CS4414 Recitation 3

C++ memory management

09/2024

Alicia Yang

Overview

- C++ memory
  - Stack vs heap
  - Scope, lifetime, ownership
- C++ pointers

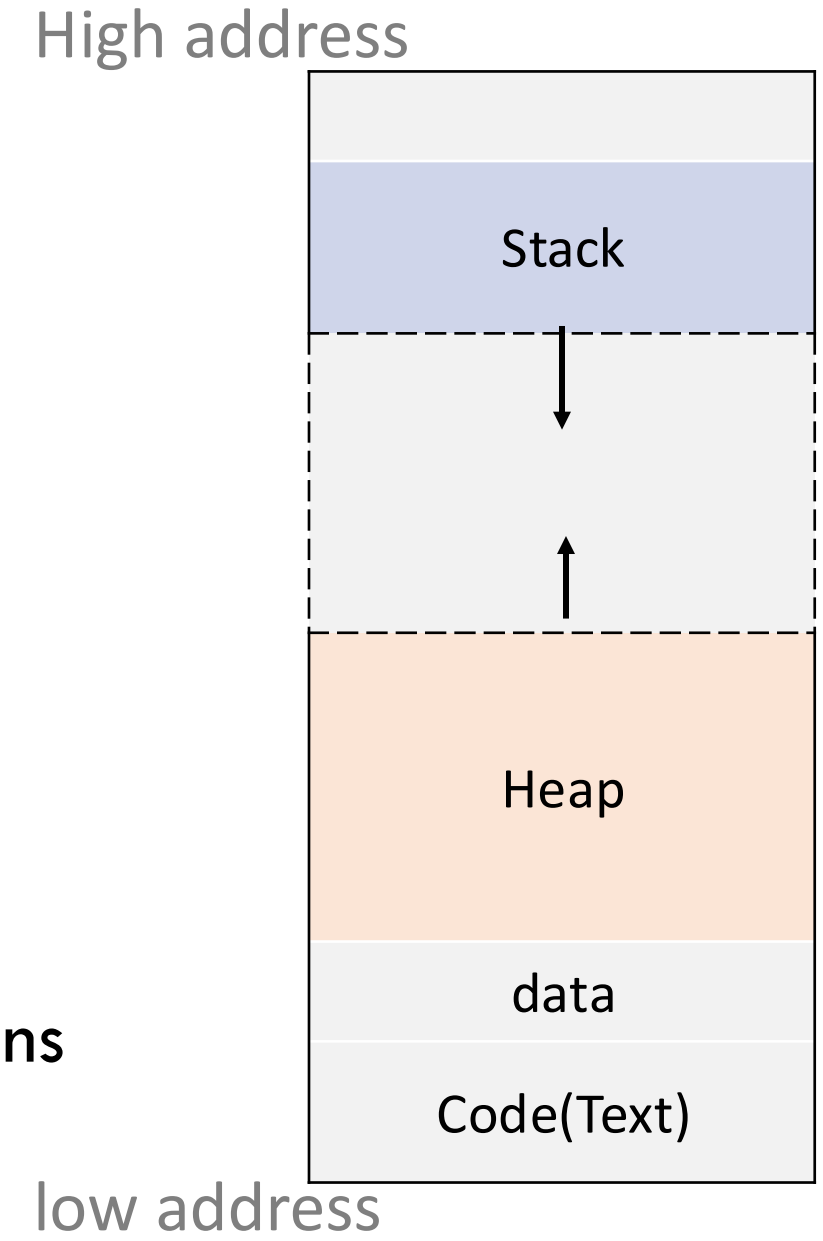
C++ Memory

- How does stack and heap memory work?
- How to use stack and heap memory in my program?

Stack memory

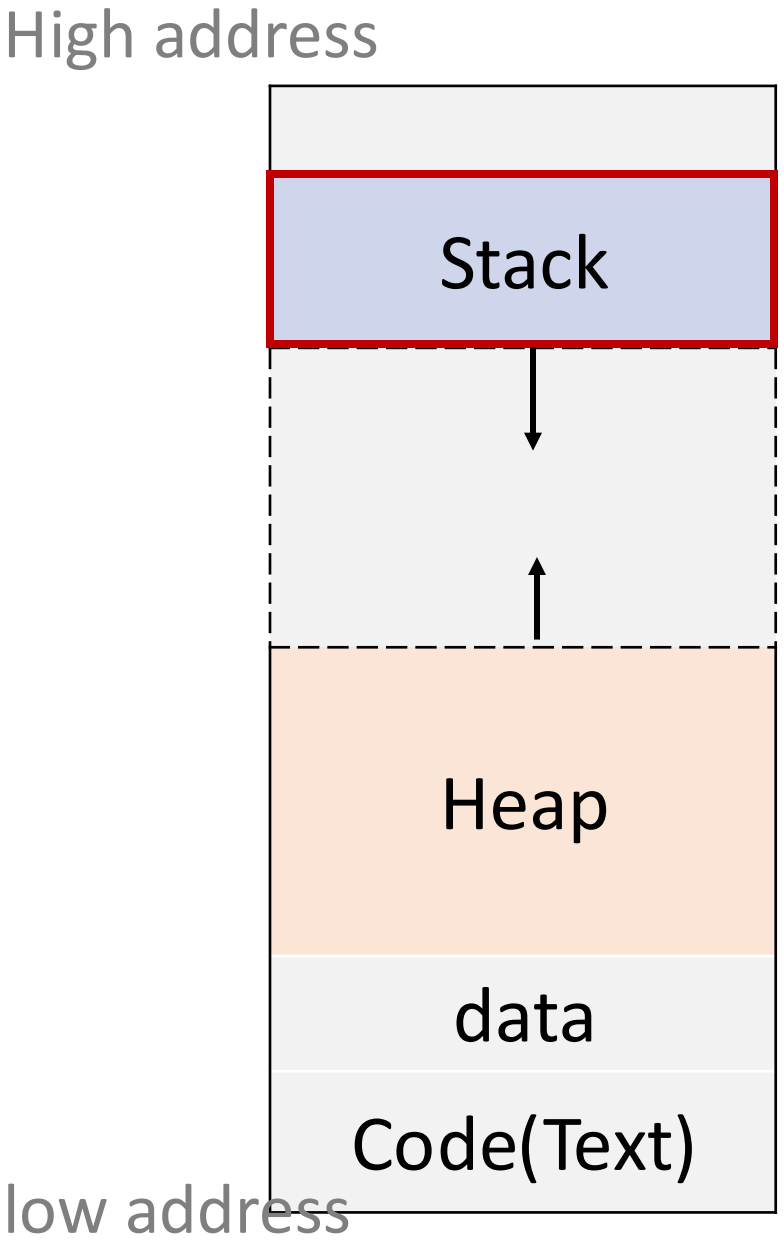
Memory

- **Stack**: used for memory needed to call methods(such as local variables), or for inline variables
- **Heap**: Dynamically memory used for programmers to allocate. The memory will often be used for longer period than stack
- **Data**: use for constants and initialized global objects
- **Code**: segments that holds compiled instructions



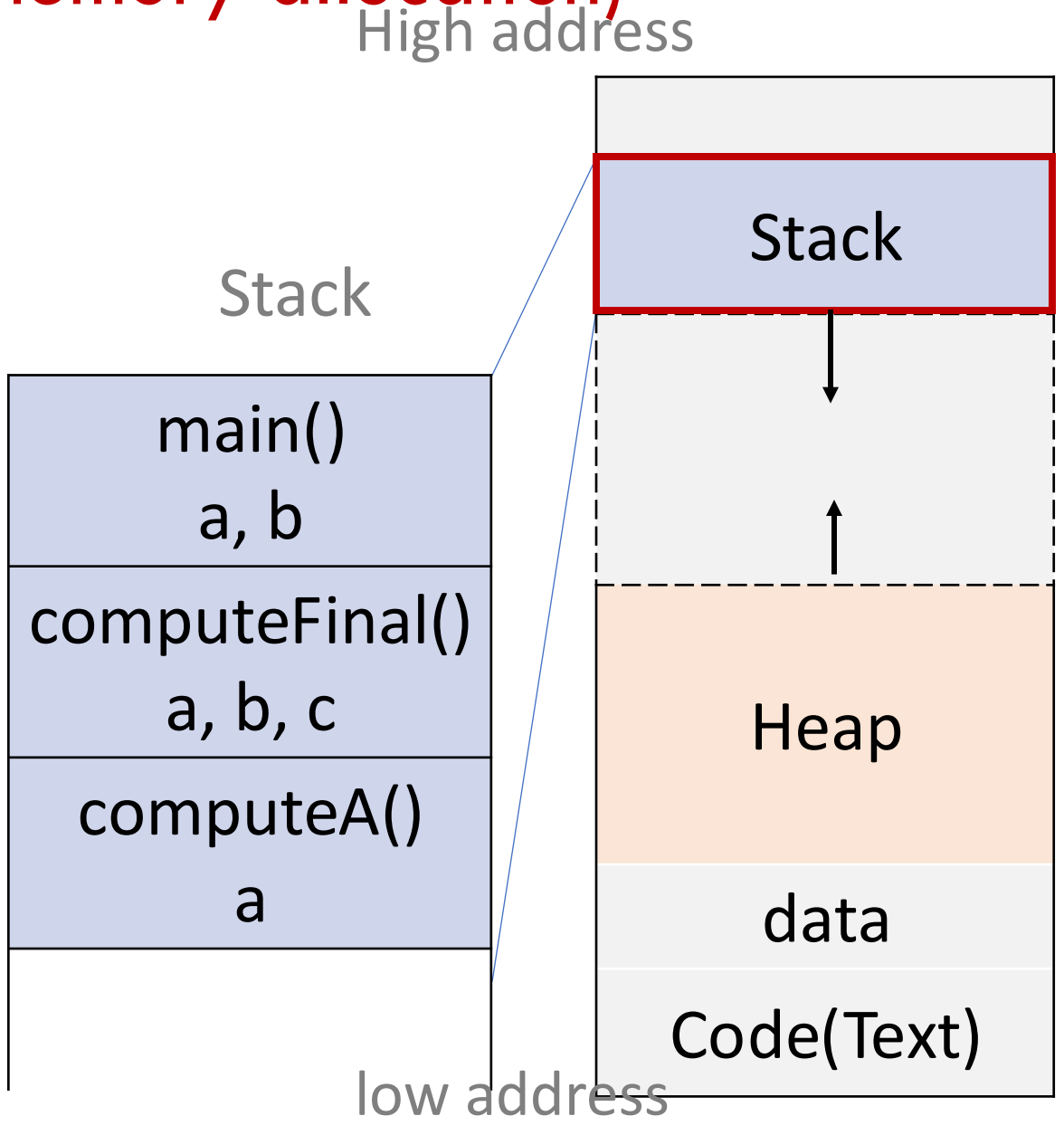
Stack Memory

- Stack Allocation (Temporary memory allocation):
  - Allocate on contiguous blocks of memory, in a fixed size
  - Allocation happens in function call stack



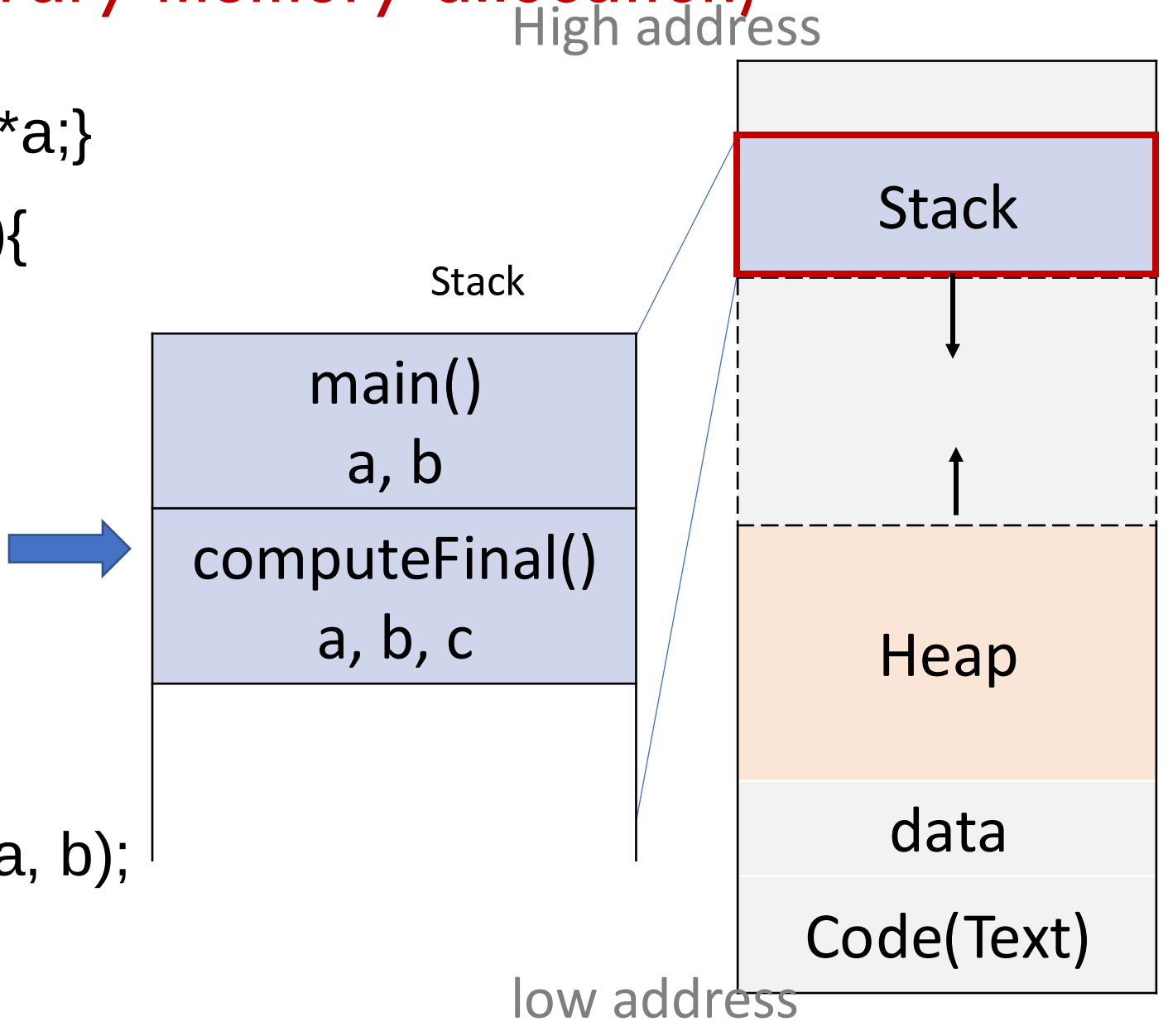
Stack Allocation (Temporary memory allocation)

```
int computeA(int a){return a*a;}

int computeFinal(int a, int b){
    int c = computeA(a) + b;
    return c;
}

int main()
{
    int a = 1;
    int b = 2;
    int total = computeFinal(a, b);
    …
}
```



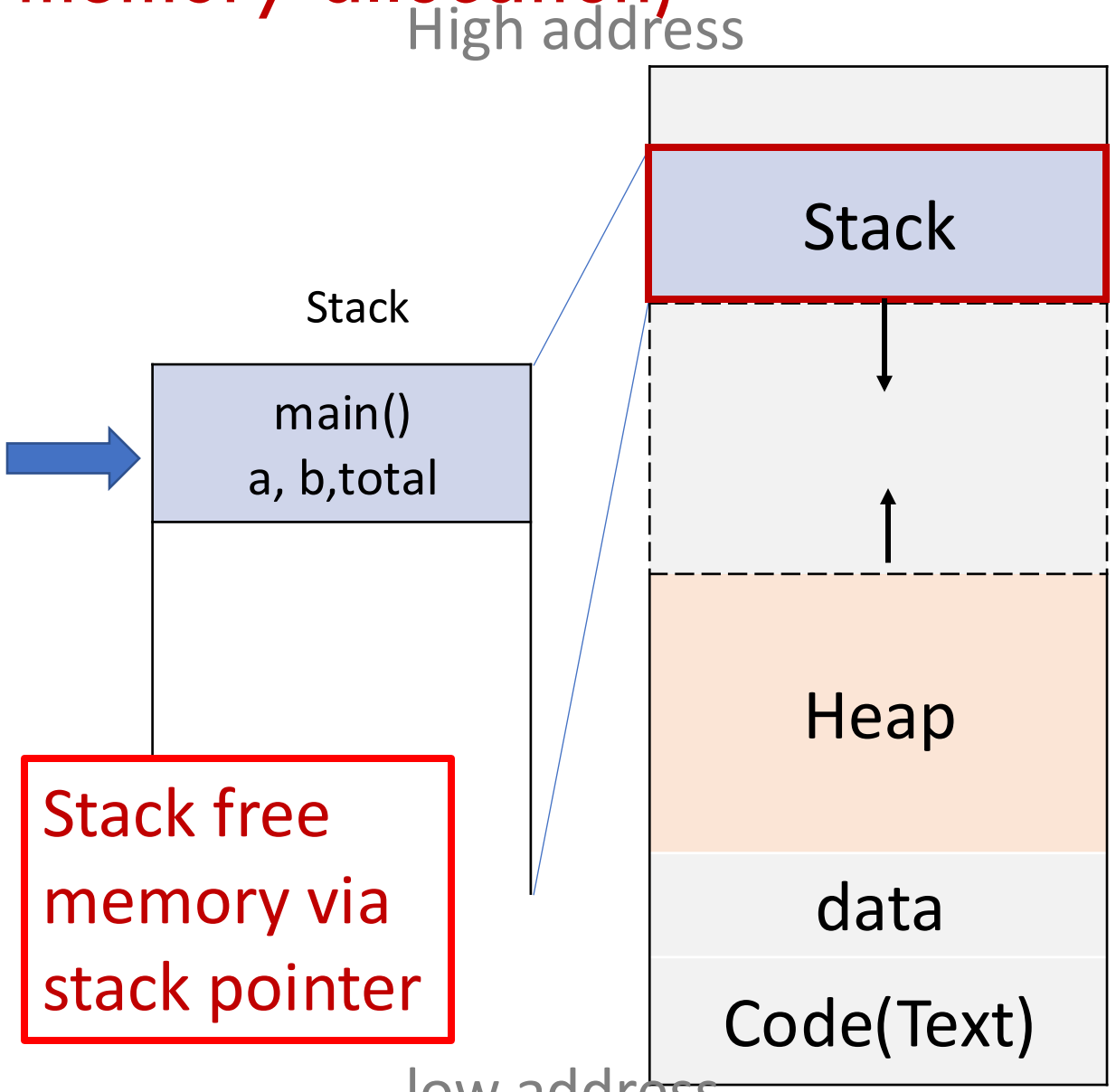
Stack Allocation (Temporary memory allocation)

```
int computeA(int a){return a*a;}

int computeFinal(int a, int b){
    int c = computeA(a) + b;
    return c;
}

int main()
{
    int a = 1;
    int b = 2;
    int total = computeFinal(a, b);
    …
}
```



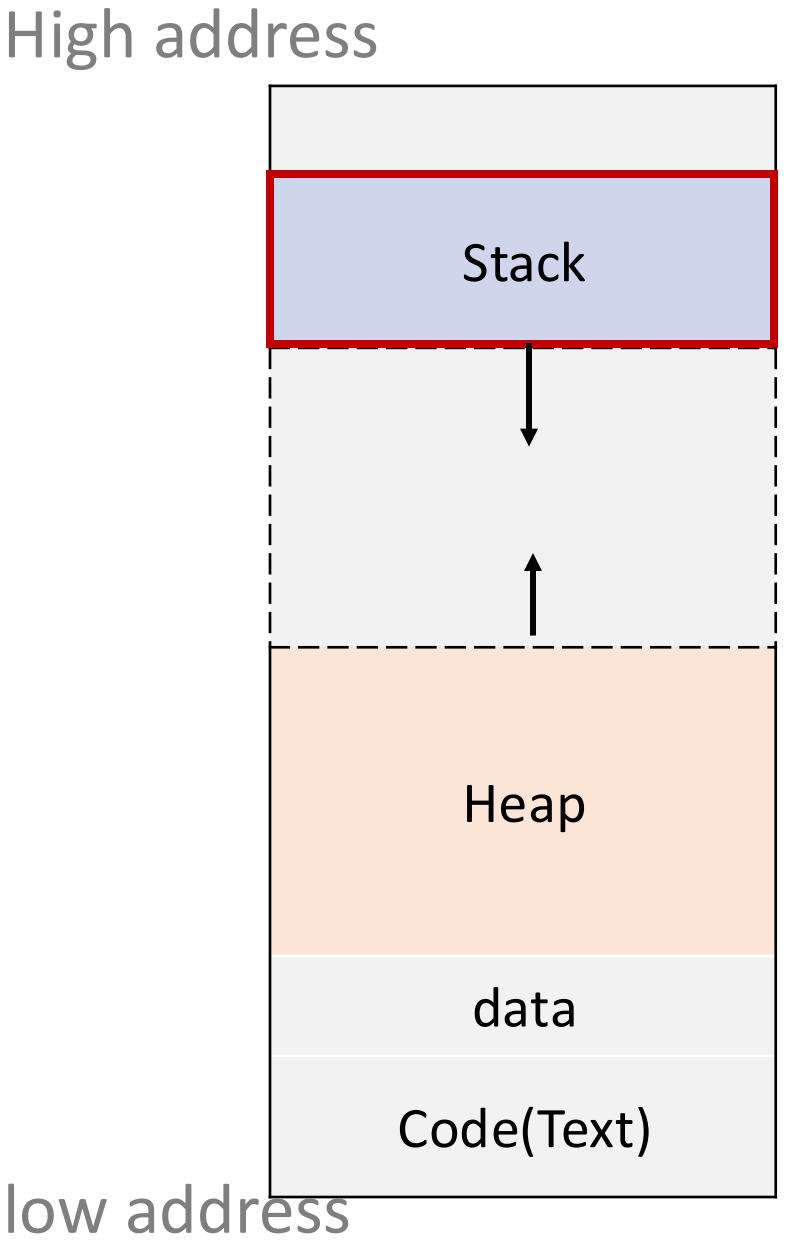
Stack Allocation (Temporary memory allocation)

```
int computeA(int a){return a*a;}

int computeFinal(int a, int b){
    int c = computeA(a) + b;
    return c;
}

int main()
{
    int a = 1;
    int b = 2;
    int total = computeFinal(a, b);
    …
}
```



Stack Memory

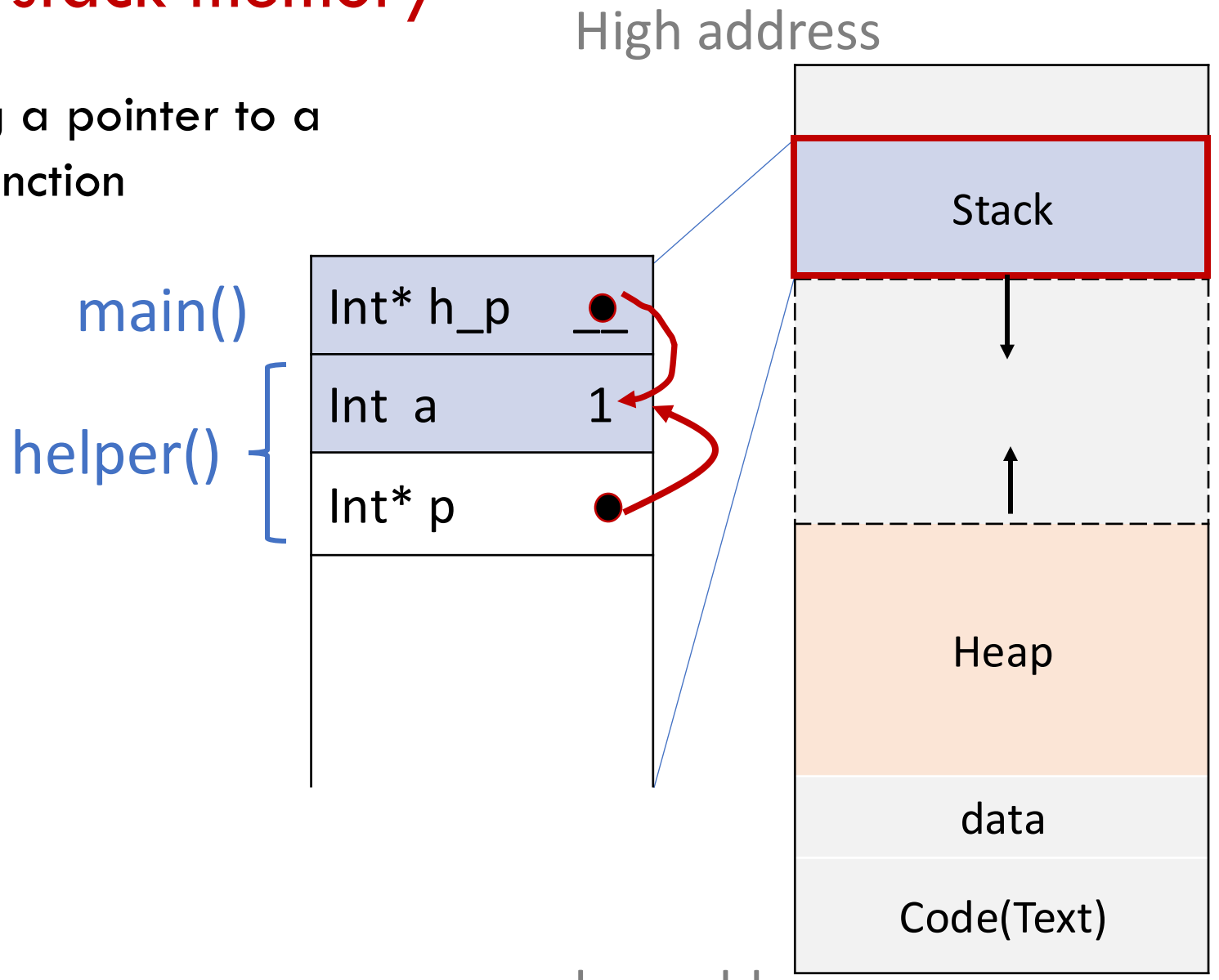
- Stack Allocation (Temporary memory allocation):
  - Allocate on contiguous blocks of memory, in a fixed size
  - Allocation happens in function call stack
  - When a function called, its variables got allocated on stack; when the function call is over, the memory for the variables is deallocated. (scope)
  - The allocation and deallocation for stack memory is **automatically done**.
  - Fast to allocate memory on stack(1CPU operation), faster than heap



Common mistake with stack memory

- A common mistake: returning a pointer to a stack variable in a helper function

```
int* helper()
{
    int a = 3;
    int * p = &a;
    return p;
}
int main(){
    int* h_p = helper();
    …
}
```

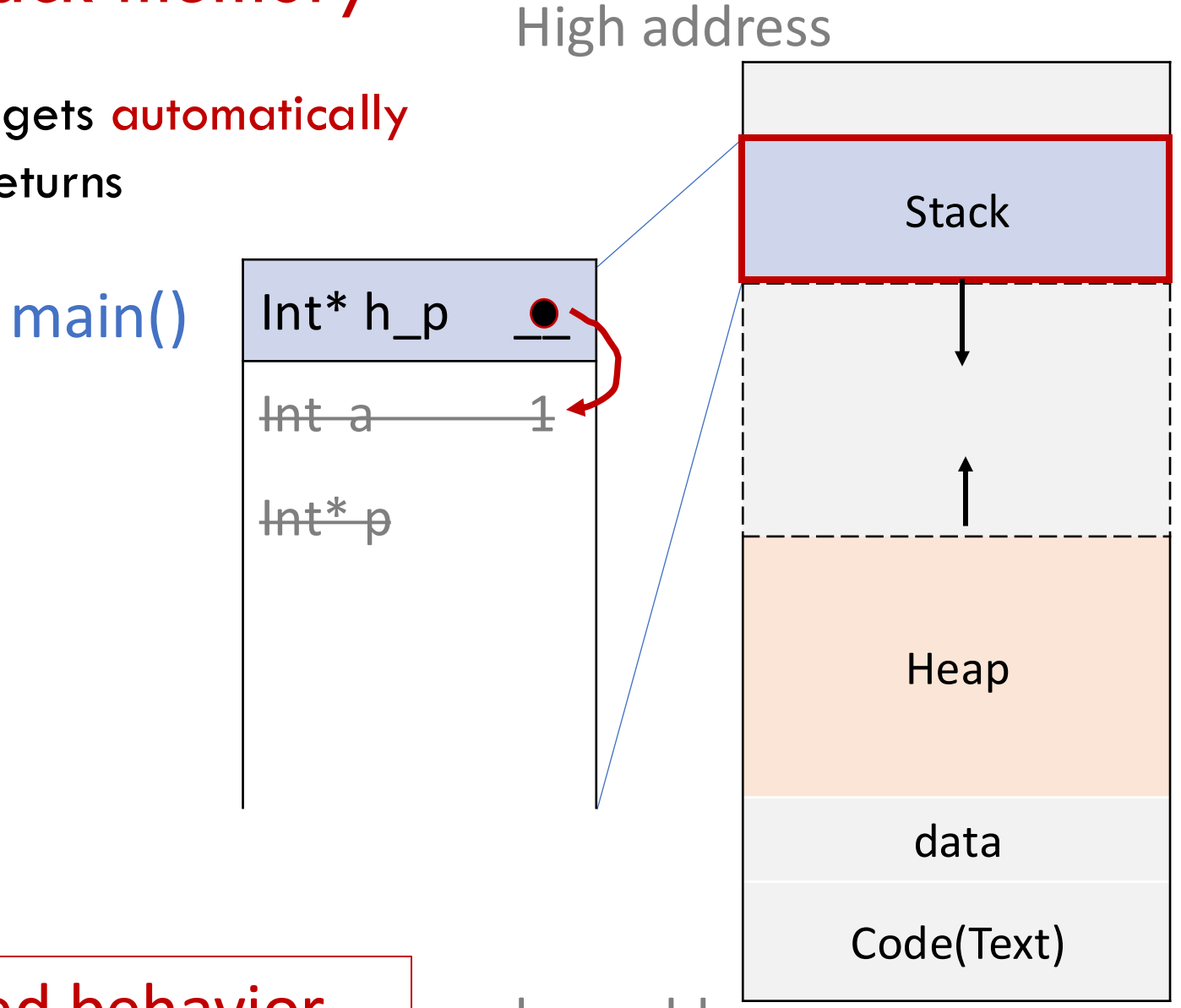


Common mistake with stack memory

- The stack memory of a function gets automatically deallocated after the function returns

```
int* helper()
{
    int a = 3;
    int * p = &a;
    return p;
}
int main(){
    int* h_p = helper();
    …
}
```

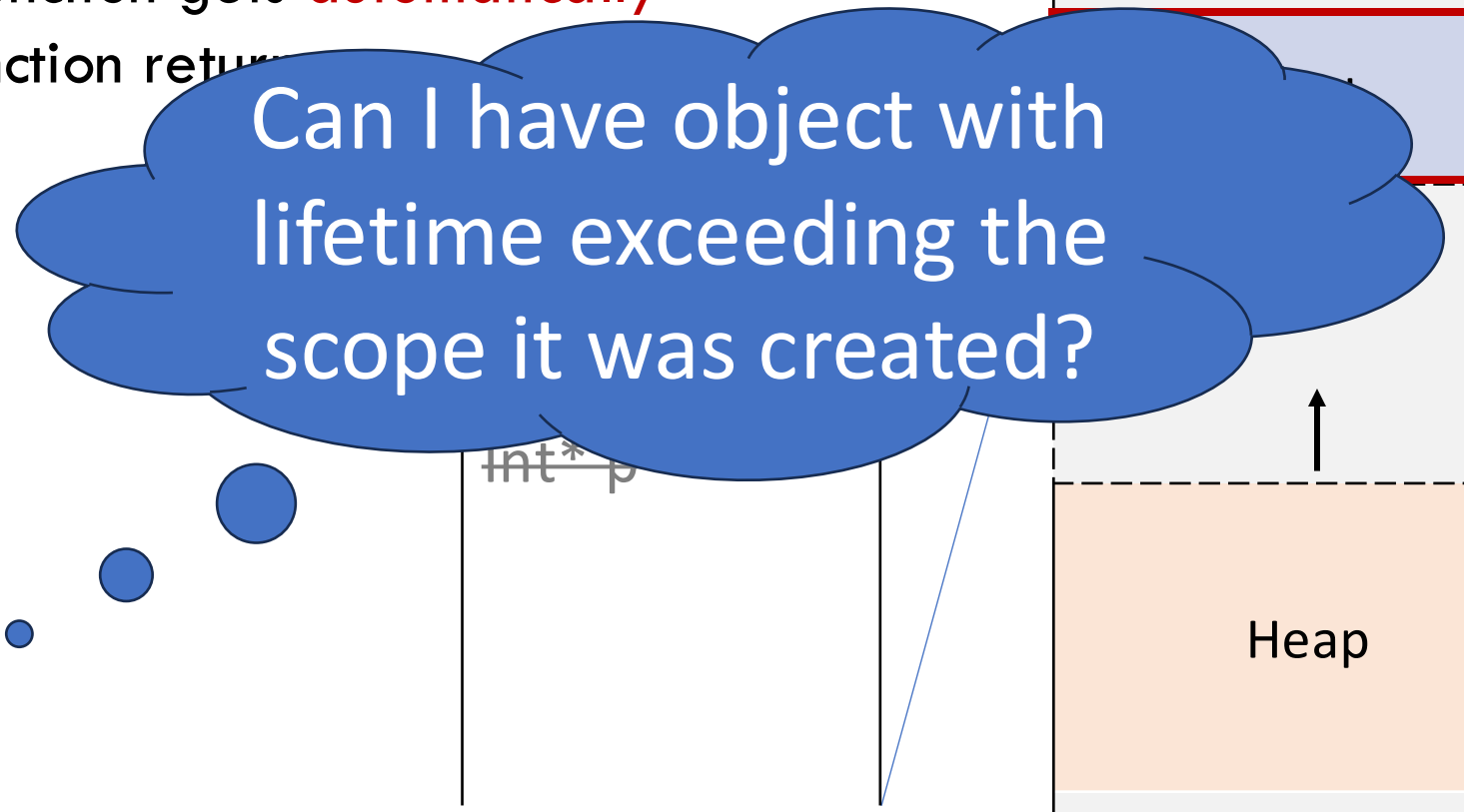
Undefined behavior



Common mistake with stack memory

- The stack memory of a function gets automatically deallocated after the function retur

```
int* helper()
{
    int a = 3;
    int * p = &a;
    return p
}
int main(){
    int* h_p                );
    …
}
```



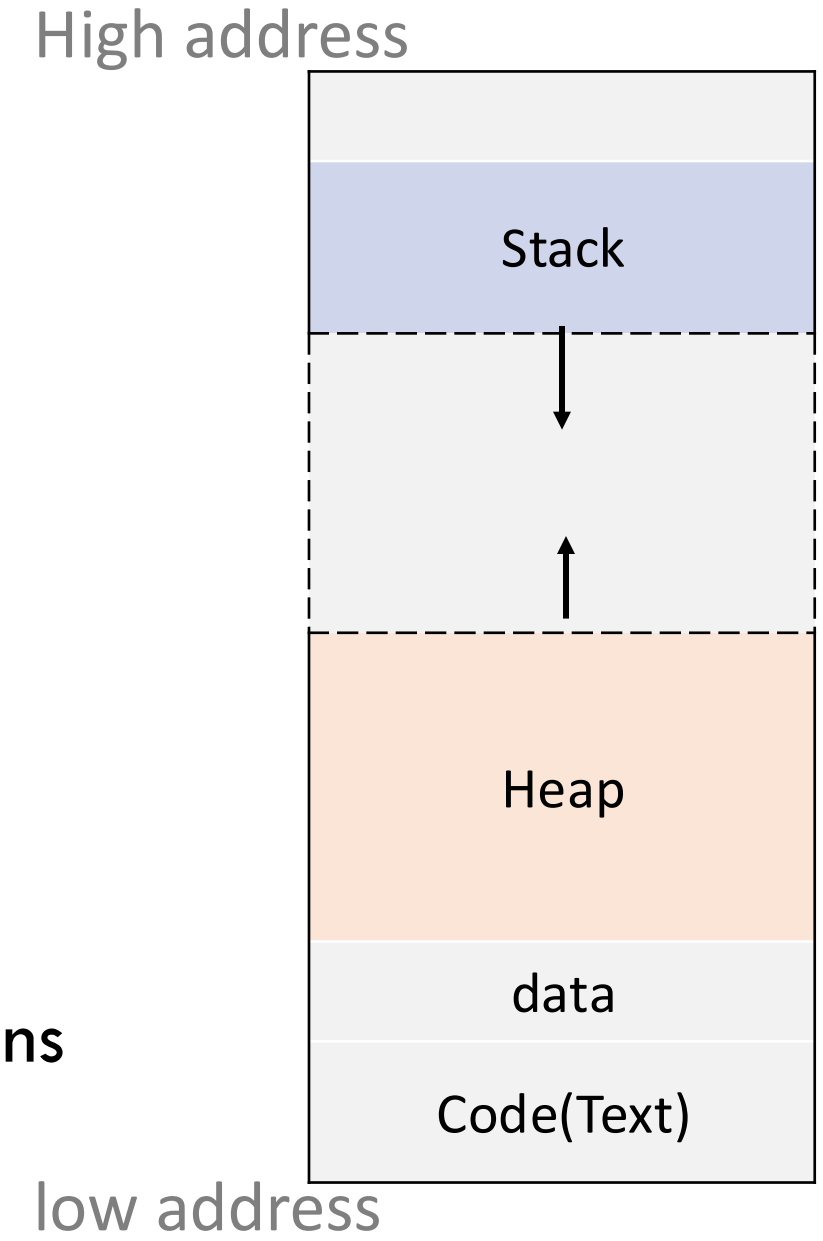
Heap memory

new

delete

Memory

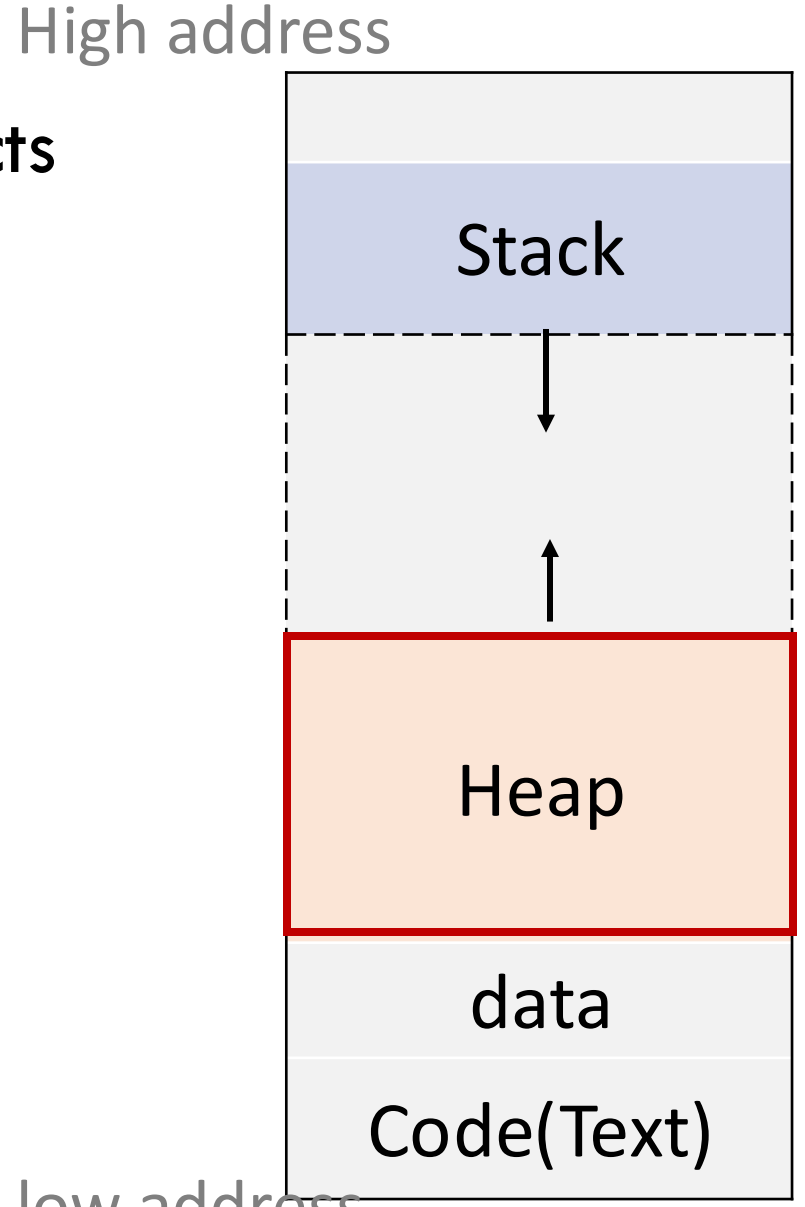
- **Stack**: used for memory needed to call methods(such as local variables), or for inline variables
- **Heap**: Dynamically memory used for programmers to allocate. The memory will often be used for longer period than stack
- **Data**: use for constants and initialized global objects
- **Code**: segments that holds compiled instructions



Heap Memory new expression

- new expression: create and initialize objects on heap (dynamic storage duration)

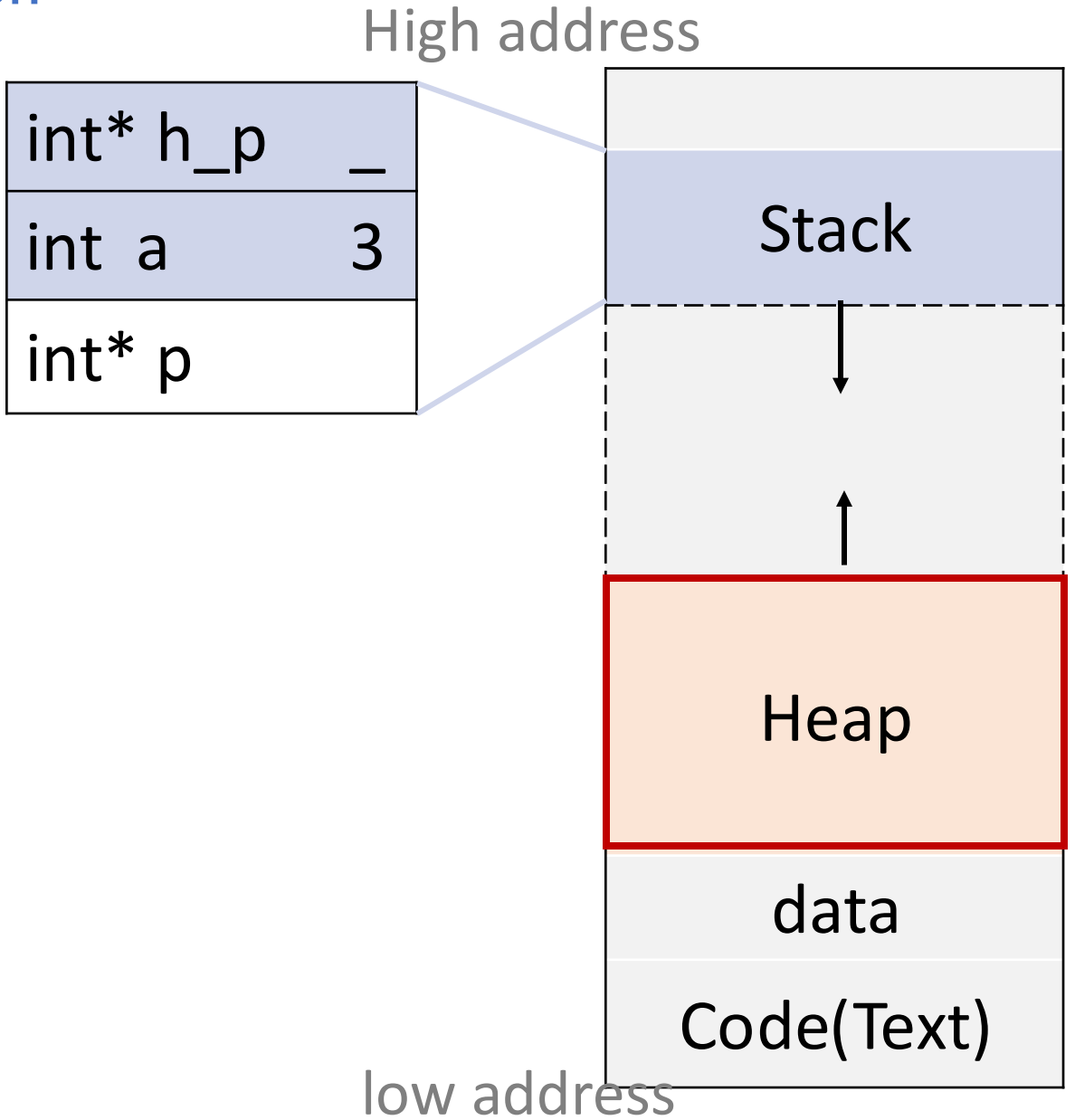
```
int* p = new int(7);

double* arr_p = new double[]{1, 2, 3};

T* obj_p = new T(arg0, arg1, arg2,…);
```



Heap Memory

new expression

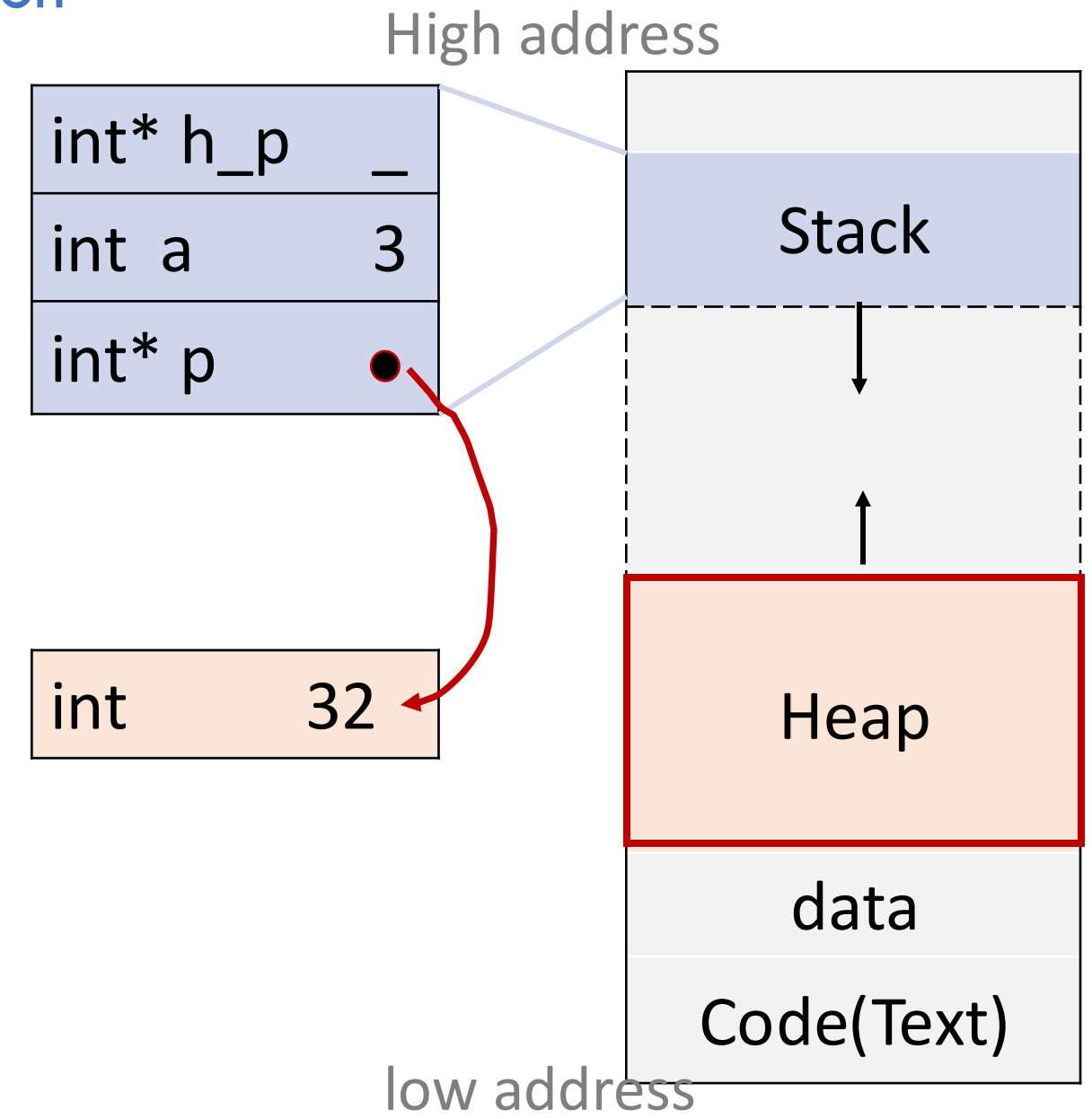
```
int* helper()
{
    int a = 3;
    int * p = new int(32);
    return p;
}

int main(){
     int* h_p = helper();
       …
}
```



Heap Memory

new expression

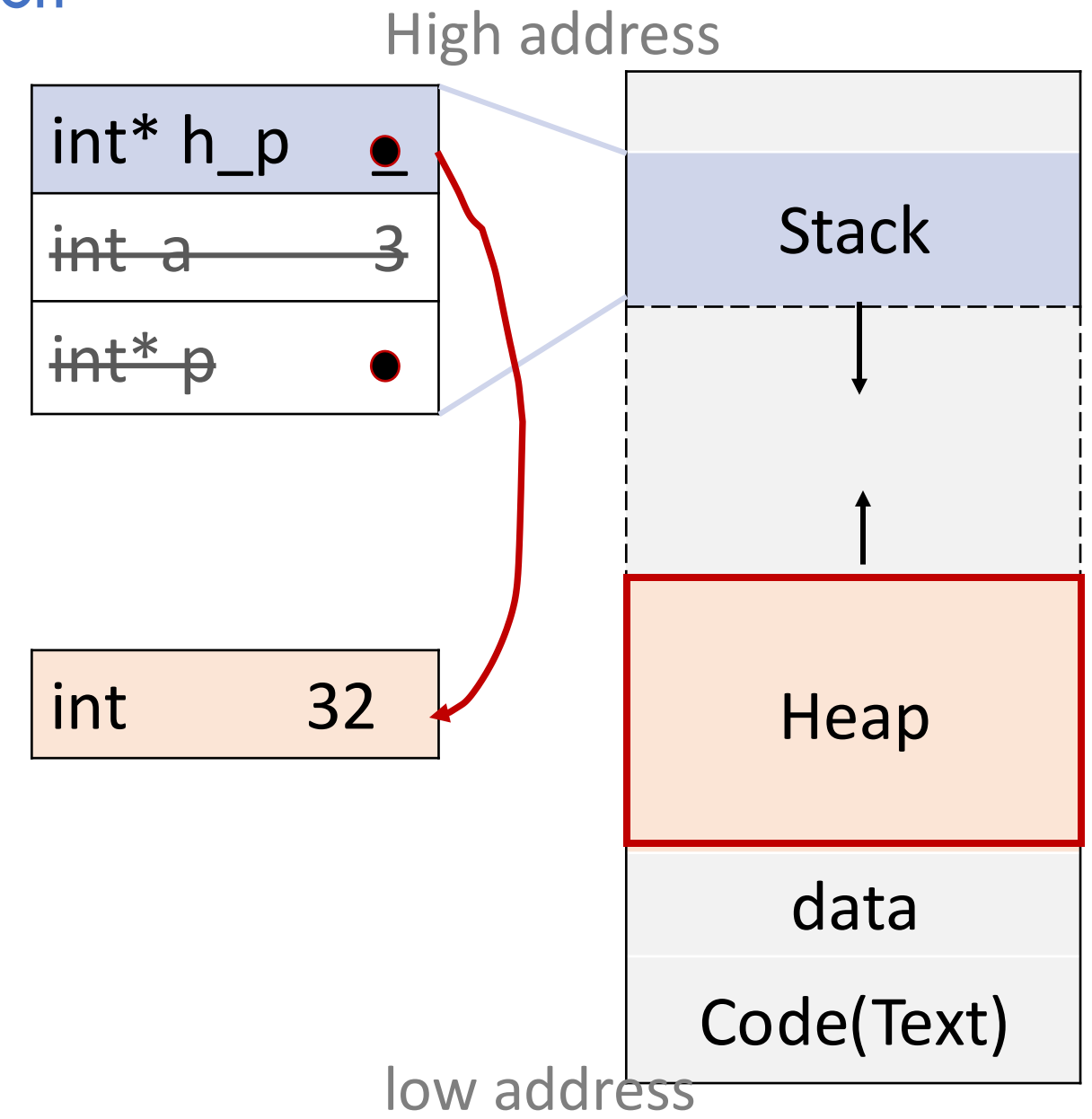
```
int* helper()
{
    int a = 3;
    int * p = new int(32);
    return p;
}

int main(){
     int* h_p = helper();
       …
}
```



Heap Memory

new expression

```
int* helper()
{
    int a = 3;
    int * p = new int(32);
    return p;
}

int main(){
    int* h_p = helper();
    …
}
```



Heap Memory

new expression

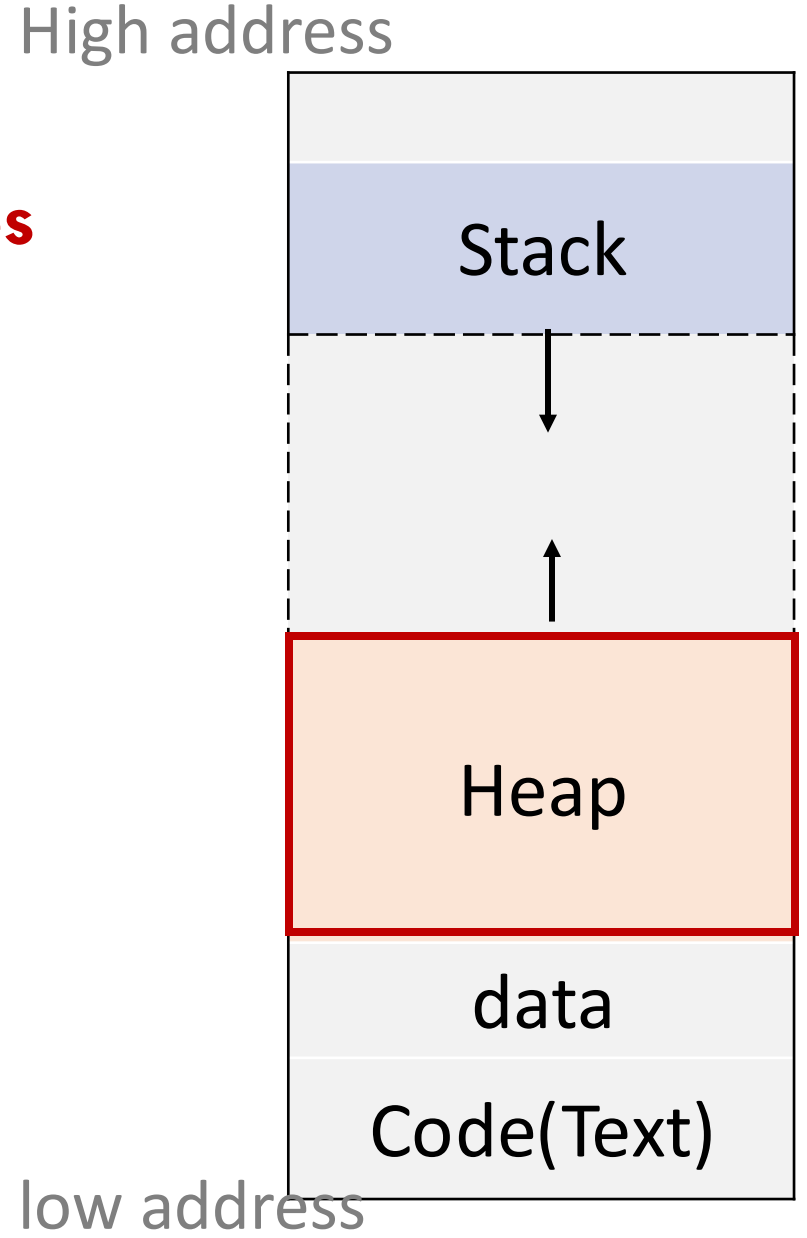
```
int* helper()
{
    int a = 3;
    int * p = new int(32);
    return p;
}

int main(){
    int* h_p = helper();
    …
}
```

NO automatic de-allocation with scope.

Heap Memory delete expression

- delete expression: **destroys** object previously allocated by the new-expression and **releases** obtained memory area back to OS.

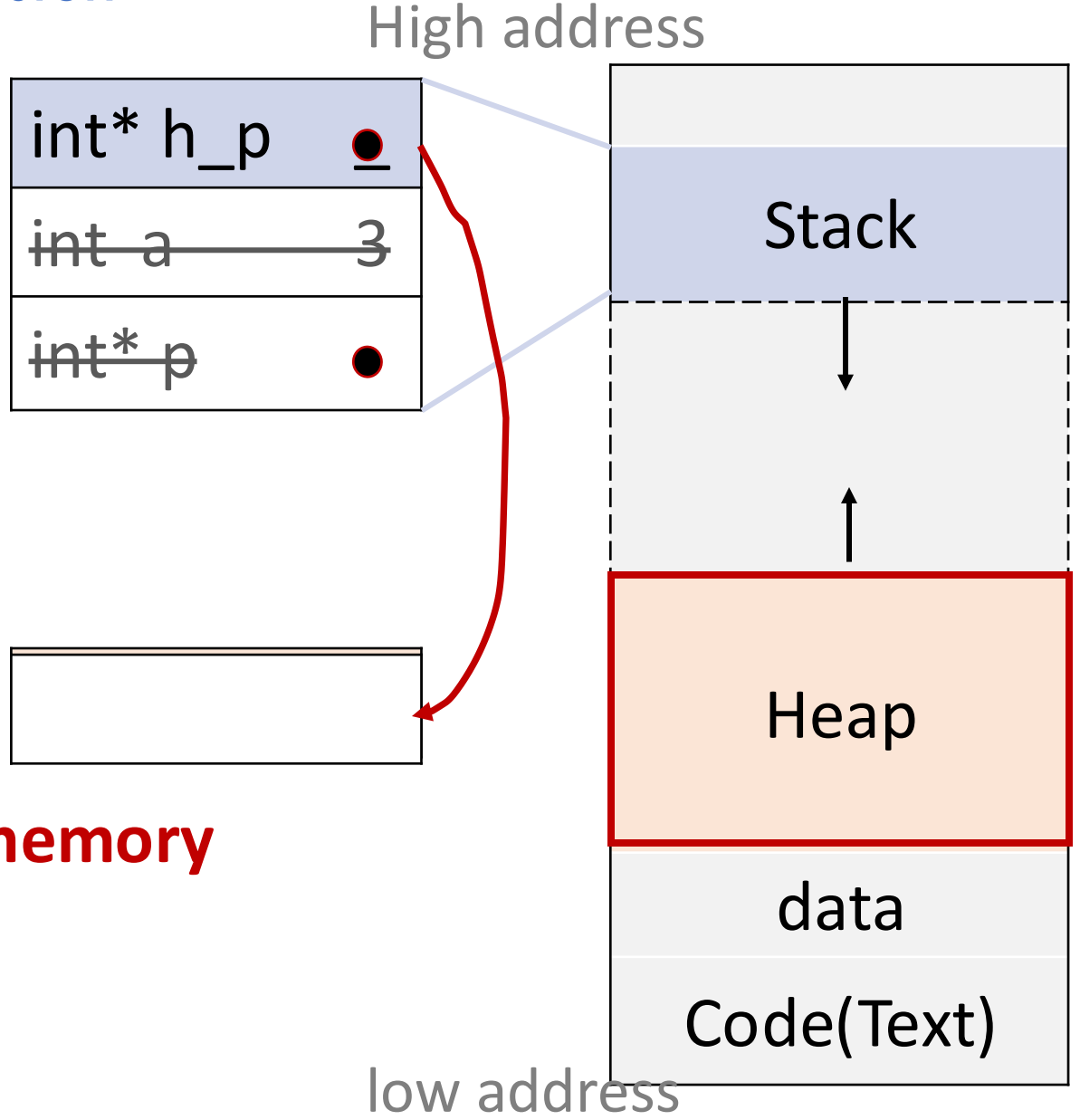
```
int* p = new int(7);
delete p;

double* arr_p = new double[]{1, 2, 3};
delete[] arr_p;

T* p = new T(arg0, arg1, arg2,…);
delete obj_p;
```



Heap Memory

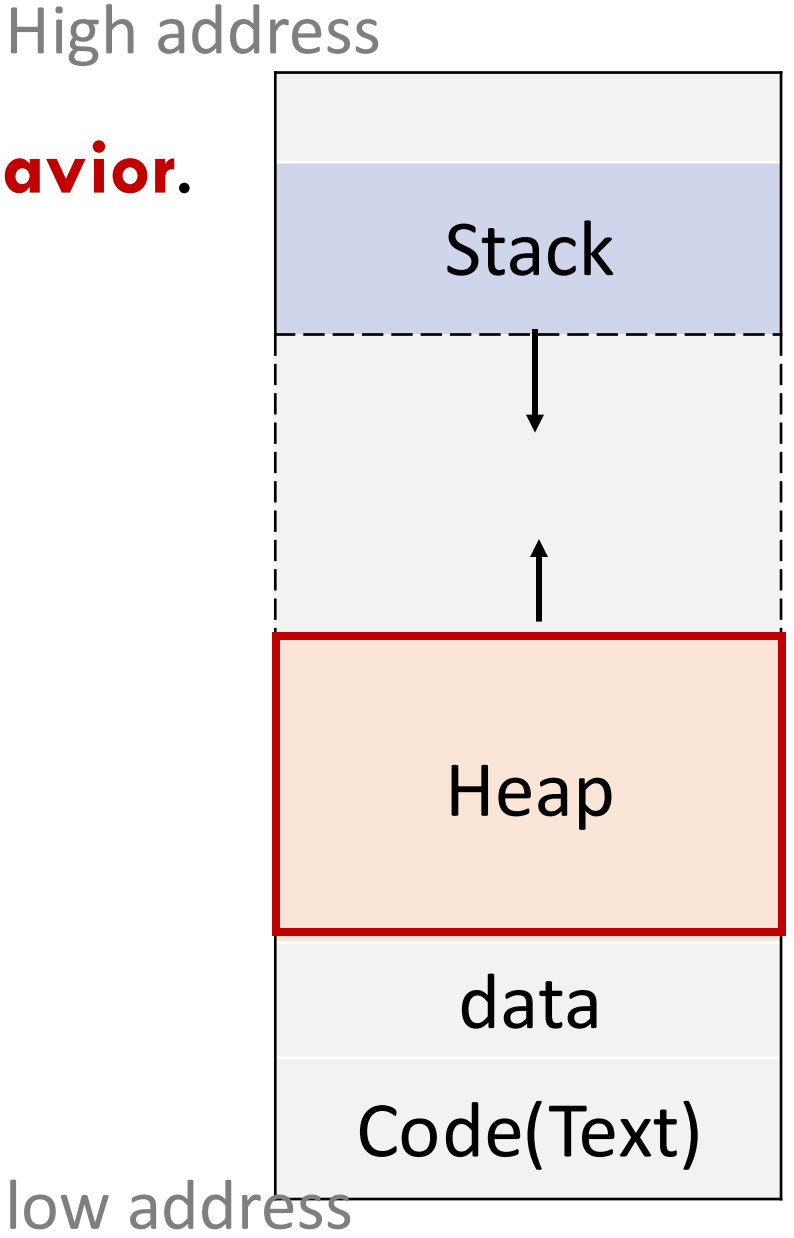
delete expression

```
int* helper()
{
    int a = 3;
    int * p = new int(32);
    return p;
}
int main(){
    int* h_p = helper();
    delete h_p;     // release the memory
    …..
}
```



Heap Memory delete expression

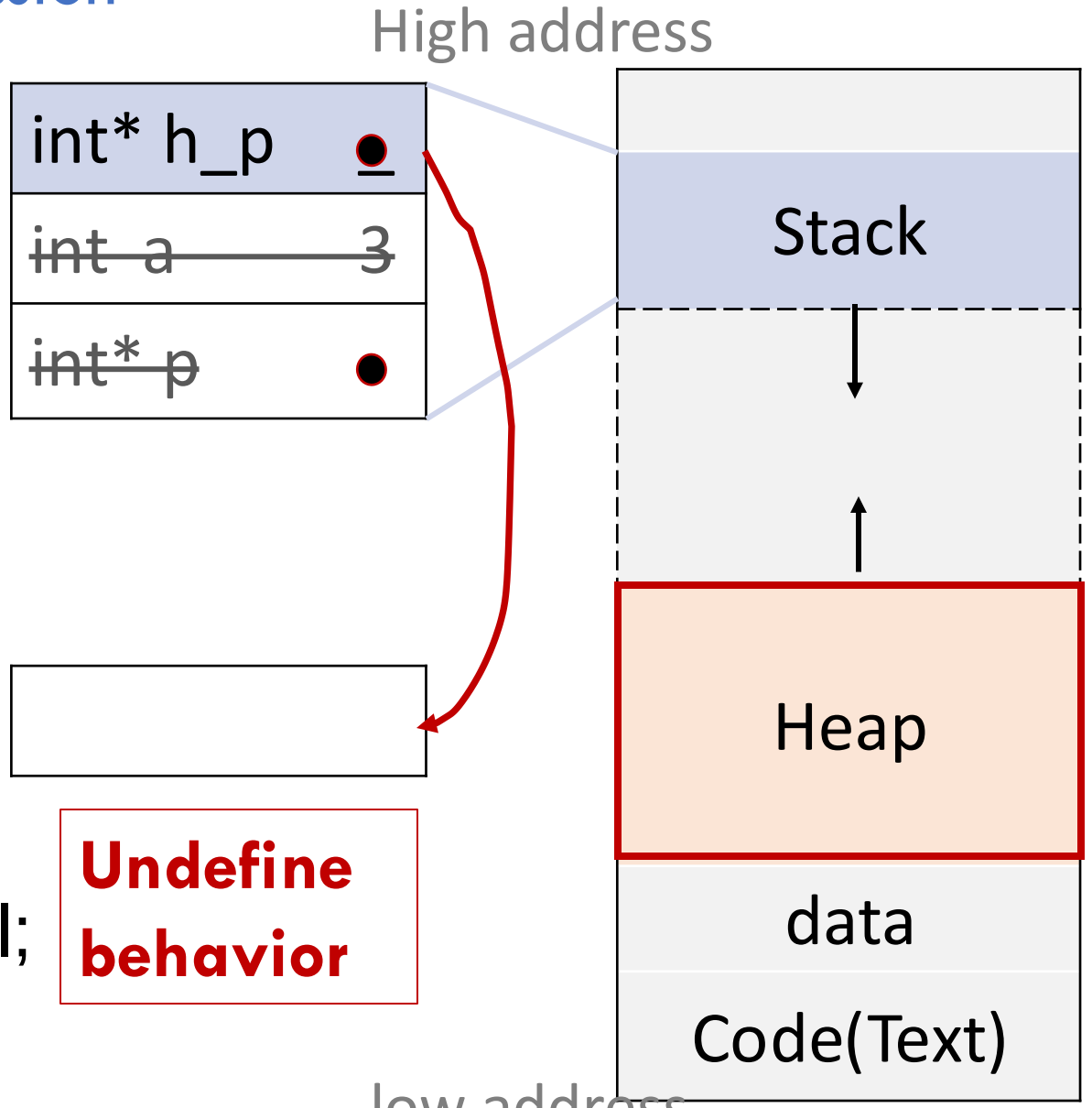
- Using **deleted pointers** causes **undefined behavior**.



Heap Memory

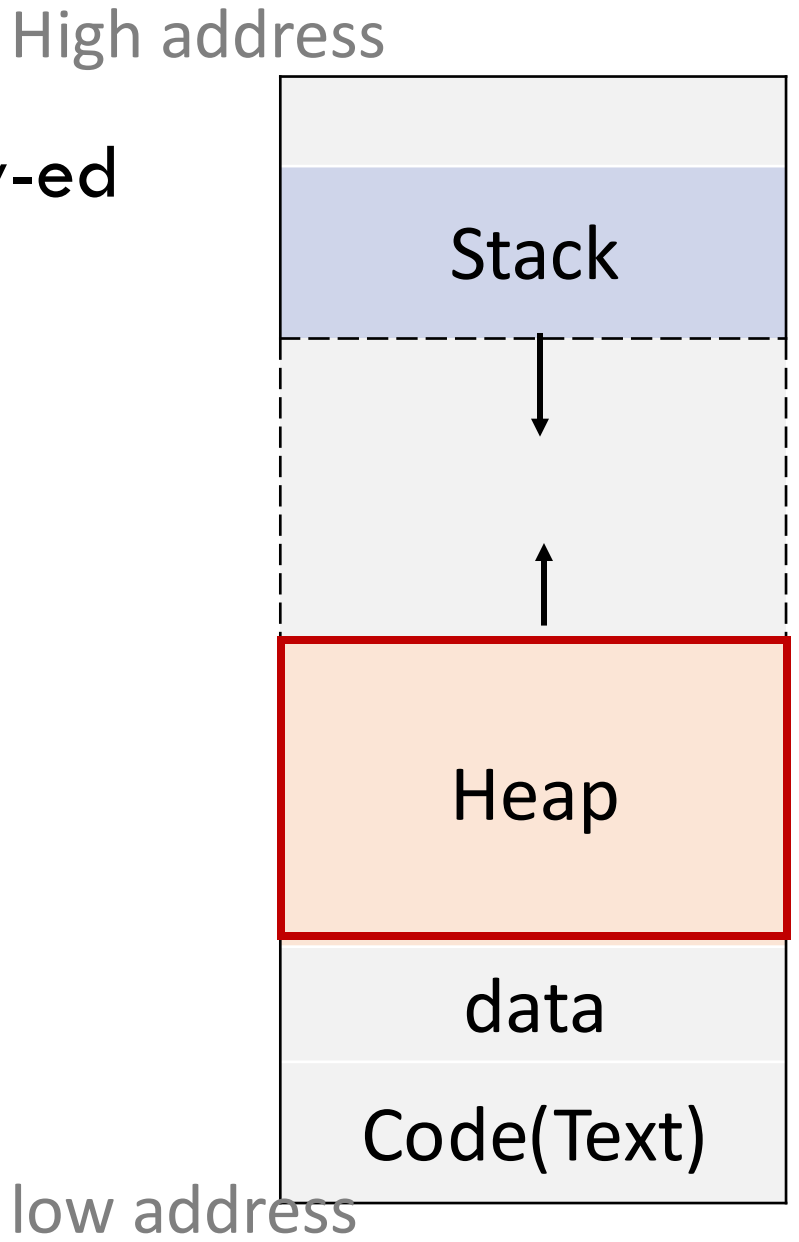
delete expression

```
int* helper()
{
    int a = 3;
    int * p = new int(32);
    return p;
}
int main(){
    int* h_p = helper();
    delete h_p;
    std::cout << *h_p << std::endl;
}
```



Heap Memory delete expression

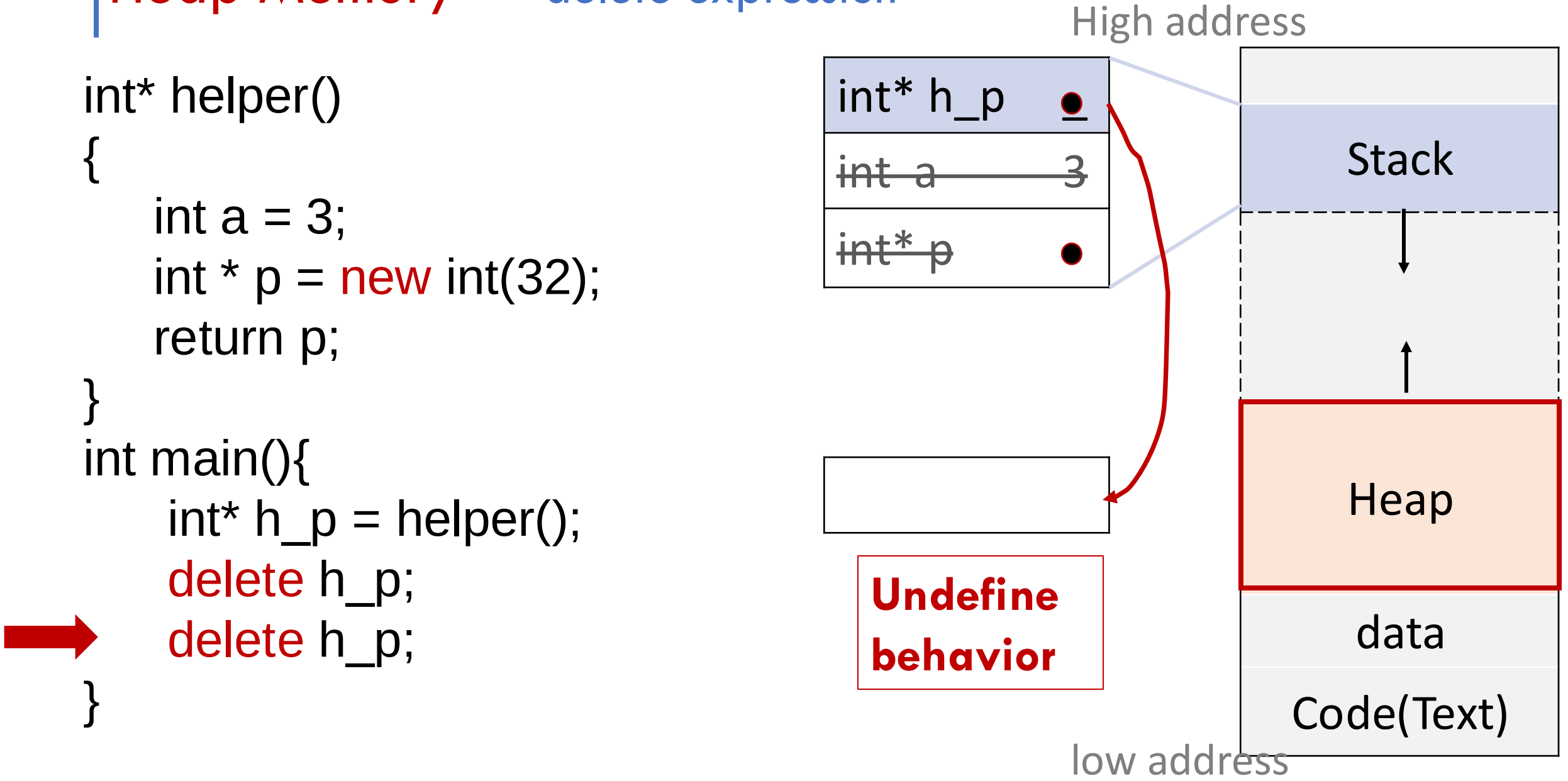
- Using **more than one delete** on the same new-ed pointer causes **undefined behavior**



Heap Memory

delete expression

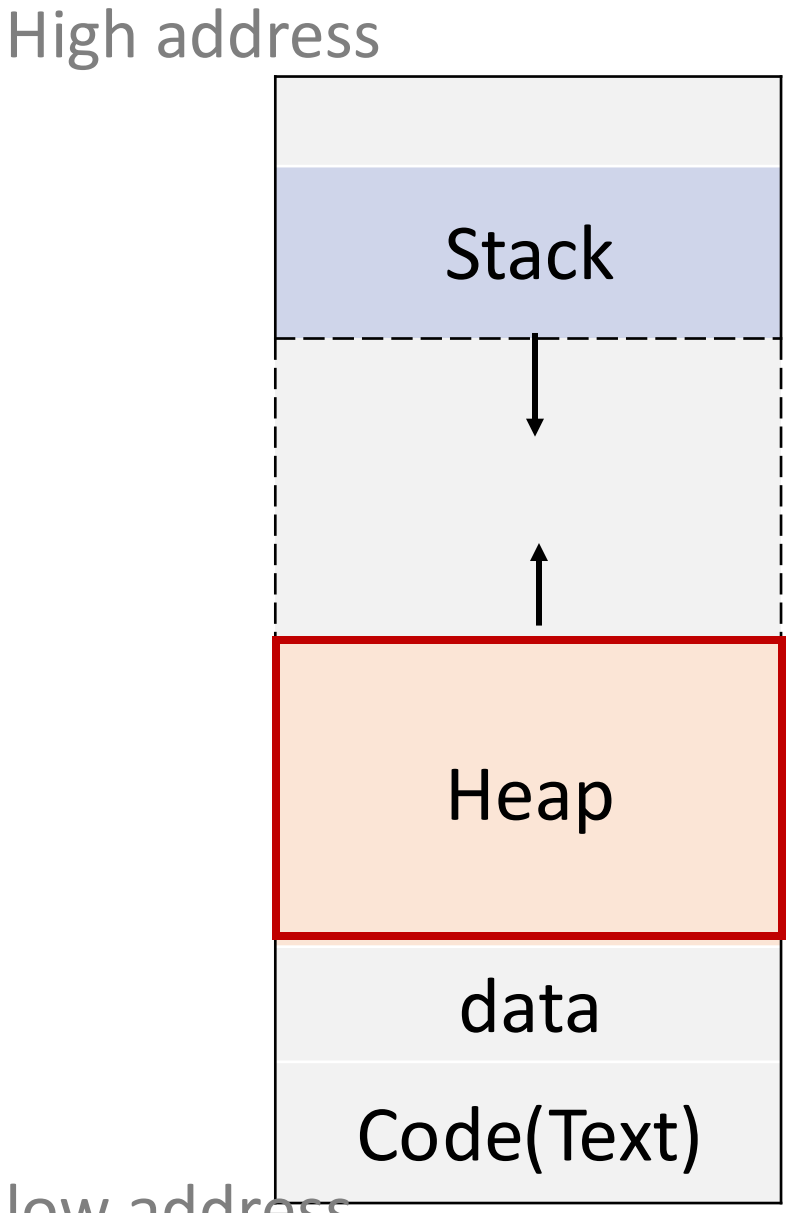
```
int* helper()
{
    int a = 3;
    int * p = new int(32);
    return p;
}
int main(){
    int* h_p = helper();
    delete h_p;
    delete h_p;
}
```



Heap Memory delete expression

- A good practice to set the freed pointers to **nullptr** immediately after delete

```
int *ptr  = new int(10);
delete ptr;
ptr = nullptr;
// set the value of the freed pointer
```



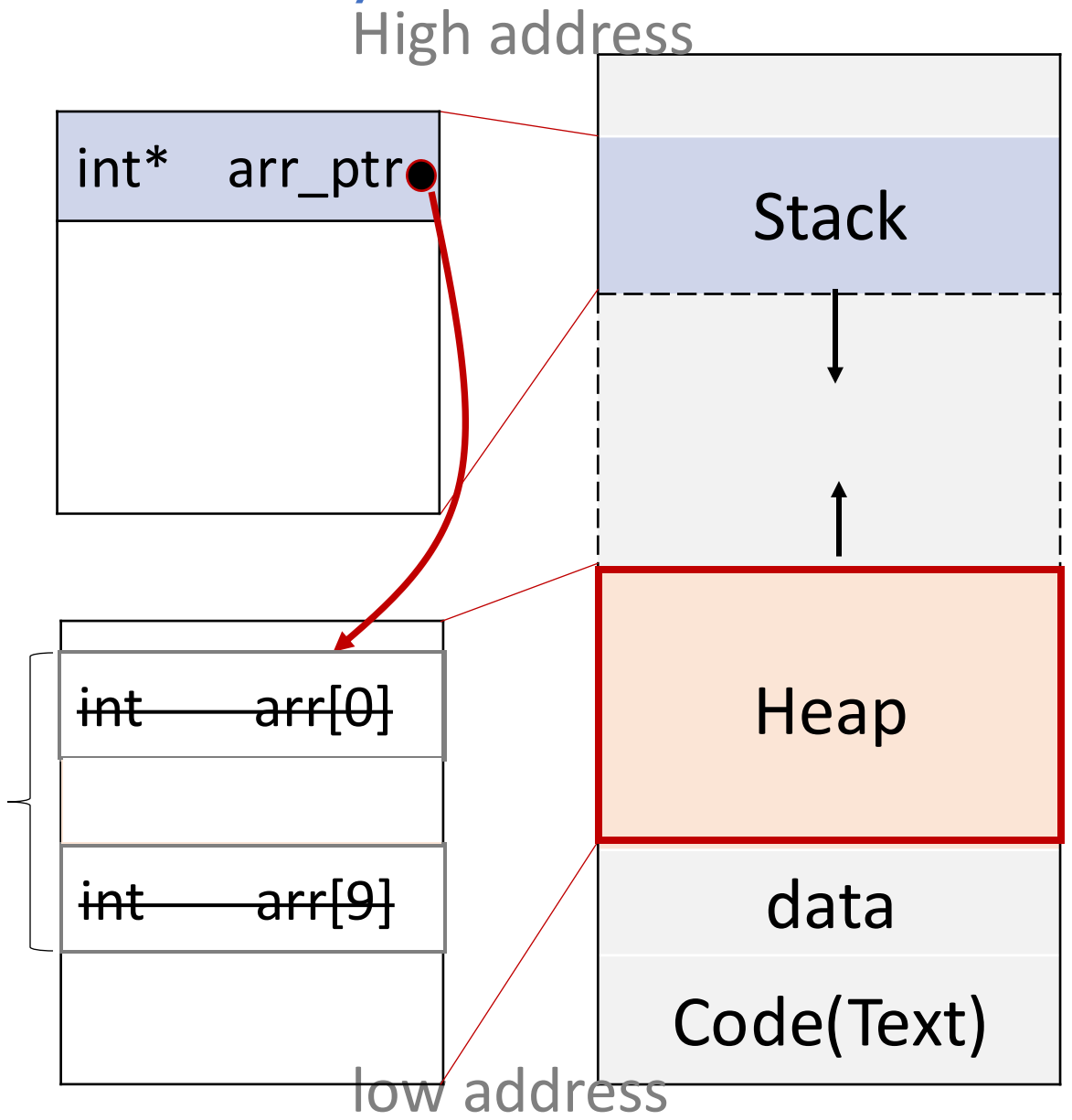
Heap Memory

delete expression for array

```
int * arr_ptr  = new int[10];
```

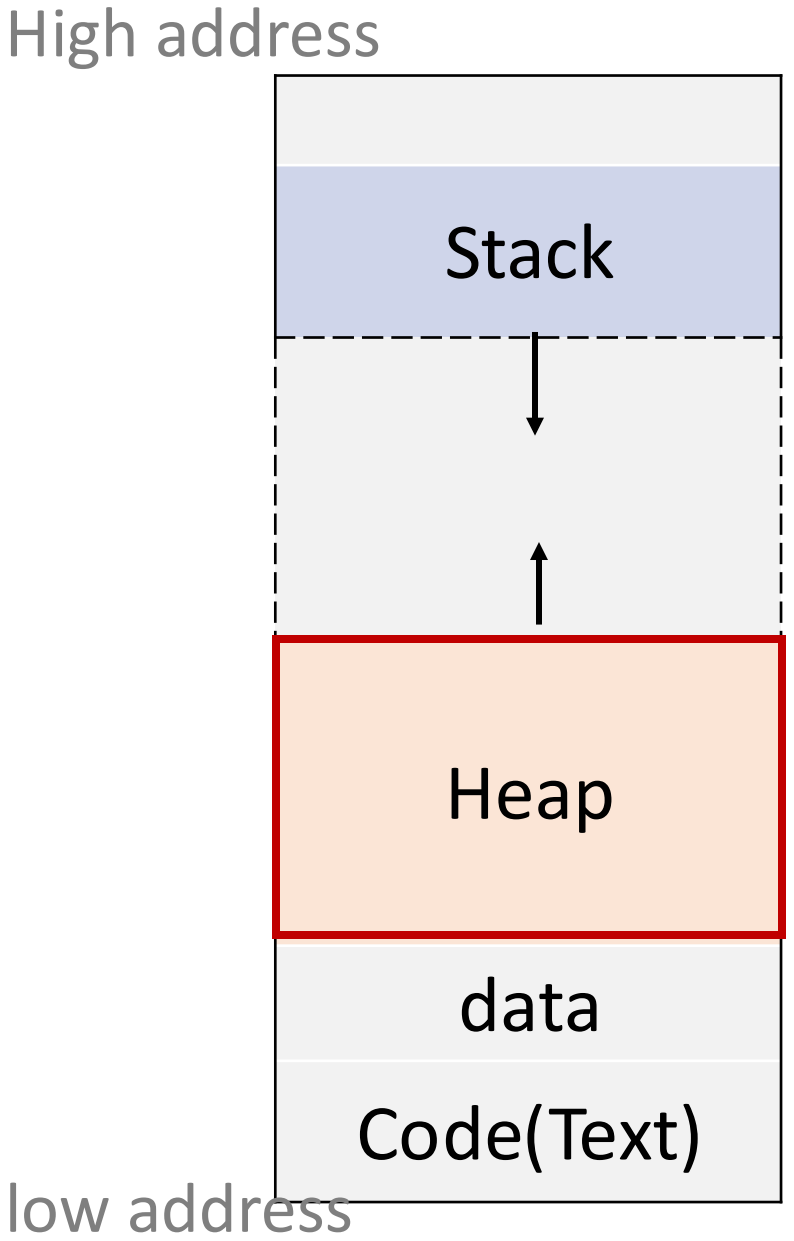
delete arr_ptr;

delete[] arr_ptr;



Heap Memory

- Heap memory is allocated explicitly by **new** expression.
- To release heap memory, program needs explicitly call **delete** expression.
- Unlike stack, memory allocated on heap is **not** necessarily **contiguous**

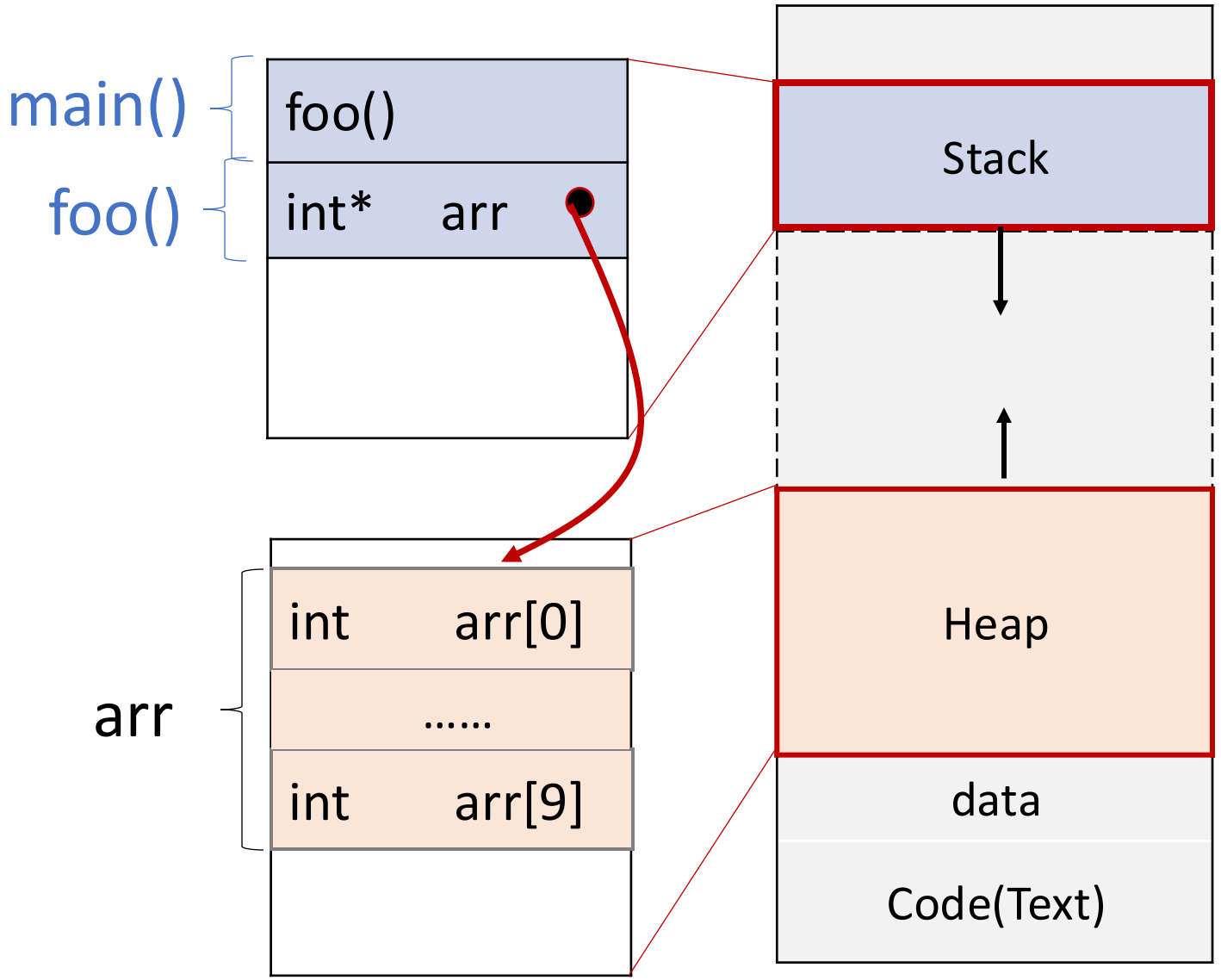


C++ Memory

- How does stack and heap memory work?
- How to use stack and heap memory in my program?

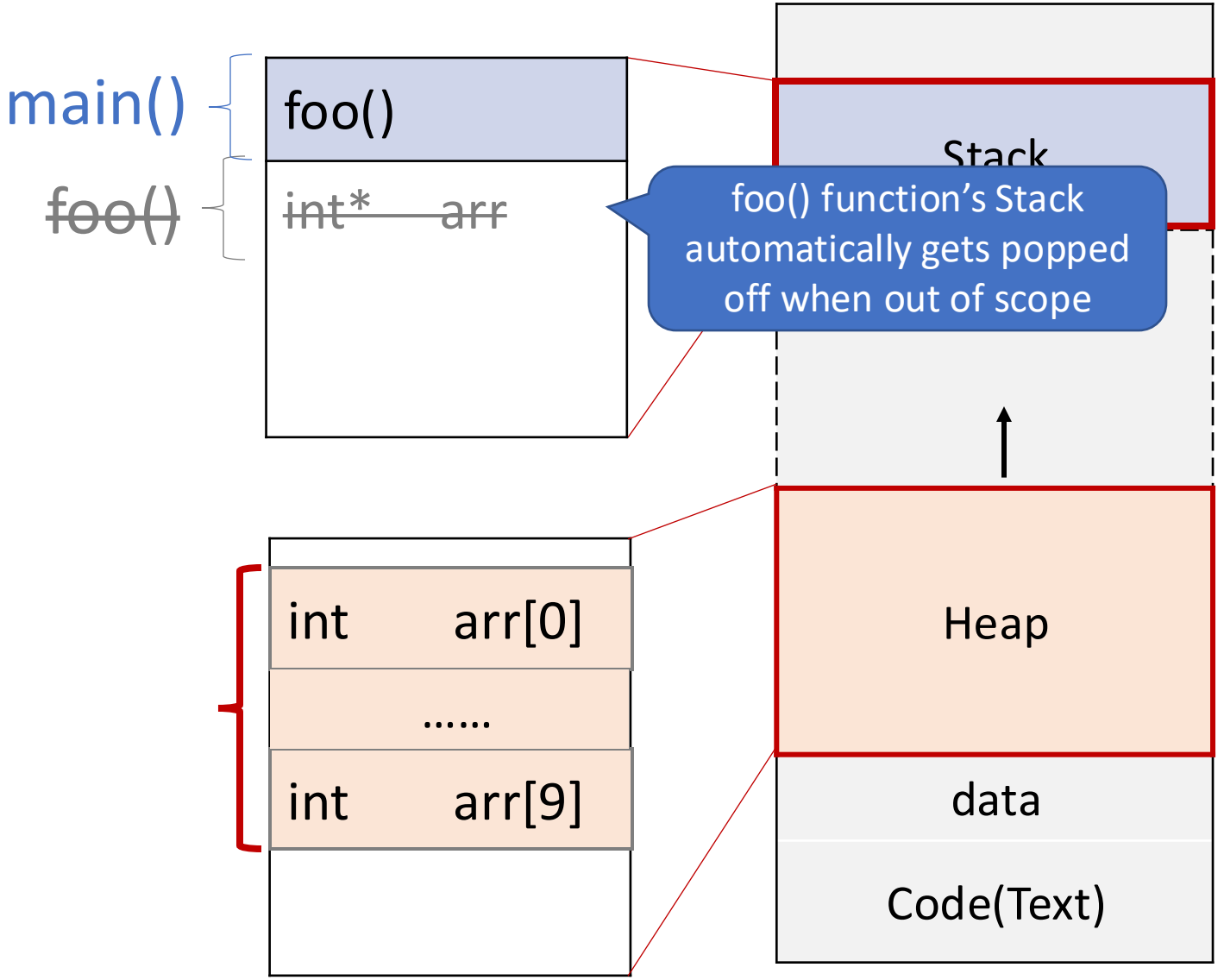
heap-based memory allocation

```
void foo(){
    int* arr = new int[10];
    arr[0] = 0;
}

int main(){
    foo();
    .......
}
```



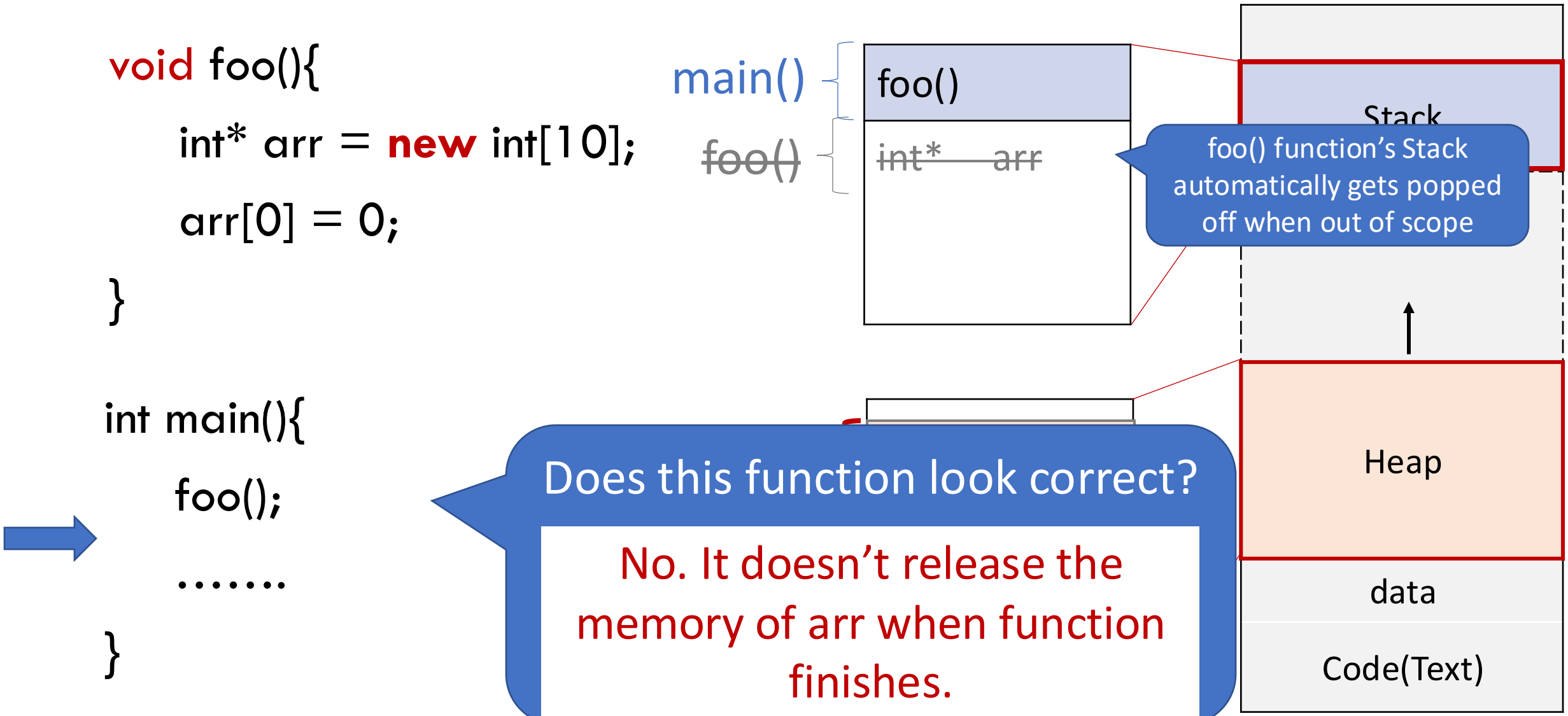
heap-based memory allocation

```
void foo(){
    int* arr = new int[10];
    arr[0] = 0;
}

int main(){
    foo();
    ……
}
```



heap-based memory allocation

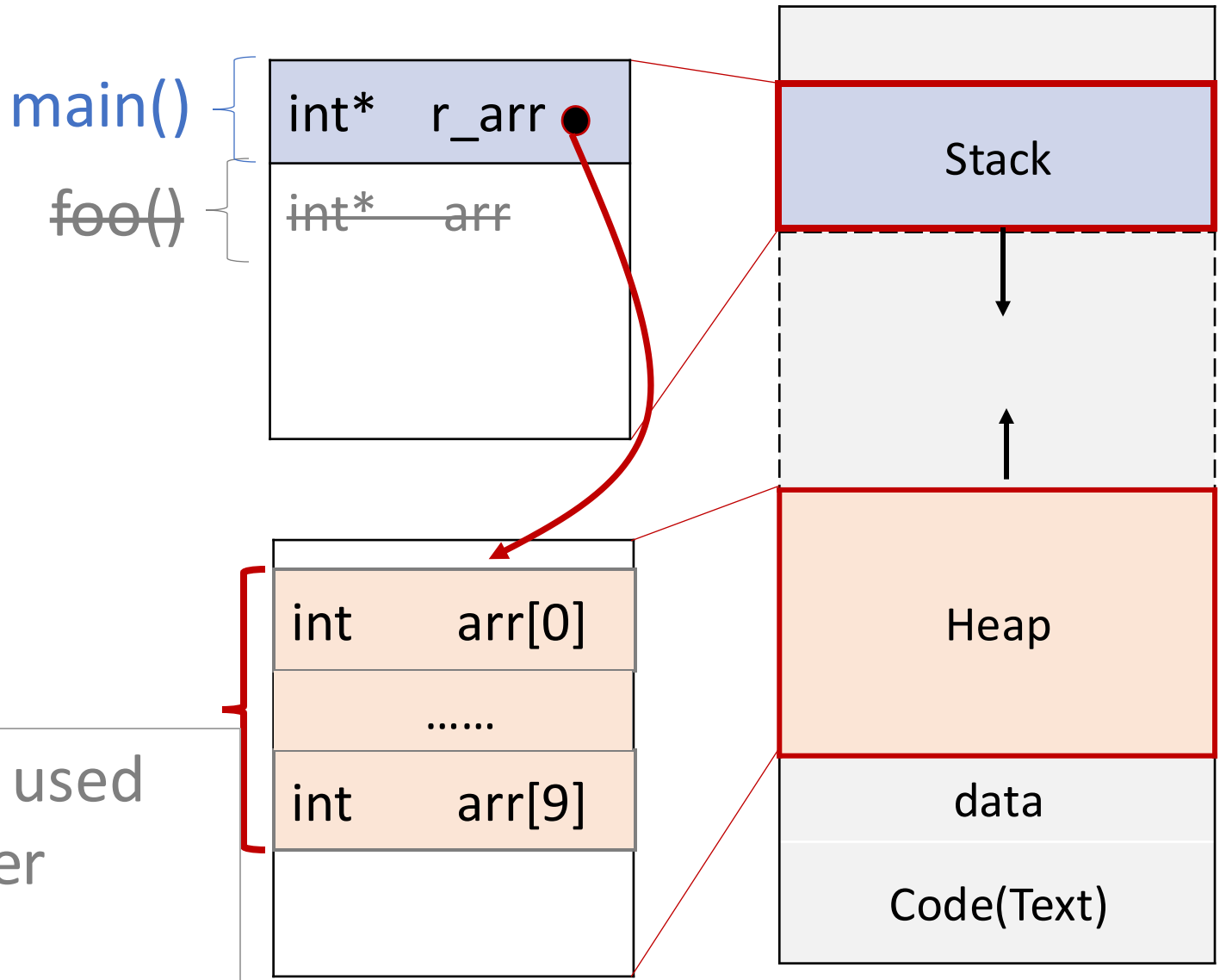
```
void foo(){
    int* arr = new int[10];
    arr[0] = 0;
}

int main(){
    foo();
    .....
}
```



heap-based memory allocation

```
int* foo(){
    int* arr = new int[10];
    arr[0] = 0;
    return arr;
}

int main(){
    int* r_arr = foo();
    ……..
}
```

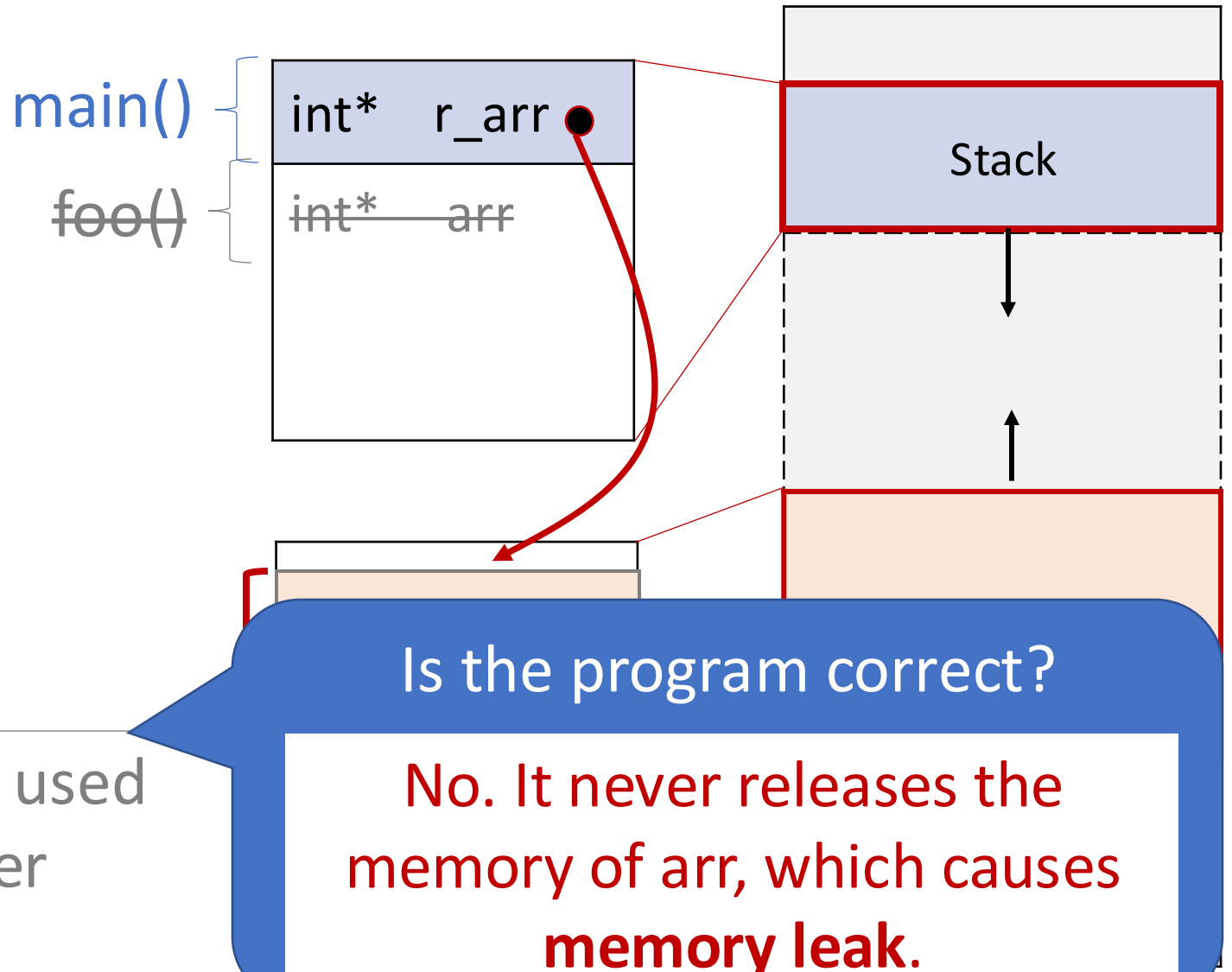
// r_arr is neither used nor deleted in later program



demo

heap-based memory allocation

```
int* foo(){
    int* arr = new int[10];
    arr[0] = 0;
    return arr;
}

int main(){
    int* r_arr = foo();
    ……. // r_arr is neither used nor deleted in later program
}
```



Is the program correct?

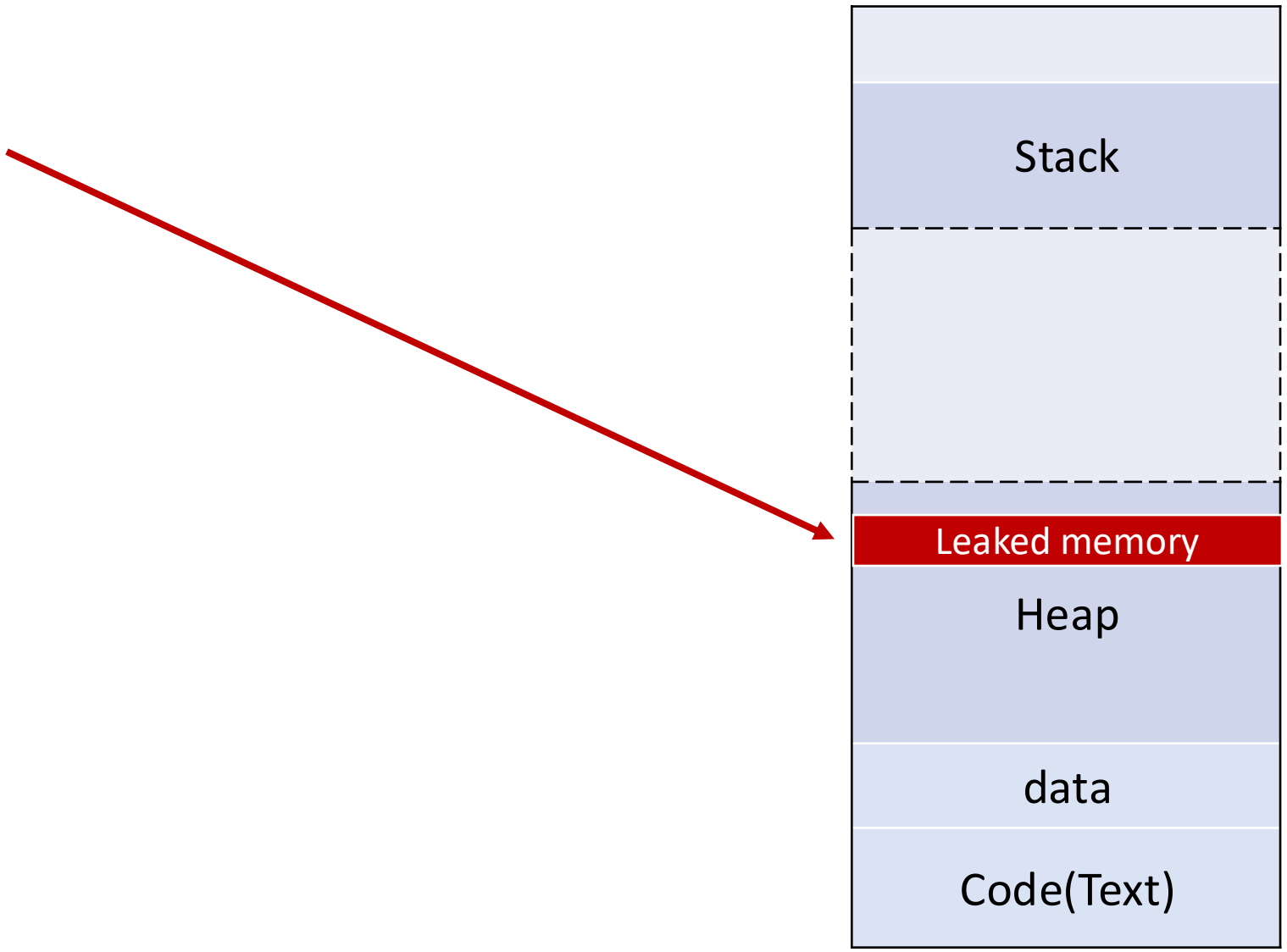
No. It never releases the memory of arr, which causes **memory leak**.

What is memory leak in C++?

- Memory leakage in C++ is when programmers allocates heap-based memory by using new keyword and forgets to deallocate the memory

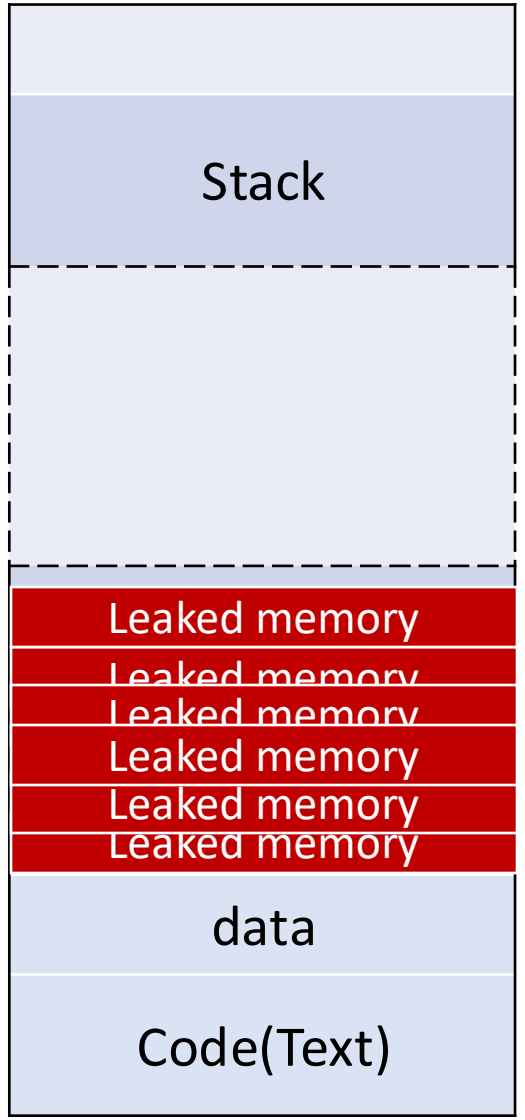
Memory Leak

```
int* foo(){
    int* arr = new int[10];
    arr[0] = 0;
    return arr;
}

int main(){
    int* r_arr = foo();
    .......
}
```



Memory Leak

```
int* foo(){
      int* arr = new int[10];
      arr[0] = 0;
      return arr;
}
int main(){
     for( int i = 0; i < 100; i++){
           int* r_arr = foo();
     }.....
}
```



Consequences of memory leak ?

- Reduces the amount of available memory, negatively impacts the runtime performance
- If memory leaks accumulate over time and left unchecked, may thrash or even crash a program

Memory Leak

demo

- What is memory leak in C++?
- Consequences of memory leak?
- How to check if my program has memory leak?
  - **Valgrind:** https://valgrind.org

```
$ valgrind --leak-check=full ./exec
```

Memory Leak

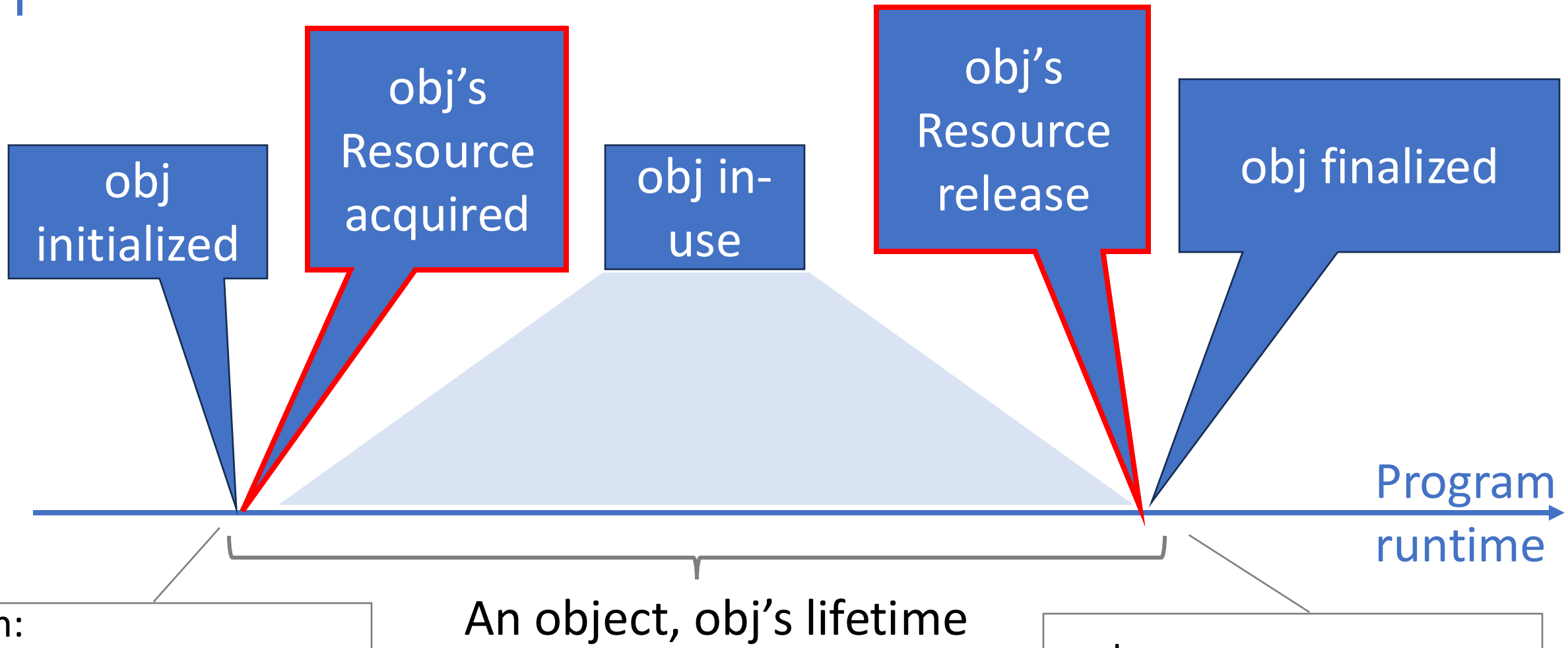
- What is memory leak in C++?
- How to check if my program has memory leak?
- How to avoid memory leak in my program?
  - Follow **RAII principle**(Resource acquisition is initialization)
  - Use **smart pointers** instead of raw pointers

RAII Principle

- **RAII principle**(Resource acquisition is initialization):
  - Resource acquisition must succeed for initialization to succeed.
  - The resource is guaranteed to be held during its lifetime(between when initialization finishes and finalization starts)
  - The resources need to be released when not used.

https://en.cppreference.com/w/cpp/language/lifetime

Example.



Begin:
- storage for its type is obtained
- initialization is complete

end:
- the object is destroyed
- Obj's storage is released

C++ pointers

Types of Pointers

- C-style raw pointers
- Smart pointers
  - unique_ptr
  - shared_ptr

Ownership

- For C++ ownership is the responsibility for cleanup.
  - C-style raw pointer : does not represents ownership — can do anything you want with it, and you can happily use it in ways which lead to memory leaks or double-frees.

C-style raw pointers

```
int* p = new int(7);
```

```
double* arr_p = new double[]{1, 2, 3};
```

```
T* obj_p = new T(arg0, arg1, arg2,…);
```

// a pointer to an object of class T on heap

Rectangle example from last recitation

```
#pragma once
class Rectangle{
        float width;
        float length;
        float area;
public:
        Rectangle();
        Rectangle(float w, float l);
        ~Rectangle();
        float& getArea();
... };
```

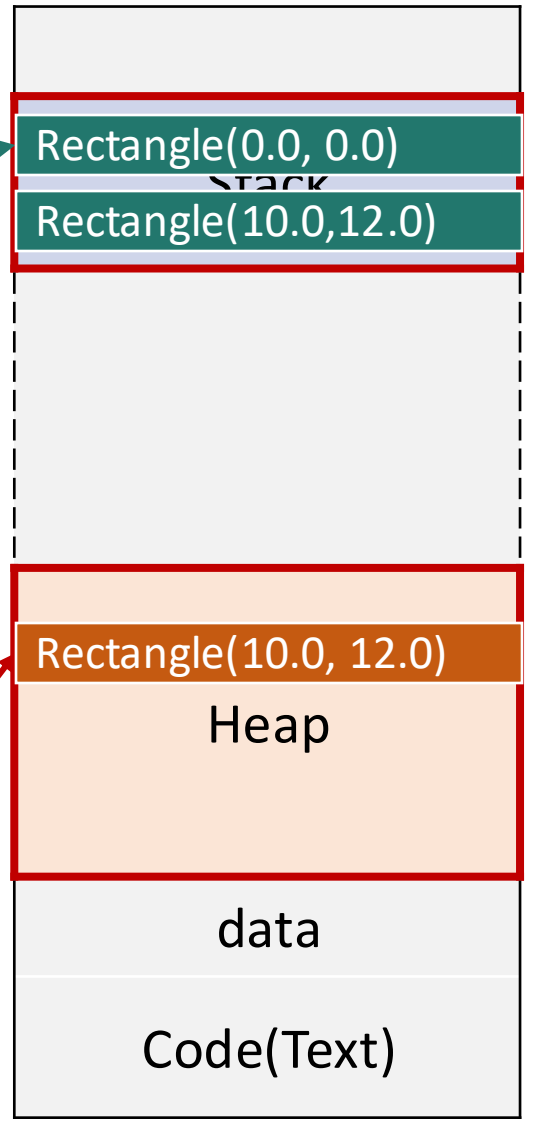
rectangle.hpp

```
#include "rectangle.hpp”
Rectangle::Rectangle(){
        … …
}
Rectangle::Rectangle(float w, float l){
        … …
}
Rectangle::~Rectangle(){
        … …
}
float& Rectangle::getArea(){
… }
```

rectangle.cpp

Rectangle example

```
#include "rectangle.hpp”

int main(){

   Rectangle rec;                    // Default constructor

   Rectangle explicit_rec = Rectangle(10.0,12.0);
                                     // Parameterized constructor

   Rectangle* rec;                   // Declare a pointer with Rectangle
                                     type, no Rectangle object created

   Rectangle* explicit_rec_ptr = new Rectangle(10.0, 12.0);
                                     // create a Rectangle object on heap
```



Types of Pointers

- C-style raw pointers
- Smart pointers: wrapper of a raw pointer and make sure the object is deleted if it is no longer used
  - unique_ptr
  - shared_ptr

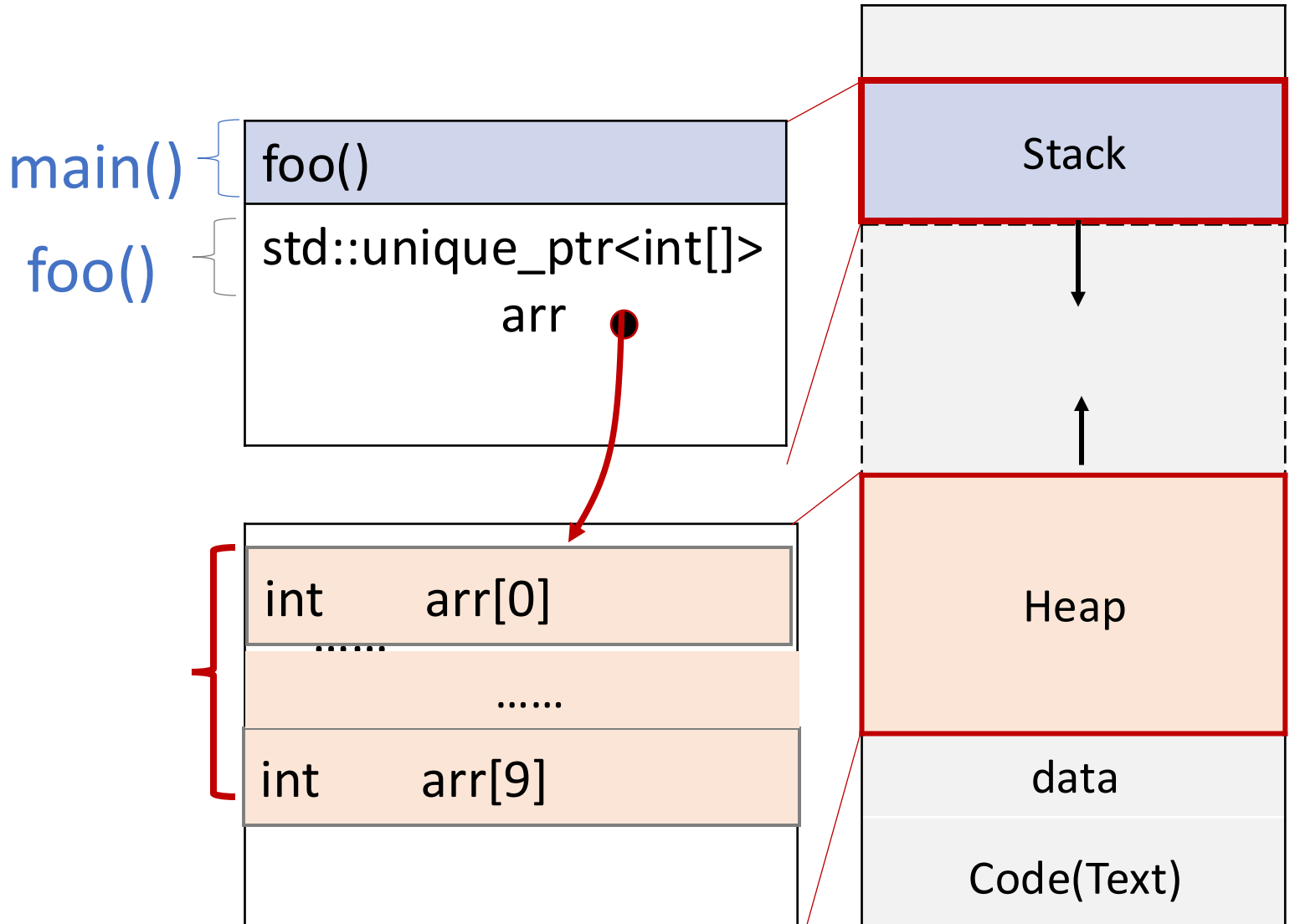
smart pointer: unique_ptr

- Owns and manages another object through a pointer and disposes of that object when the unique_ptr goes out of scope
- Represents the sole owner of resource and will get destroyed and cleaned up correctly
- Provides safety to classes and functions that handle objects with dynamic lifetime, by guaranteeing deletion on both normal exit and exit through exception

```
template <
    class T,
    class Deleter
> class unique_ptr<T[], Deleter>;
```

https://en.cppreference.com/w/cpp/memory/unique_ptr

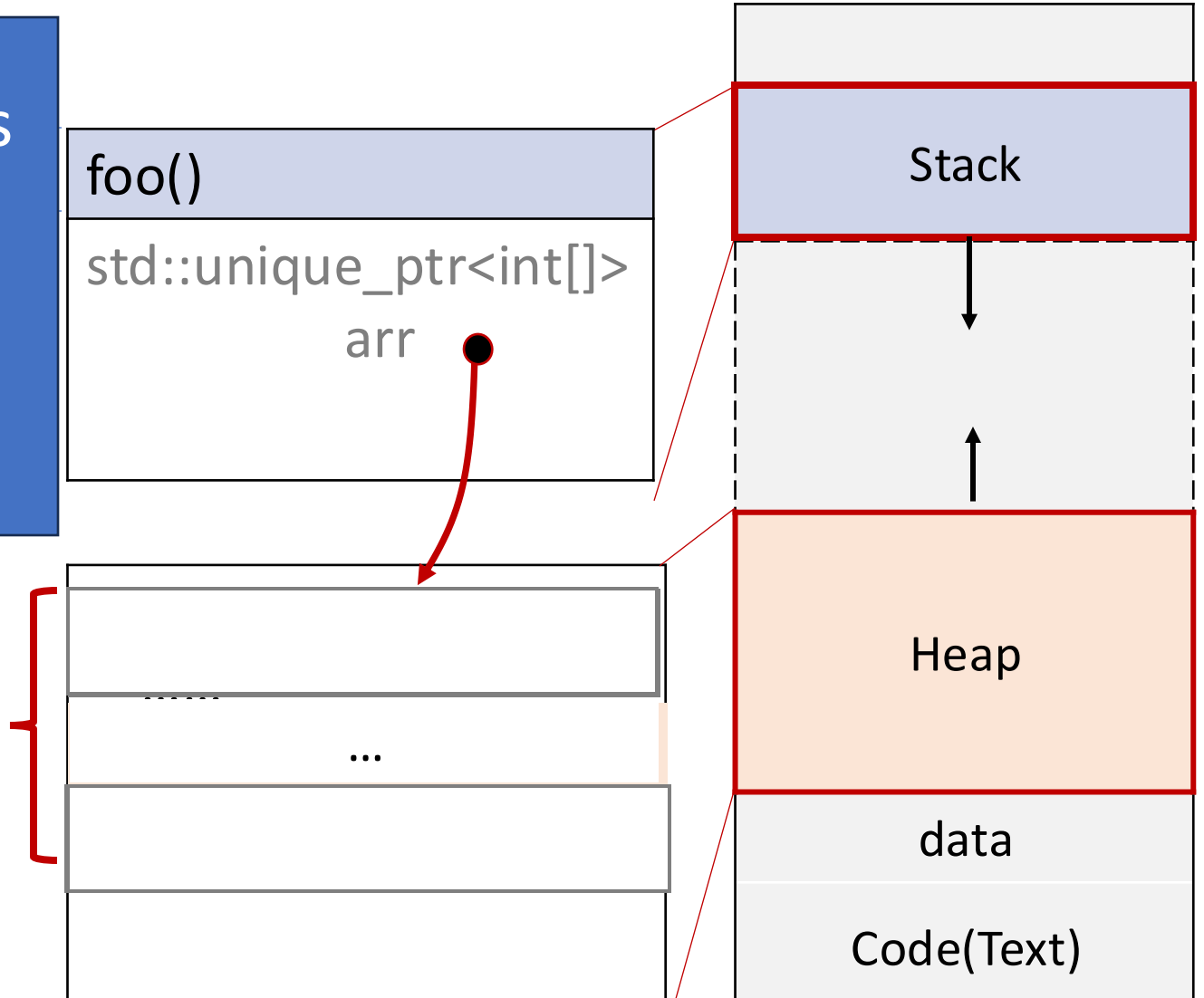
```
void foo(){
    std::unique_ptr<int[]> arr = std::make_unique<int[]>(10);
    arr[0] = 0;
}

int main(){
    foo();
    .......
}
```



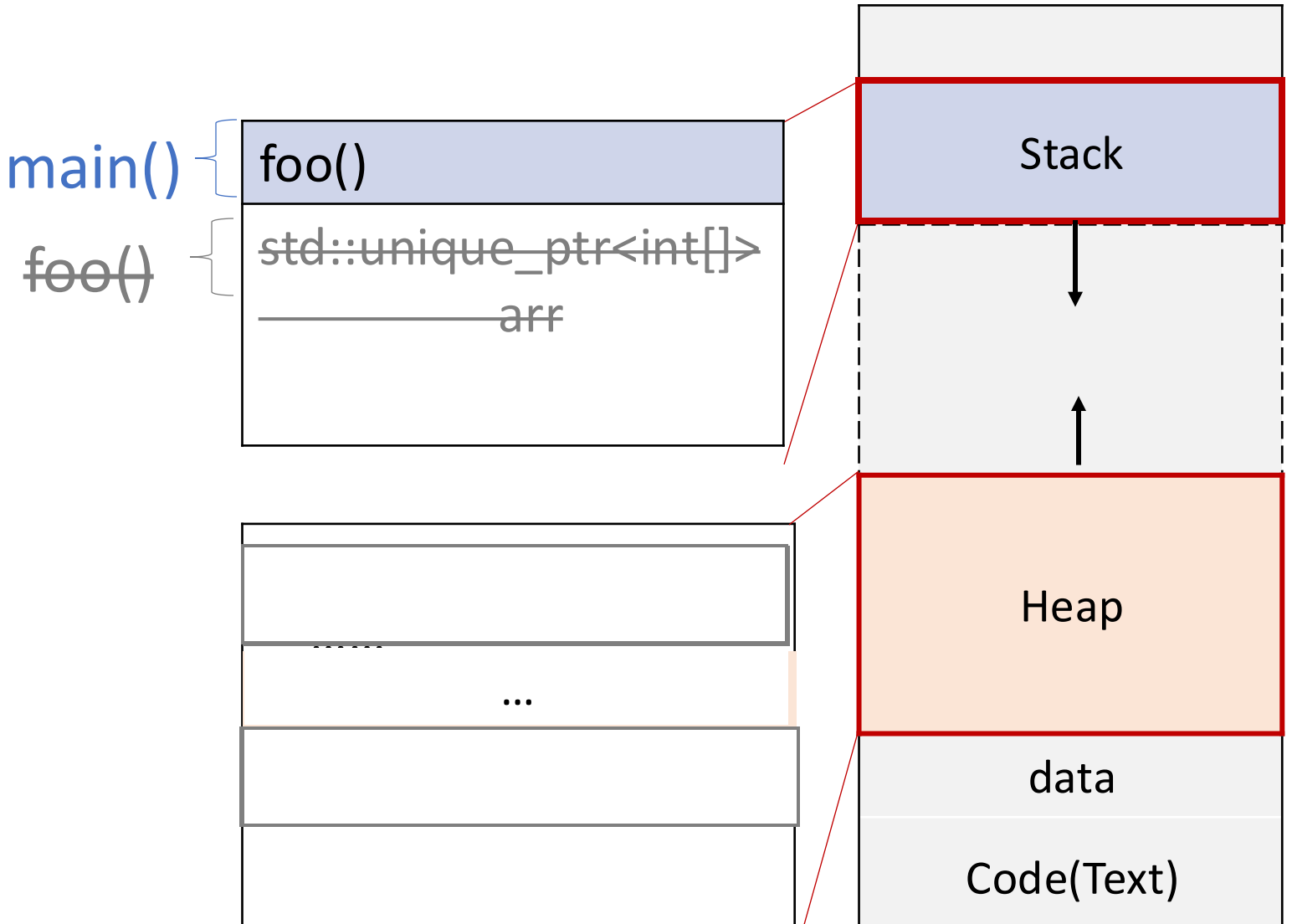
```
void foo(){
    std::unique_ptr<int[]> arr = std::make_unique<int[]>(10);
    arr[0] = 0;
}

int main(){
    foo();
    .......
}
```

std::unique_ptr is disposed:

delete[] arr;
is called



```
void foo(){
    std::unique_ptr<int[]> arr = std::make_unique<int[]>(10);
     arr[0] = 0;
}
int main(){
     foo();
     .......
}
```



smart pointer: unique_ptr

```
std::unique_ptr<Rectangle> rec = new Rectangle(10.0,12.0);
```

Unique_ptr needs to call the constructor explicitly

```
std::unique_ptr< Rectangle > default_rec(new Rectangle());
```

```
std::unique_ptr< Rectangle > explicit_rec = std::make_unique< Rectangle>();
```

```
std::unique_ptr< Rectangle > rec2 = explicit_rec ;
```


unique_ptr class doesn't allow copy of unique_ptr

```
std::unique_ptr< Rectangle > rec2 = std::move(explicit_rec );
```

std::move() : transferring of ownership(resources) from one object to another

smart pointer: shared_ptr

- a group of owners who are collectively responsible for the resource. The last of them to get destroyed will clean it up.

smart pointer: shared_ptr

demo

- std::shared_ptr: a smart pointer that retains shared ownership of an object through a pointer. Several shared_ptr objects may own the same object.
- The object is destroyed and its memory deallocated, when the last shared_ptr owning the object is destroyed or is assigned to another pointer. (when Reference counting==0)

```
std::shared_ptr<Rectangle> rec = std::make_shared<Rectangle>();
```

```
std::shared_ptr< Rectangle> rec2(new Rectangle());
```

```
std::shared_ptr< Rectangle> rec3 = rec2;
```

Where to find the resources?

- Memory Heap and Stack: https://courses.engr.illinois.edu/cs225/fa2022/resources/stack-heap/
- RAII: https://learn.microsoft.com/en-us/cpp/cpp/object-lifetime-and-resource-management-modern-cpp?view=msvc-170
- Move semantics: https://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html
- Passing arguments by reference: https://www.learncpp.com/cpp-tutorial/passing-arguments-by-reference/
- Effective C++: 55 specific ways to improve your programs and designs, Scott Meyers, 3rd edition
- A Tour of C++, Bjarne Stroustrup