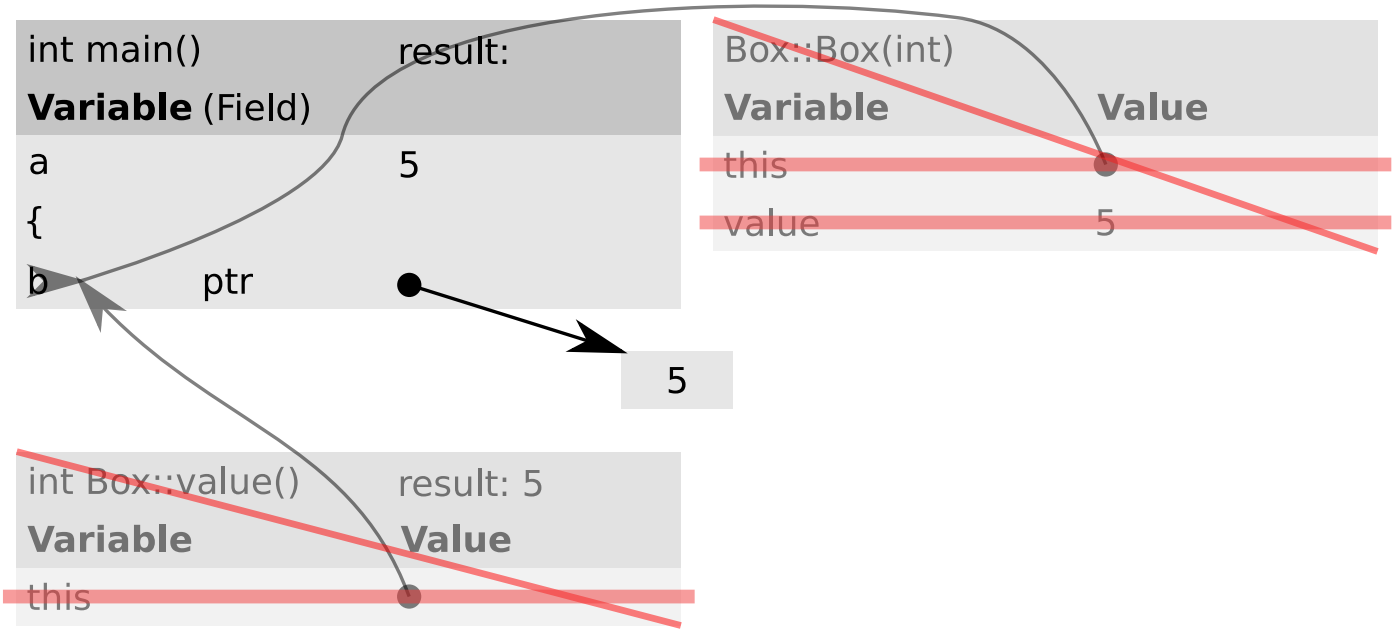
4. The statement `a = b.value();` calls the method `value()`:



1. The only statement `return *(this->ptr);` in the method `value` returns the integer stored at location pointed at by `ptr`:



And assigns the result to `a`:

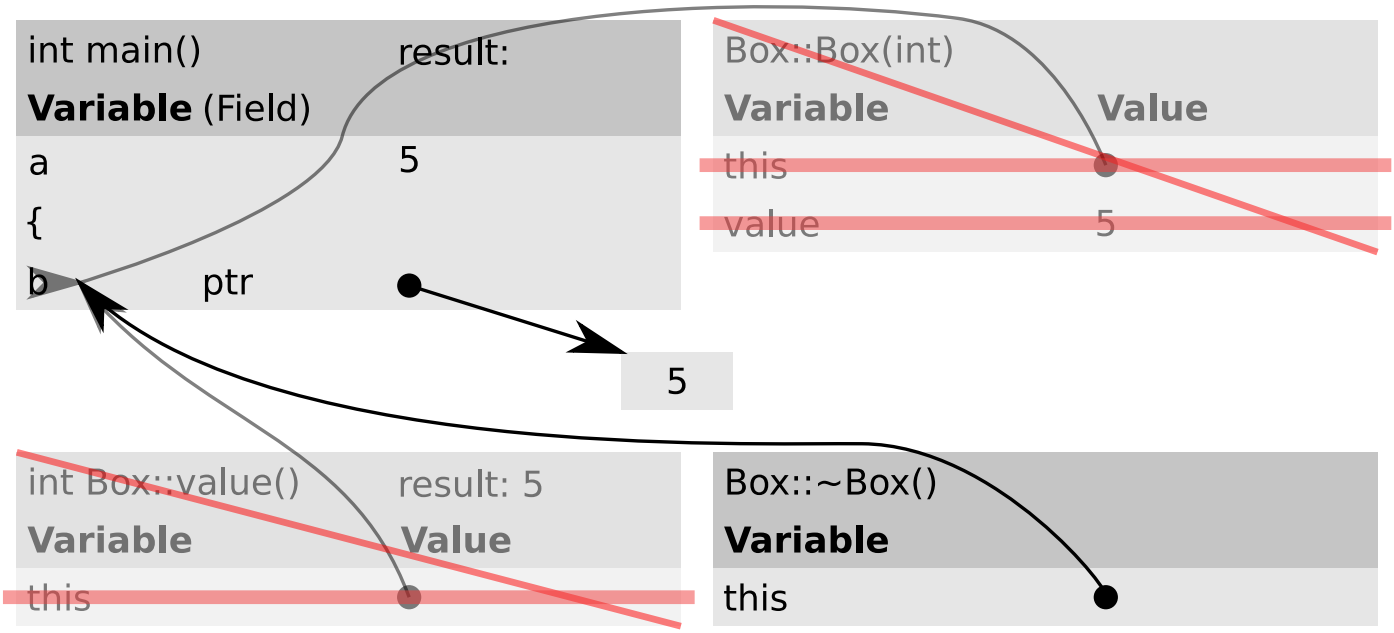


5. The next step exits the block and the corresponding scope:

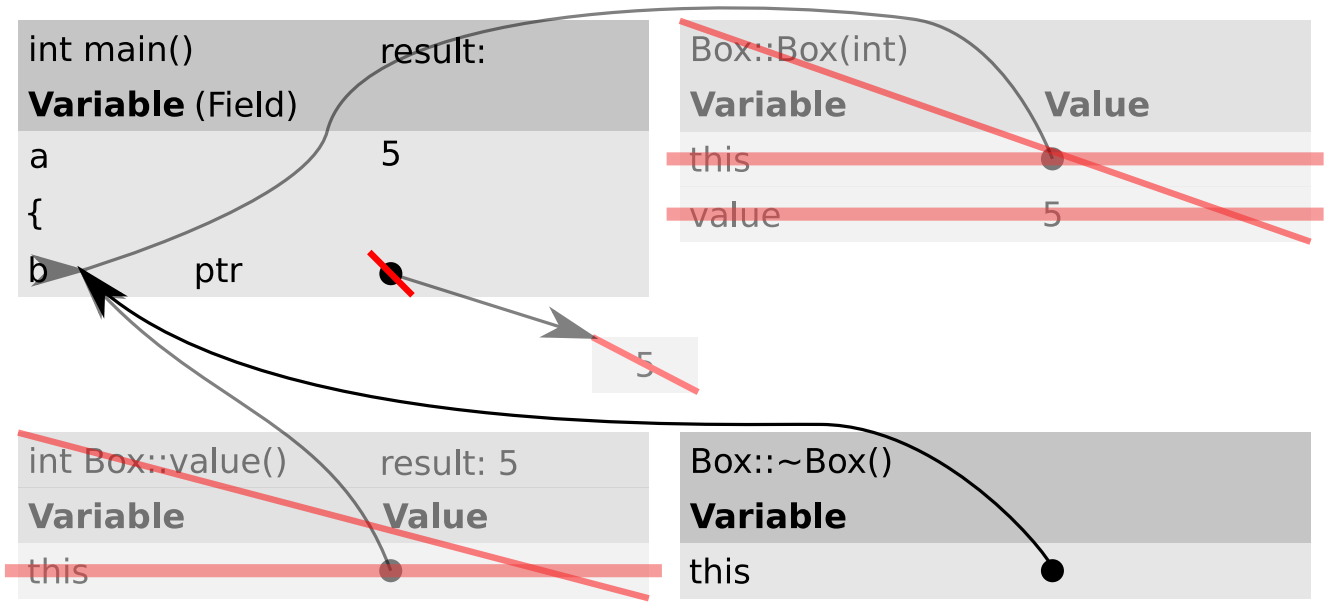
    1. We find the lowest row that contains a not crossed-out `{`.
    2. We cross out that row and all rows that are below it starting from the lowest one.
    3. When we cross out a variable, we need to check if its type implements a

destructor. If yes, we execute it.

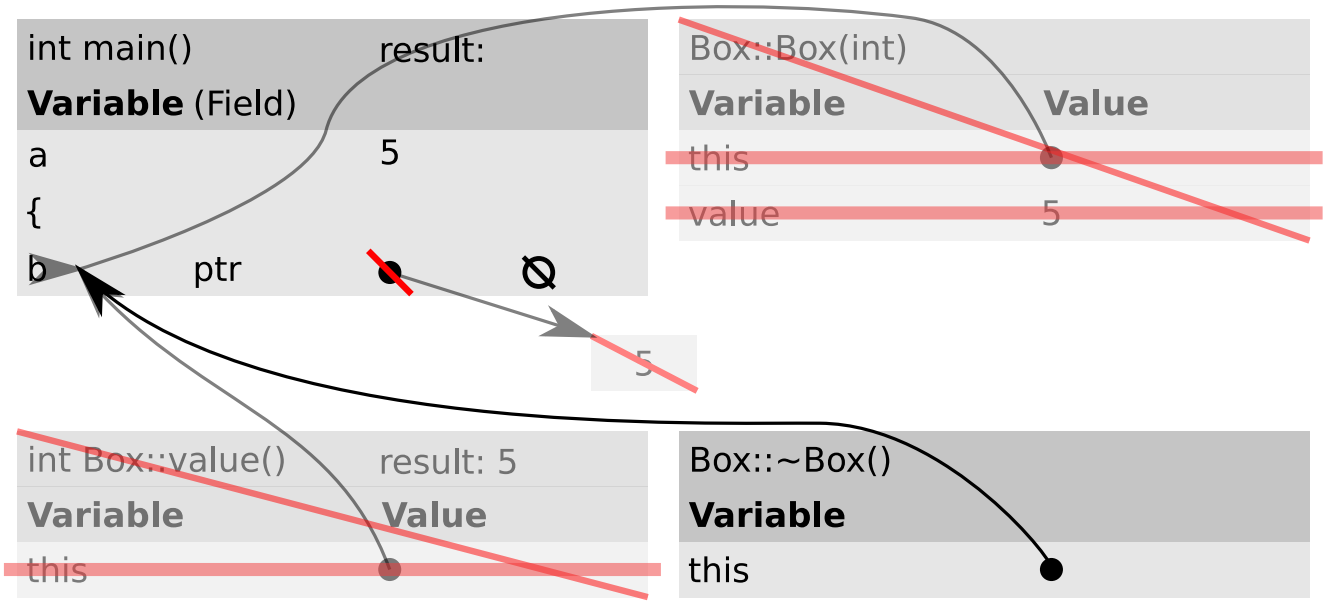
In this case, `b` has a destructor which we need to execute:



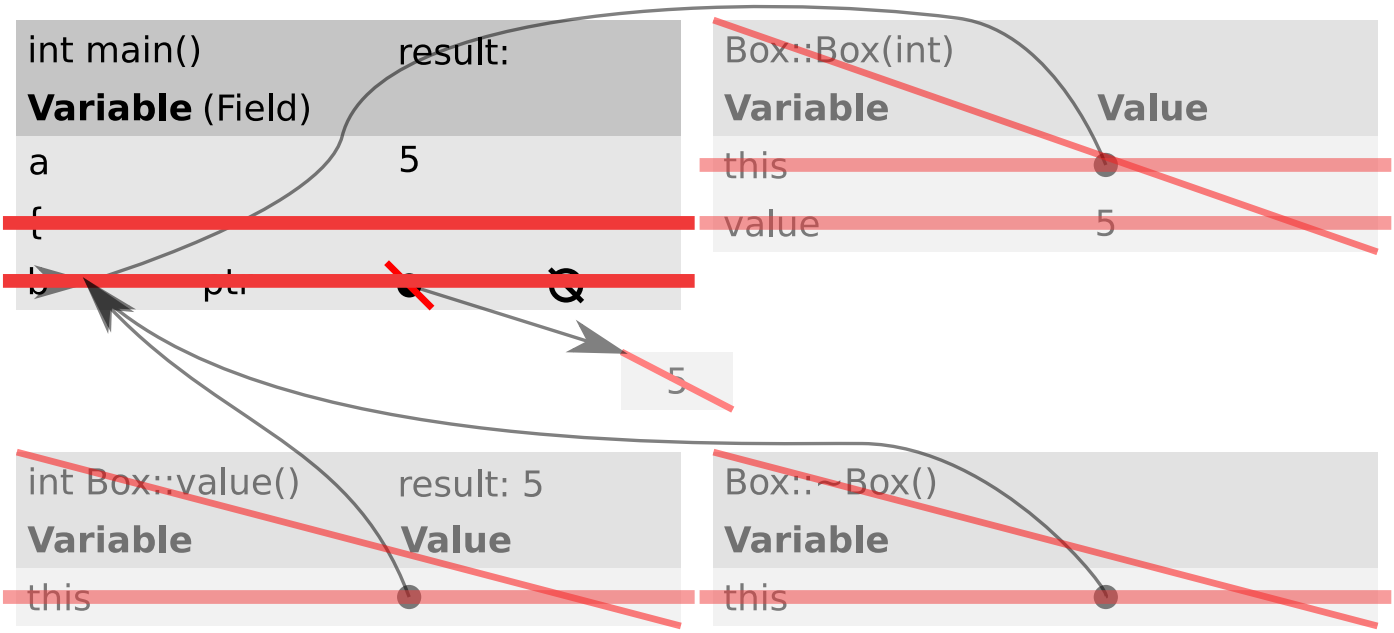
1. The statement `delete this->ptr;` in the destructor deallocates the integer pointed at by `this->ptr` and, as a result, `this->ptr` becomes dangling:



2. The statement `this->ptr = nullptr;` sets the pointer to `nullptr` (indicated by the empty set symbol):



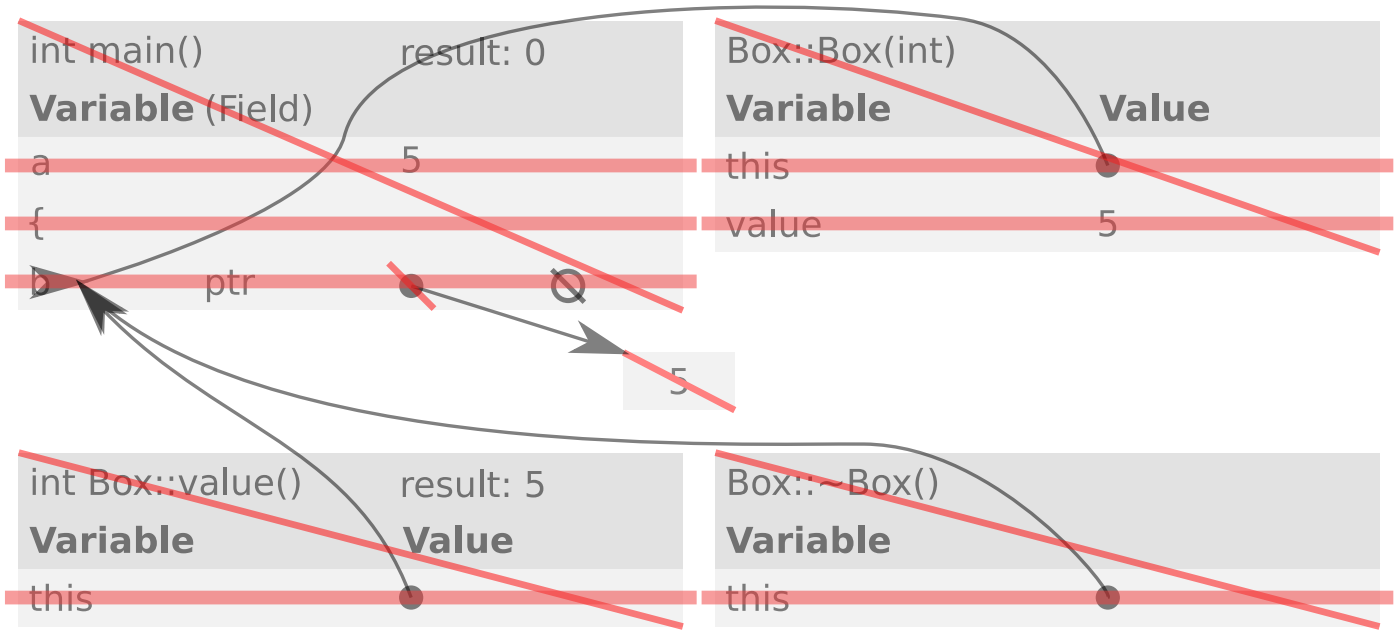
When the `b` destructor finishes, the state looks as follows:



6. The statement `std::cout << a << std::endl;` prints the value of `a` to the standard output:

Program Output
5

7. The last statement returns from the `main` function and terminates the program:


int main()
result: 0
Variable (Field)
a
5
{
b
ptr
∅
5
Box::Box(int)
Variable
Value
this
value
5
int Box::value()
result: 5
Variable
Value
this
Box::~Box()
Variable
this