

CS541 6 Recitation 2

C++ functions and memory management

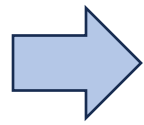
09/05/2024

Jamal Hashim

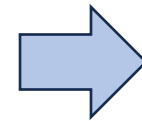
Overview

- C++ functions
 - Function parameters
 - Function return
- C++ memory management
- C++ smart pointer

Animation
(How it works?)



How to use it
correctly?



Code example

Functions

- Function parameters
- Function returns

C++ Function

A sequence of statements with a name and a list of parameters

Return type
(function derived type)

↓

```
int add(int a, int b){  
    return a+b;  
}
```

Function parameters



Function body



Ways to pass in function parameters

- Pass by value : passing a copy of the value
- Pass by pointer : passing the copy of the value's pointer
- Pass by reference : passing a reference

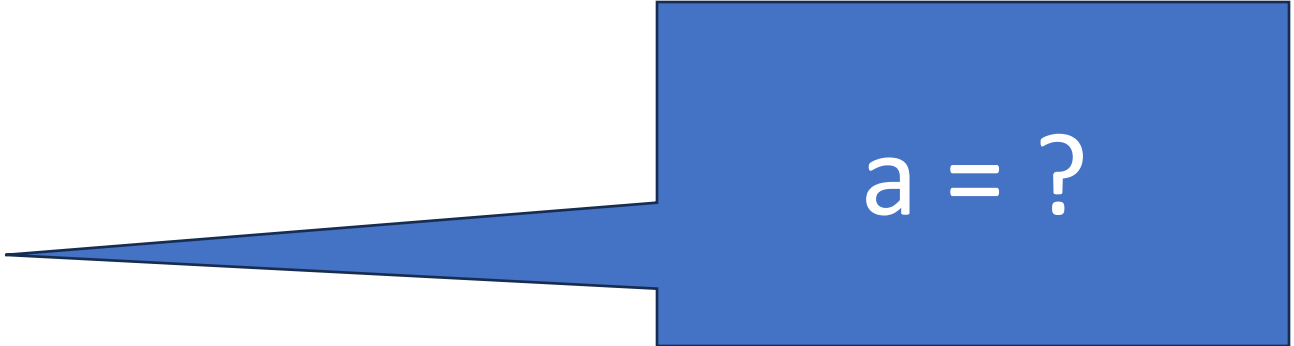
Function Parameter

--- Passing value

When a value is passed to a function, a **copy** of the value is created.

```
void increment(int value){  
    value ++;  
}
```

```
int main(){  
    int a = 0;  
    increment(a);  
    ...  
}
```



a = ?

Function Parameter

--- Passing value

When a value is passed to a function, a **copy** of the value is created.

```
void increment(int value){  
    value ++;  
}  
  
int main(){  
    int a = 0;  
    increment(a);  
    std::cout << a << std::endl;  
    // print 0  
}
```

1. changes inside the function are **NOT** reflected after the function call.

Function Parameter

--- Passing value

When a value is passed to a function, a **copy** of the value is created.

```
void print_str(std::string value){  
    std::cout << value << std::endl;  
}  
  
int main(){  
    std::string paragraph;  
    ... ..  
    print_str(a);  
}
```

2. Copying is time-consuming for large objects

Function Parameter

- Pass by value : p

What if I want to
change an external
variable?
Can I avoid copying of
parameters?

- Pass by pointer : passing the copy of the value's pointer

- Pass by reference : passing a reference



* Function Parameters

--- Passing pointer

Semantics:

providing the function with the **address** of the variable rather than its **value**.

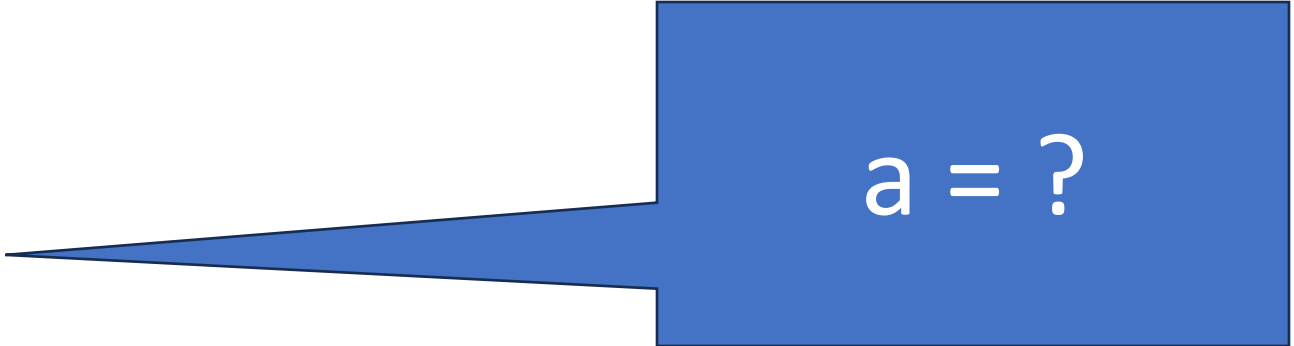
- Function can **modify** the original value through dereferencing
- **Direct access** to original variable
- **Memory efficiency**

* Function Parameter

--- Passing pointer

```
void increment(int* a){  
    (*a)++;  
}
```

```
int main(){  
    int a = 0;  
    increment(&a);  
    ...  
}
```



a = ?



Function Parameter

--- Passing reference

Semantics: allowing the function to operate directly on the original variable, rather than on a copy

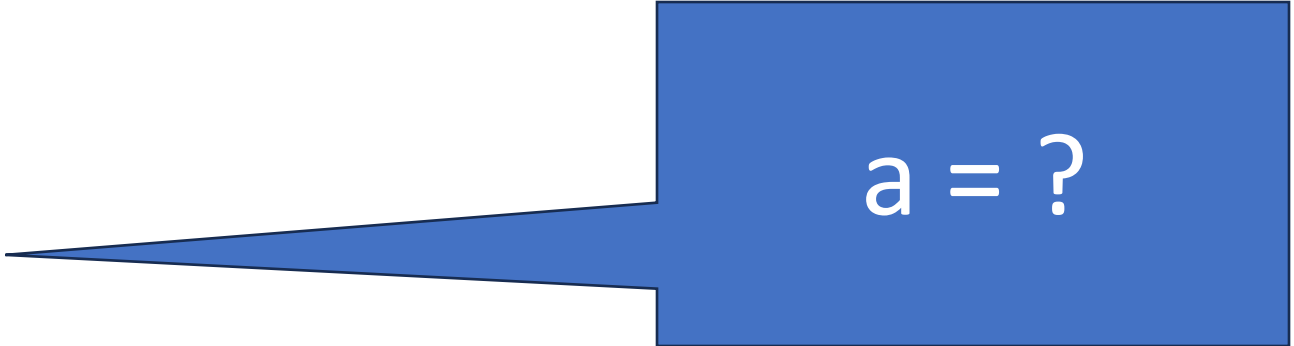
- Function can modify the argument
- Direct access to original variable
- No copy is made

& Function Parameter

--- Passing reference

```
void increment(int& a){  
    a++;  
}
```

```
int main(){  
    int a = 0;  
    increment(a);  
    ...  
}
```



a = ?

Poll

```
#include <iostream>
```

```
void func(int* x, int y) {  
    *x = 5;  
    y = y + *x;  
    x = &y;  
    *x = *x + 10;  
}
```

```
int main() {  
    int x = 0;  
    int y = 10;  
    func(&x, y);  
    std::cout << x << std::endl;  
    return 0;  
}
```

What will this program print?



Poll

```
#include <iostream>
```

```
void func(int* x, int y) {
```

```
    *x = 5;
```

```
    y = y + *x;
```

```
    x = &y; //the pointer variable itself is passed by copy
```

```
    *x = *x + 10;
```

```
}
```

```
int main() {
```

```
    int x = 0;
```

```
    int y = 10;
```

```
    func(&x, y);
```

```
    std::cout << x << std::endl;
```

```
    return 0;
```

```
}
```

Functions

- Function parameters
- Function returns

Function Returns

--- value

- Return by value : returning a copy of the value

```
int value( int a ) {  
    int b = a * a;  
    return b;    // return a copy of b  
}
```

Note: Return by value could avoid copying under compiler's C++ Return Value Optimization (RVO)

* | Function Returns

--- pointer

Why return pointers?

- Allow direct access to memory



Function Returns

--- pointer

Correct ways of returning a pointer

- Returning a pointer to a global or static variable
- Returning a pointer to a non-local array element
- Returning a pointer from a class member function
- Returning a pointer to memory on heap



Function Returns

--- pointer

Incorrect way of returning a pointer

- return a pointer to a **local variable**

*

Function Returns

--- pointer

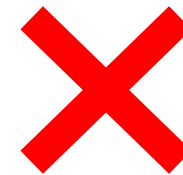
What can go wrong?

Dangling pointers

```
int* dangerousFunc() {  
    int localVar = 100;  
    return &localVar;  
}
```

```
int main() {  
    int* res = dangerousFunc();  
    std::cout << *res << std::endl;  
}
```

Undefined behavior!



*

Function Returns

--- pointer

```
int* dangerousFunc(int* localVar) {  
    *localVar = 100;  
    return localVar;  
}
```

```
int main() {  
    int localVar = 0  
    int* res = dangerousFunc(&localVar);  
    std::cout << *res << std::endl;  
}
```



C++ Memory Management



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

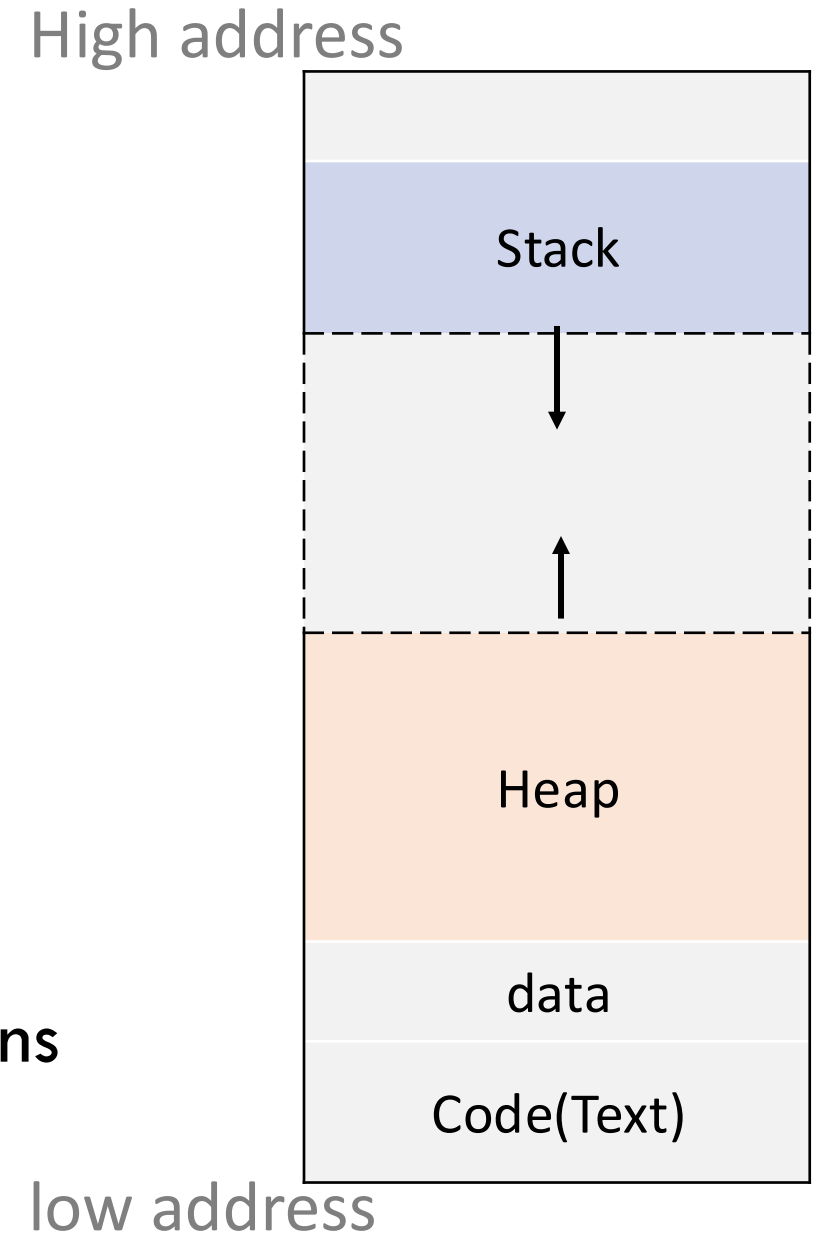


Stack memory



Memory

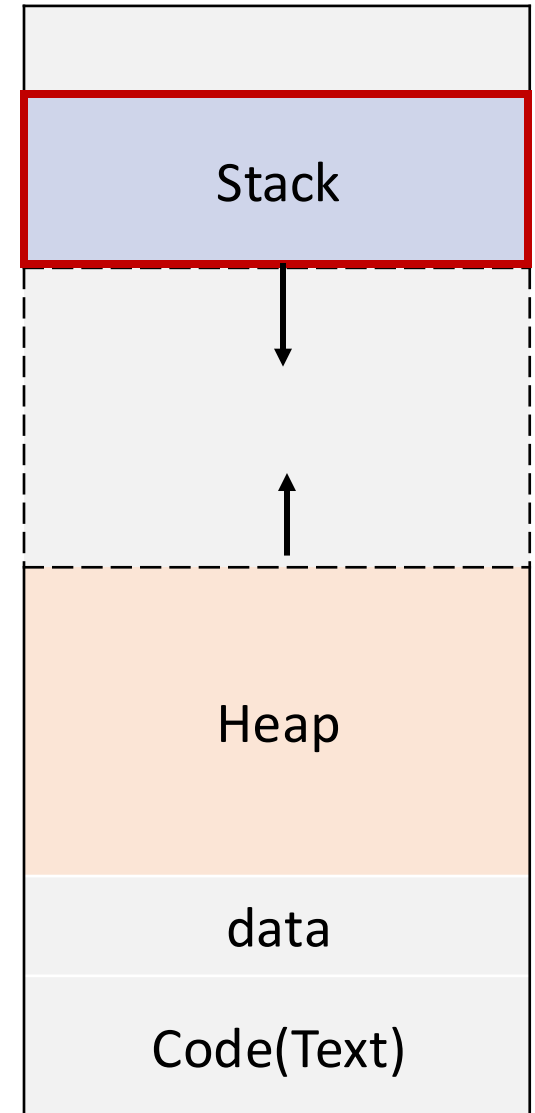
- **Stack:** used for memory needed to call methods(such as local variables), or for inline variables
- **Heap:** Dynamically memory used for programmers to allocate. The memory will often be used for longer period than stack
- **Data:** use for constants and initialized global objects
- **Code:** segments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

High address

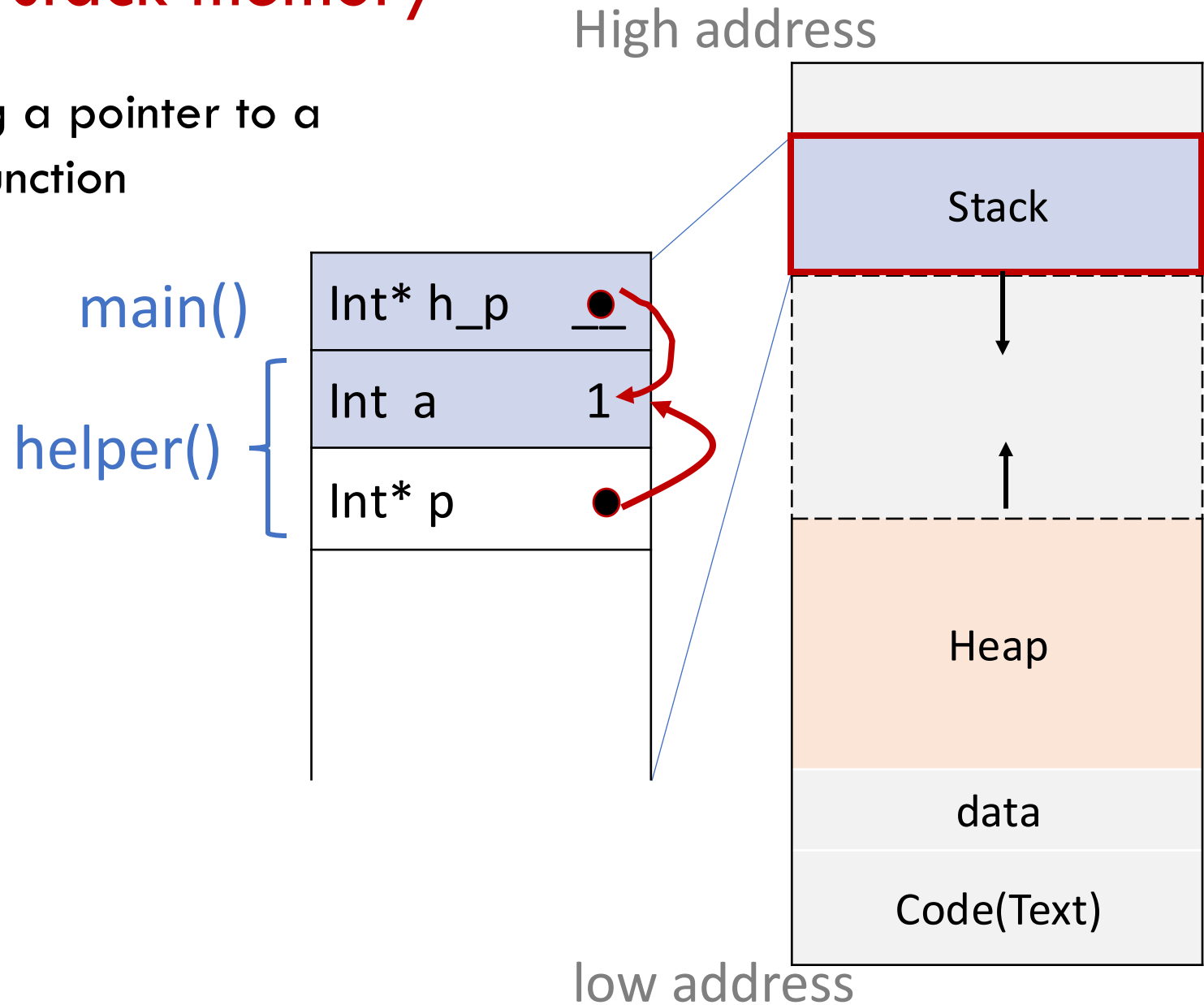


low address

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

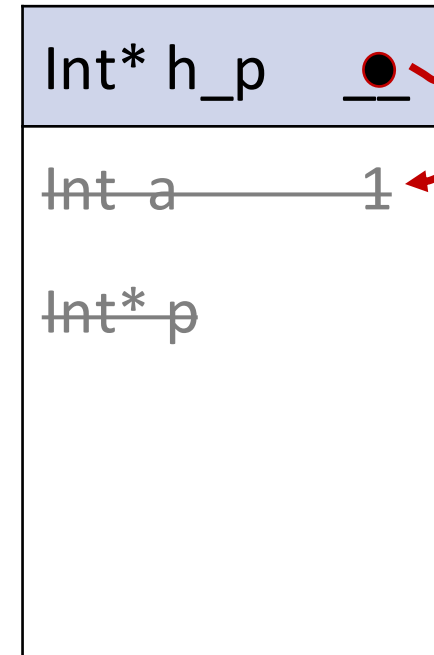
```
int* helper()
{
    int a = 3;
    int * p = &a;
    return p;
}
```

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

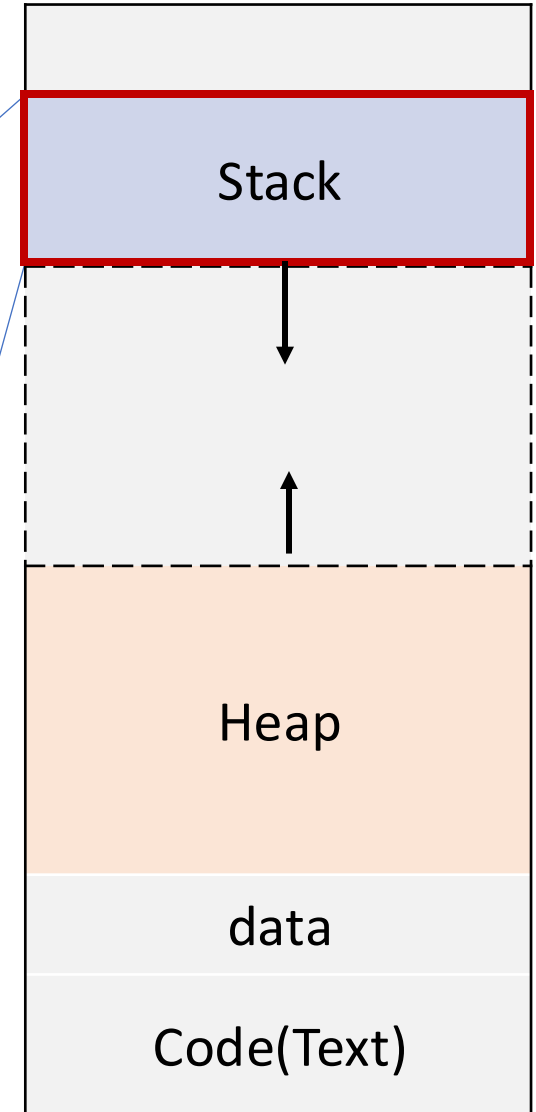
...

```
}
```

main()



High address



low address

Undefined behavior

Common mistake with stack memory

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

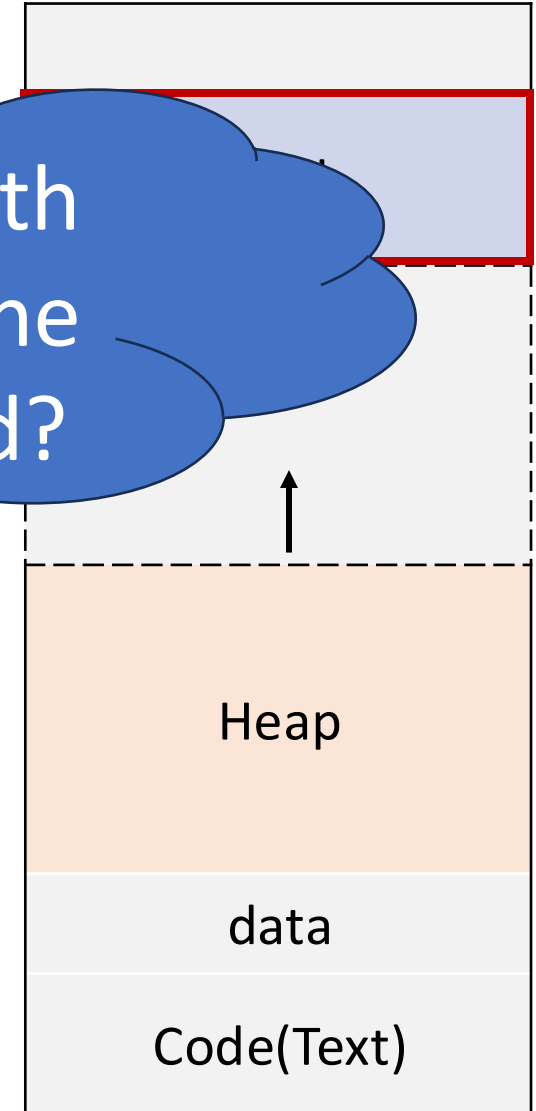
```
int* helper()  
{  
    int a = 3;  
    int * p = &a;  
    return p  
}  
int main(){  
    int* h_p  
    ...  
}
```



Can I have object with lifetime exceeding the scope it was created?

High address

low address



Heap memory

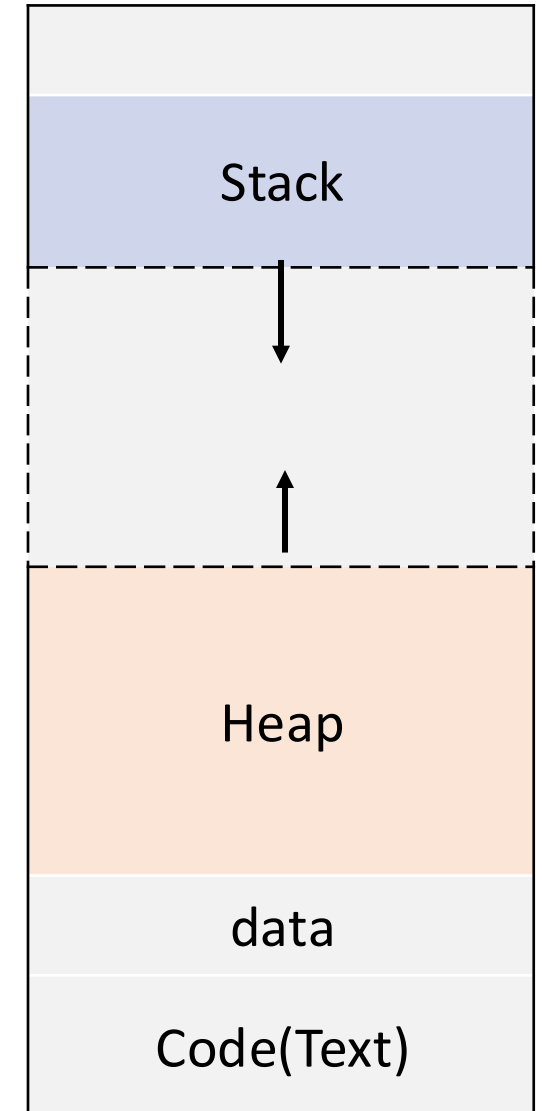
new
delete



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

High address



low address

Heap Memory new expression

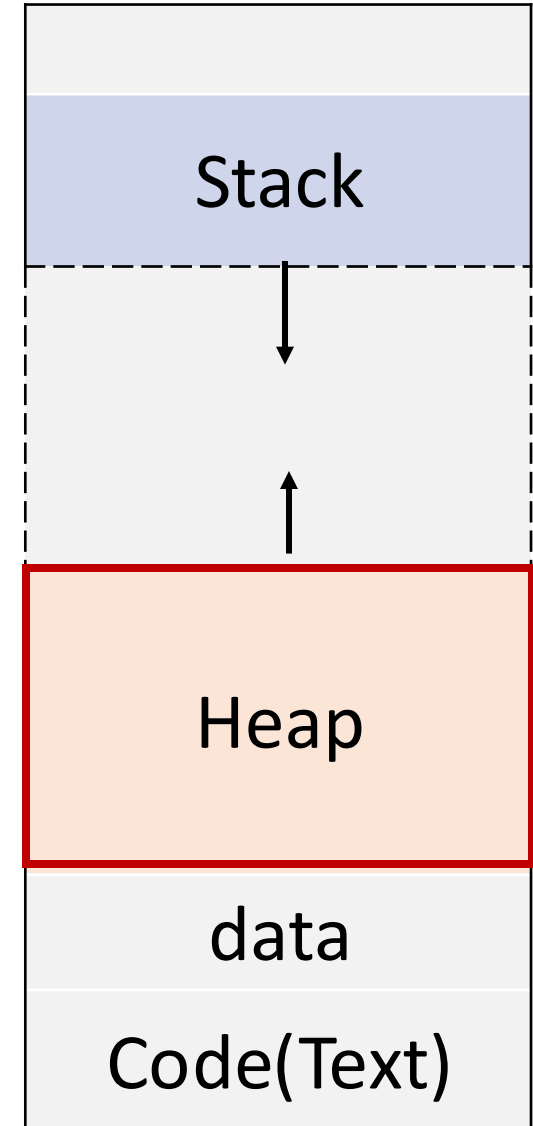
- **new** expression: create and initialize objects on heap (dynamic storage duration)

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

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

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

High address



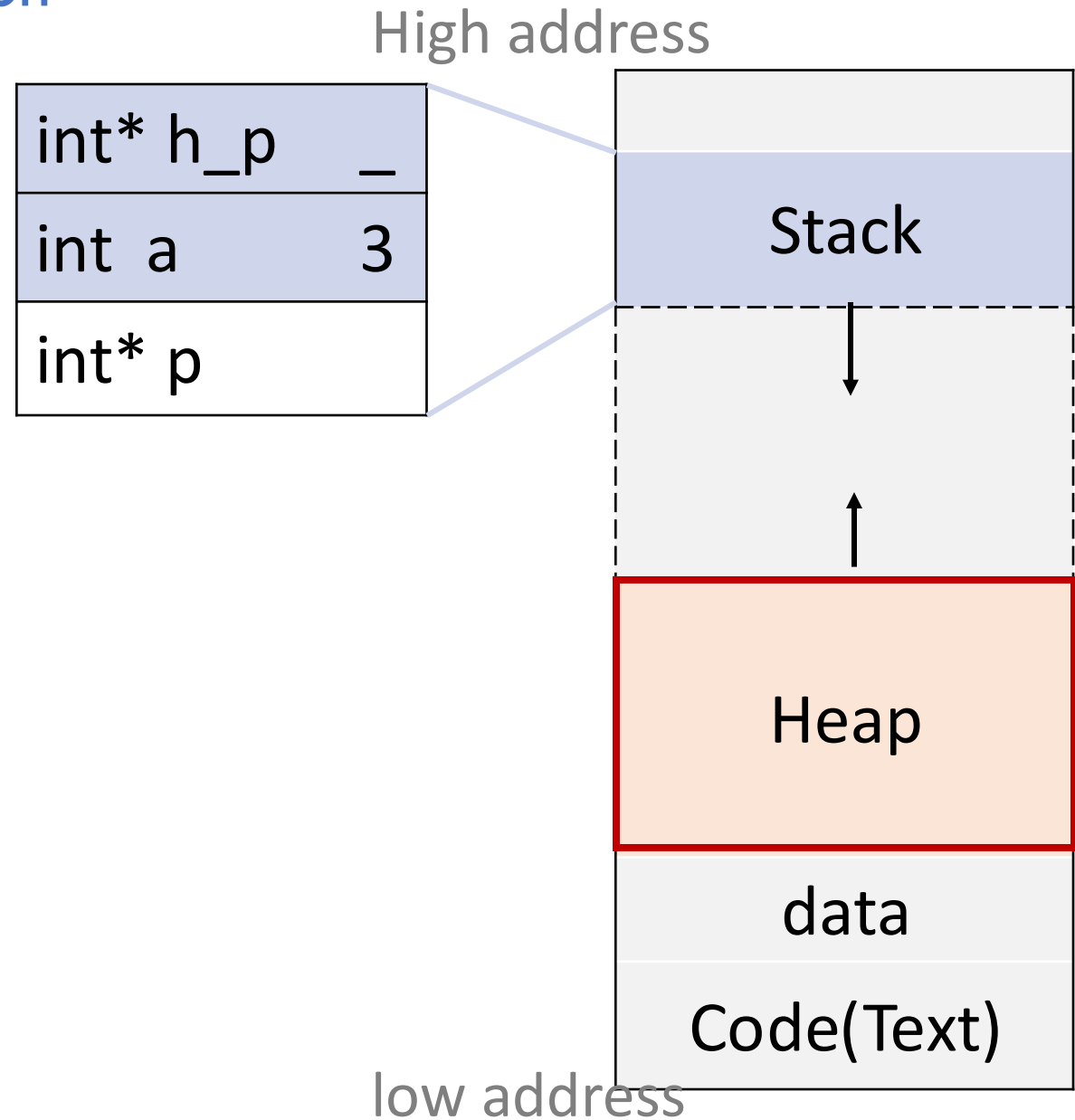
low address

Heap Memory

new expression

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

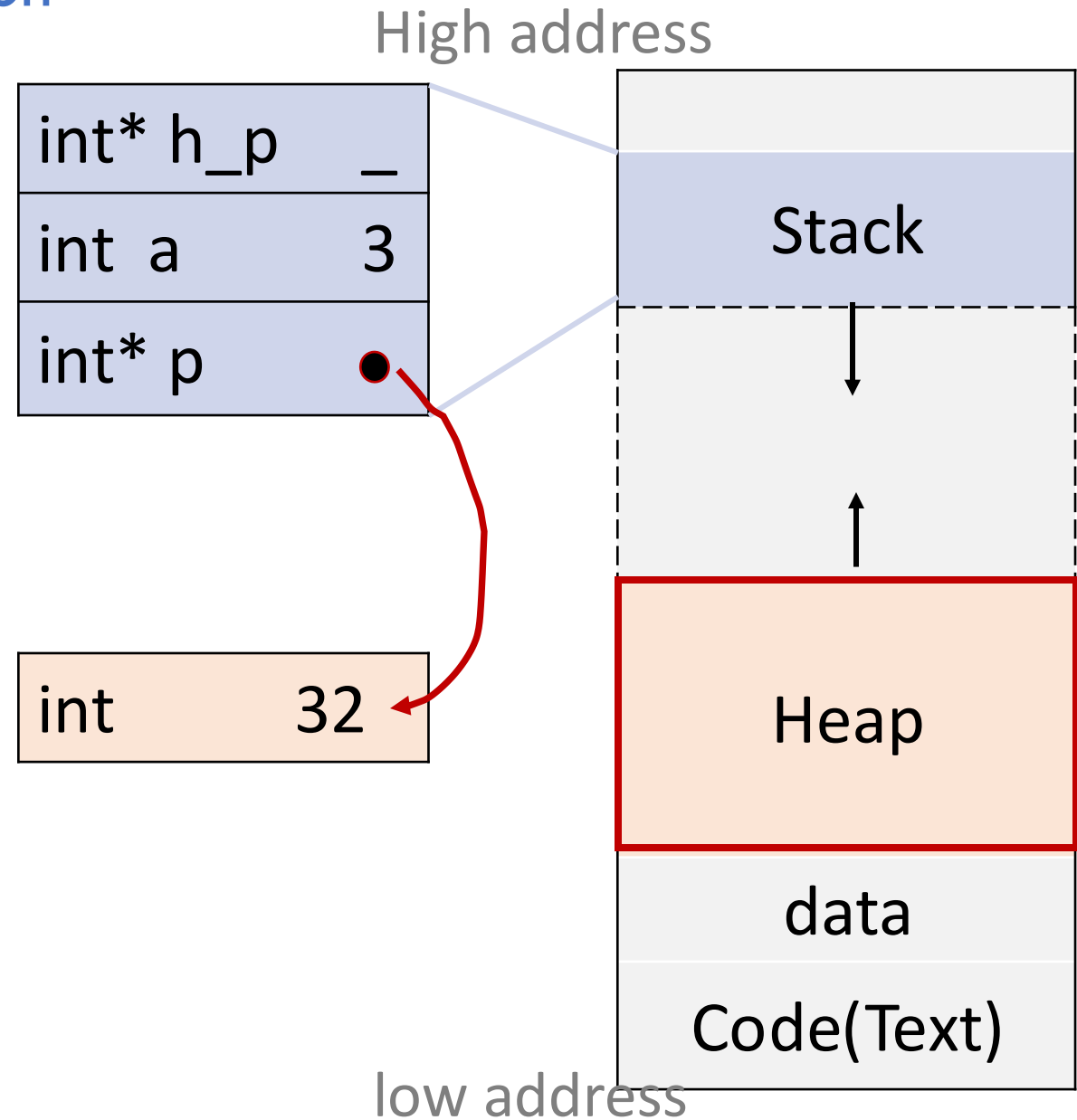
```
int main(){  
    int* h_p = helper();  
    ...  
}
```



Heap Memory

new expression

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

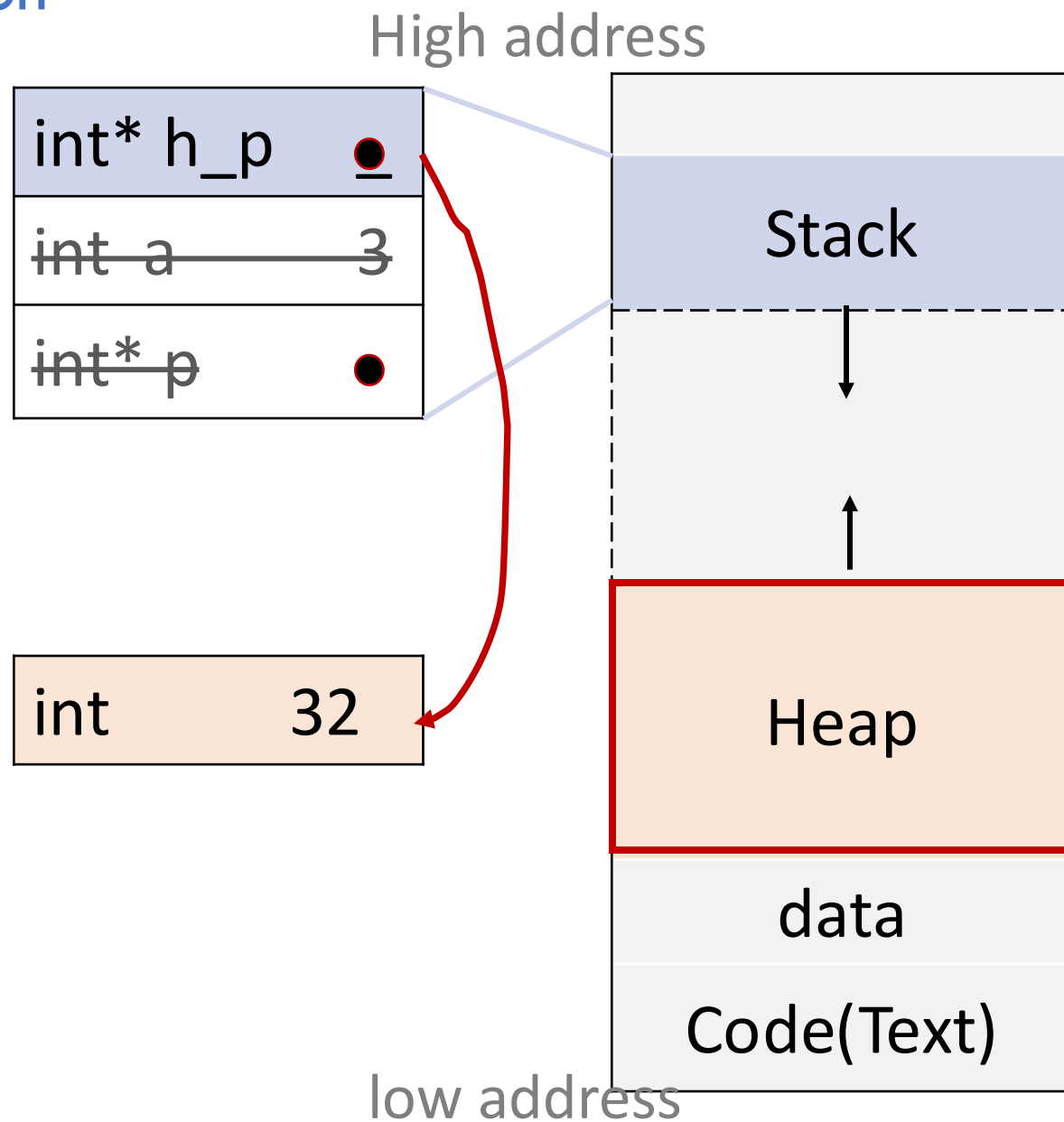


Heap Memory

new expression

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

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



Heap Memory

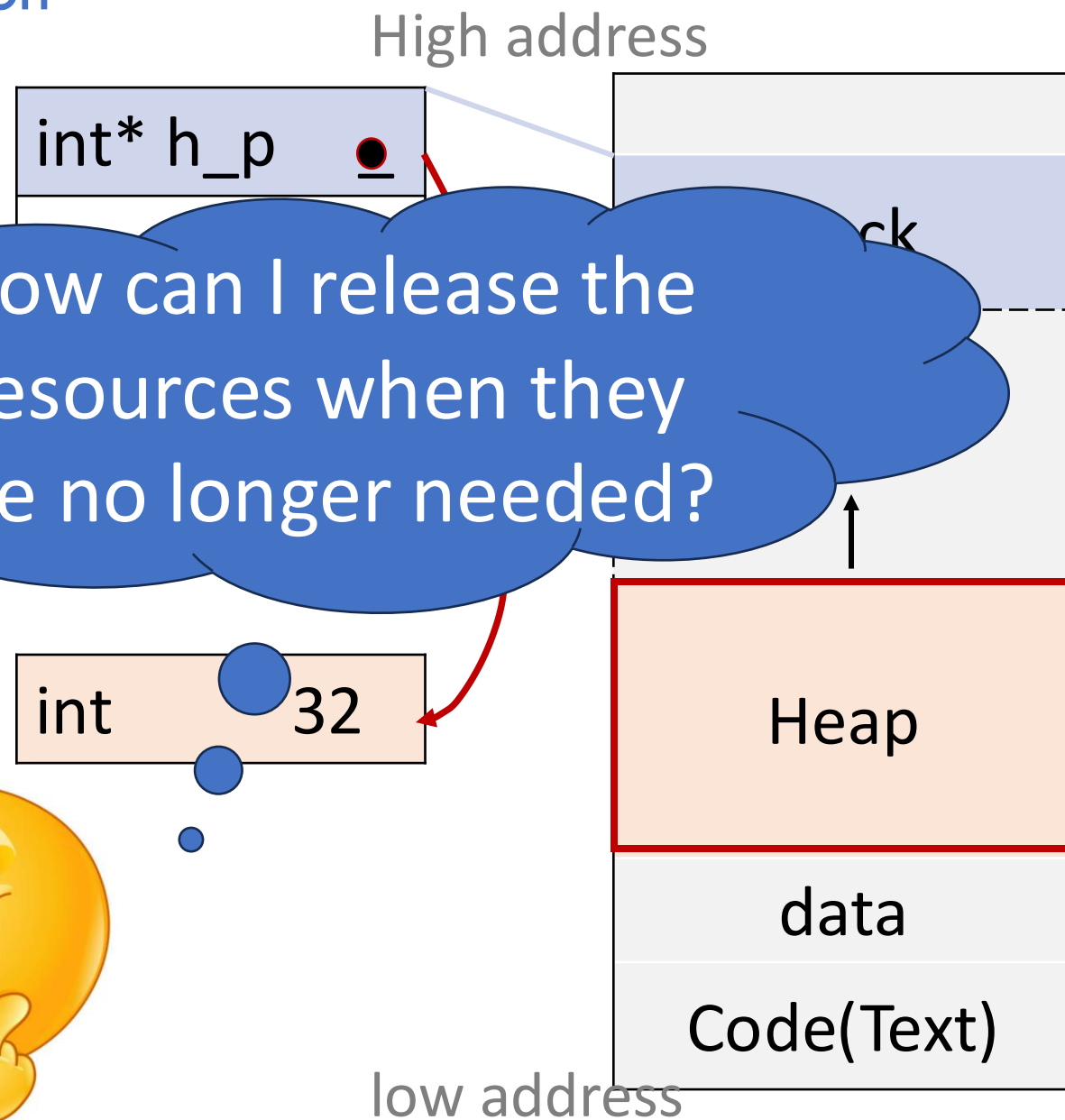
new expression

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

```
int main(){
    int* h_p = helper(
        ...
    }
}
```

NO automatic de-allocation with scope.

How can I release the resources when they are no longer needed?



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;
```

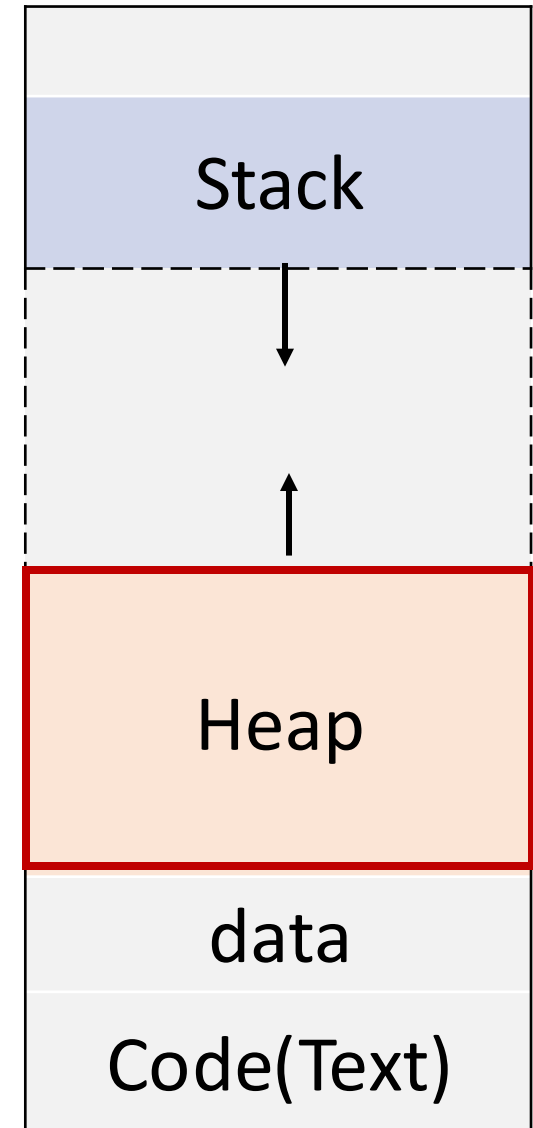
```
double* arr_p = new double[]{1, 2, 3};
```

```
delete[] arr_p;
```

```
T* p = new T(arg0, arg1, arg2,...);
```

```
delete obj_p;
```

High address



low address

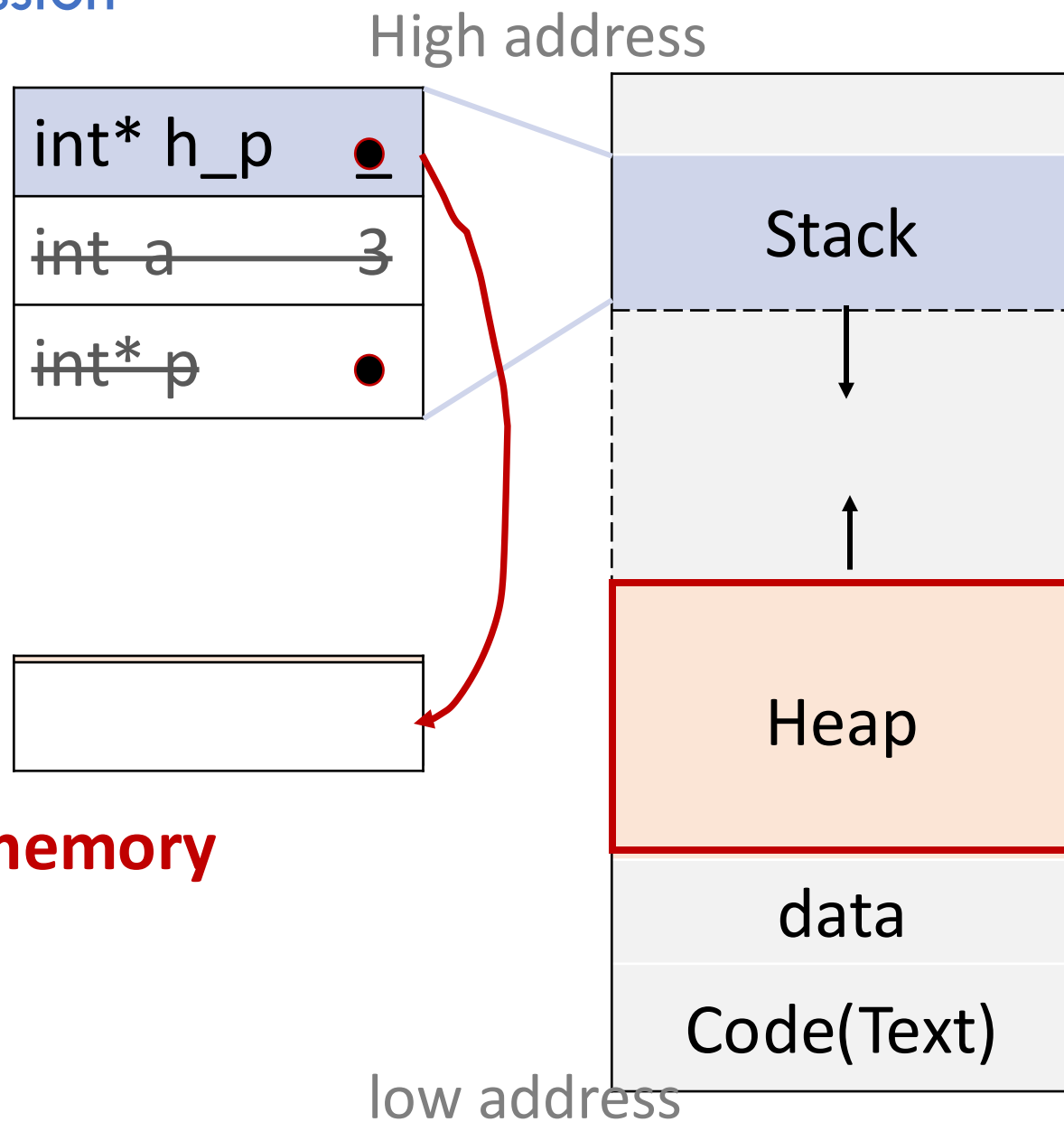
Heap Memory

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

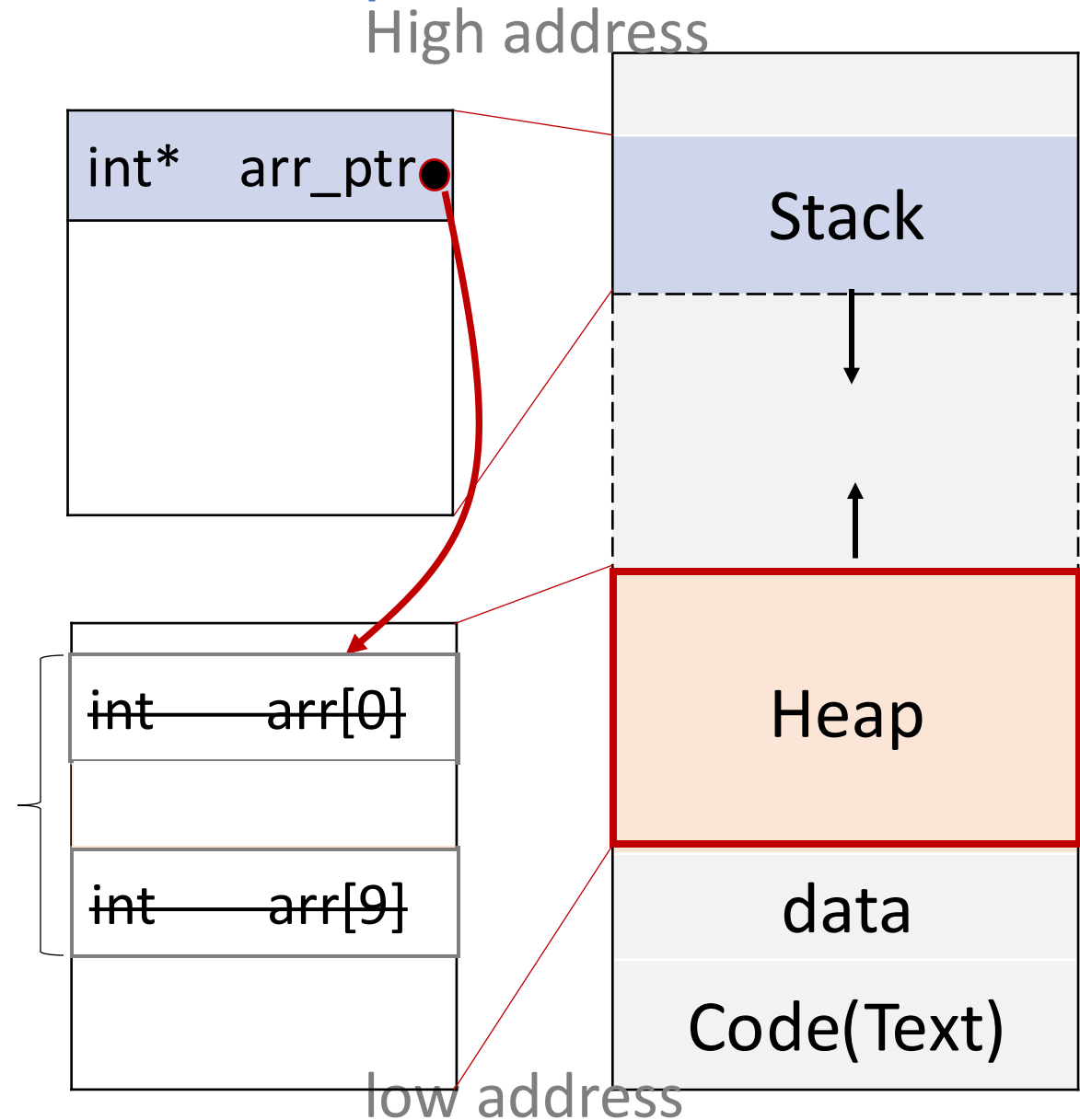
delete expression for array

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

```
delete arr_ptr;
```

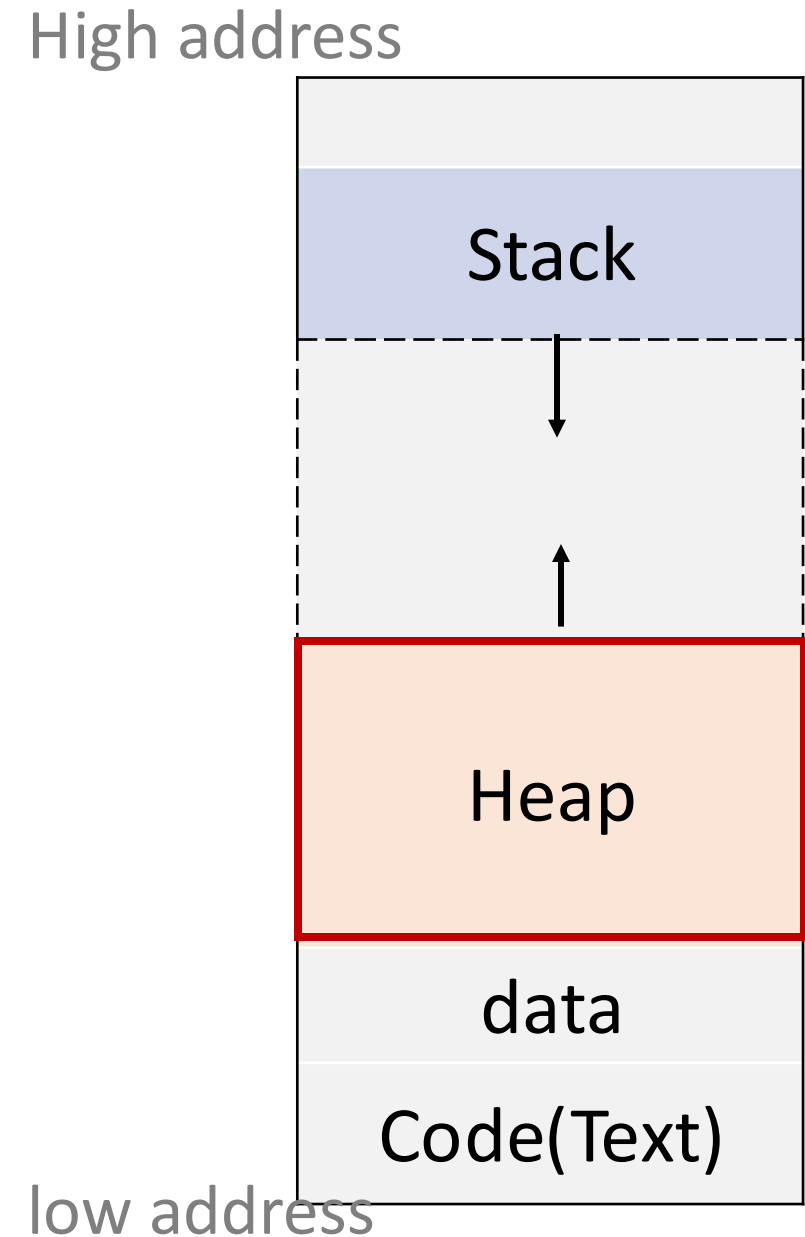


```
delete[] arr_ptr;
```



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?

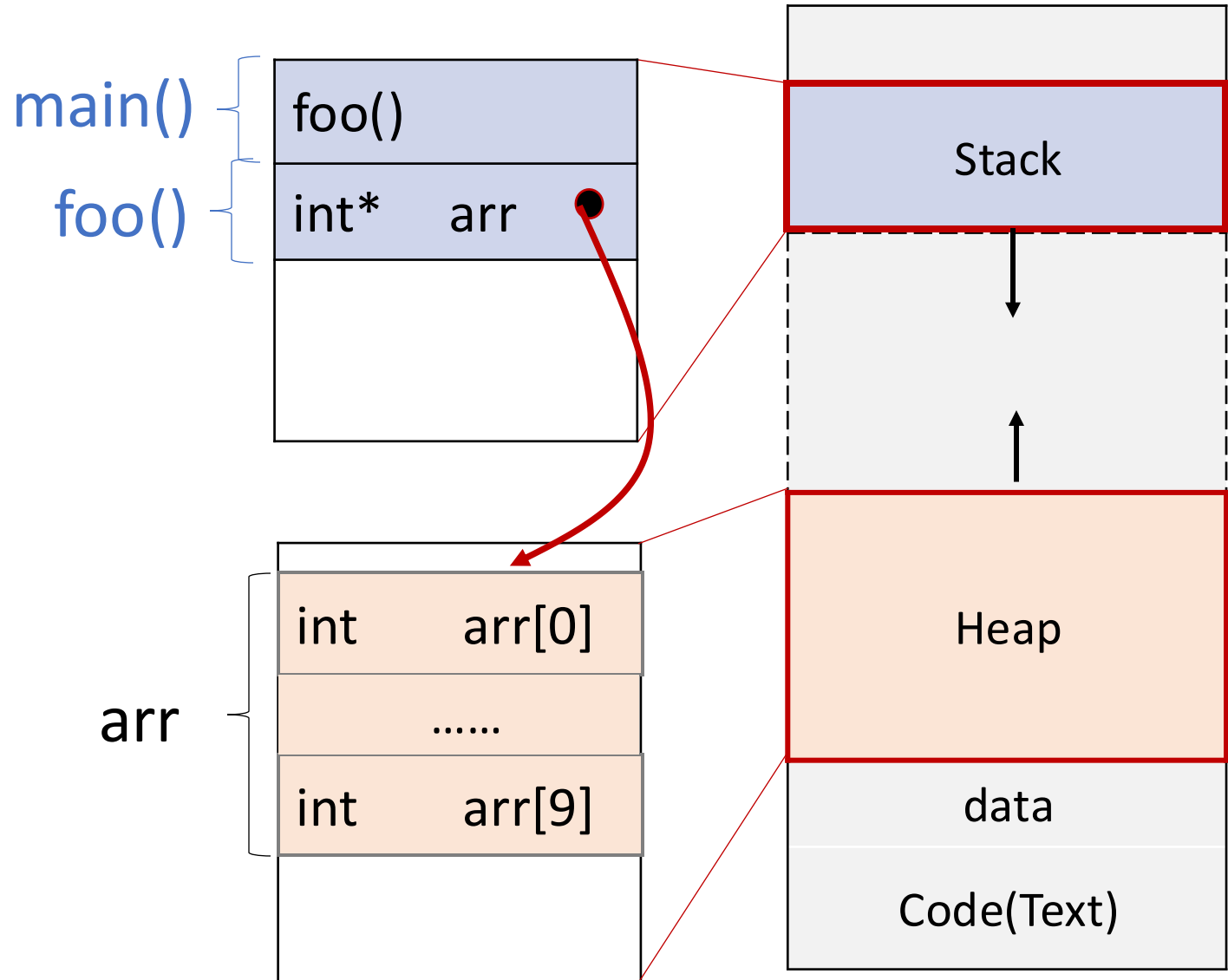


heap-based memory allocation

→

```
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();  
    .....  
}
```

main()

~~foo()~~

foo()

int* arr

Stack

foo() function's Stack automatically gets popped off when out of scope



Heap

int arr[0]

.....

int arr[9]

data

Code(Text)

heap-based memory allocation

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

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

main()

~~foo()~~

foo()

int* arr

Stack

foo() function's Stack automatically gets popped off when out of scope

Heap

data

Code(Text)

Does this function look correct?

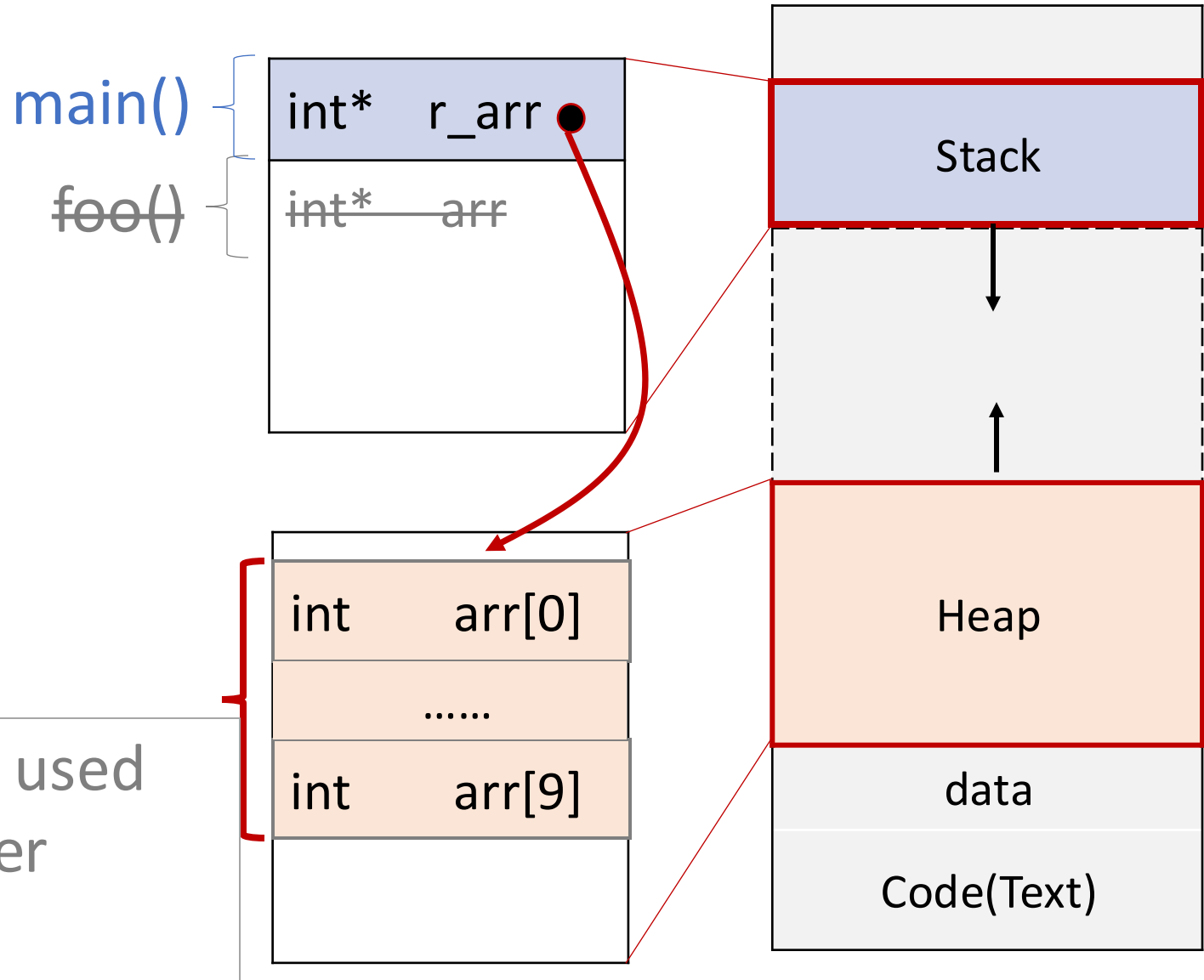
No. It doesn't release the memory of arr when function finishes.

heap-based memory allocation

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

➔

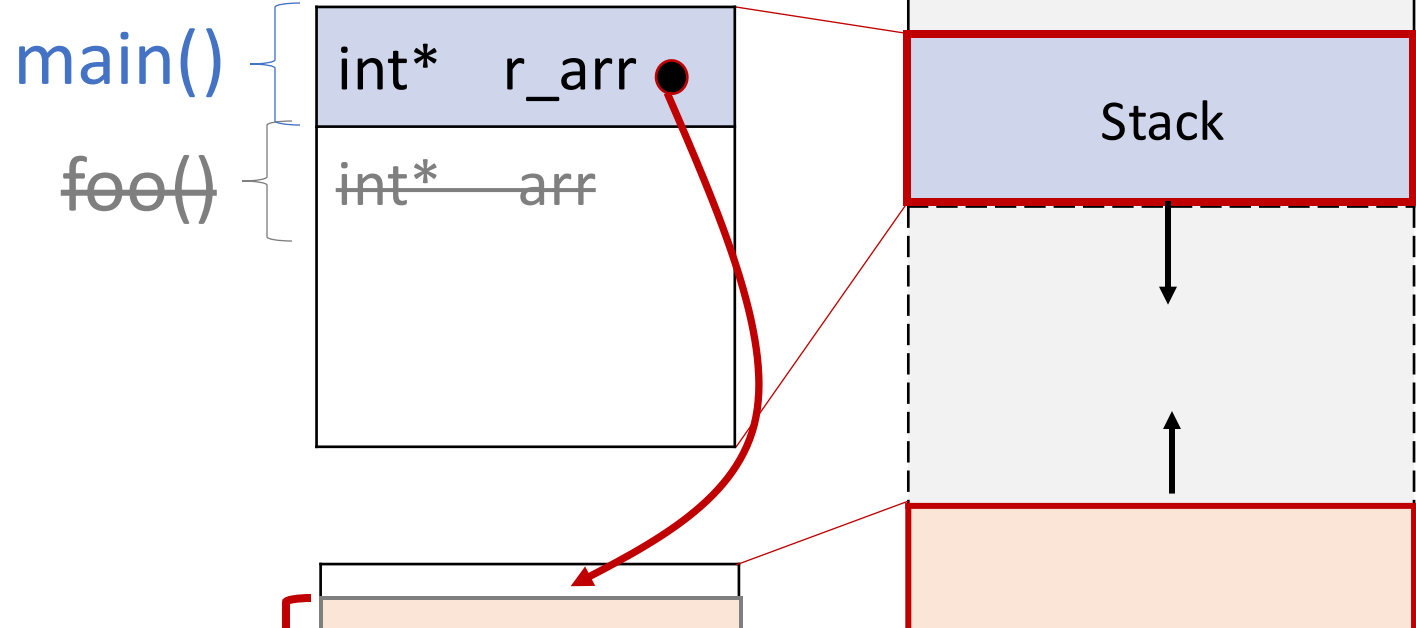
```
int main(){  
    int* r_arr = foo();  
    ..... // r_arr is neither used  
           nor deleted in later  
           program  
}
```



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.

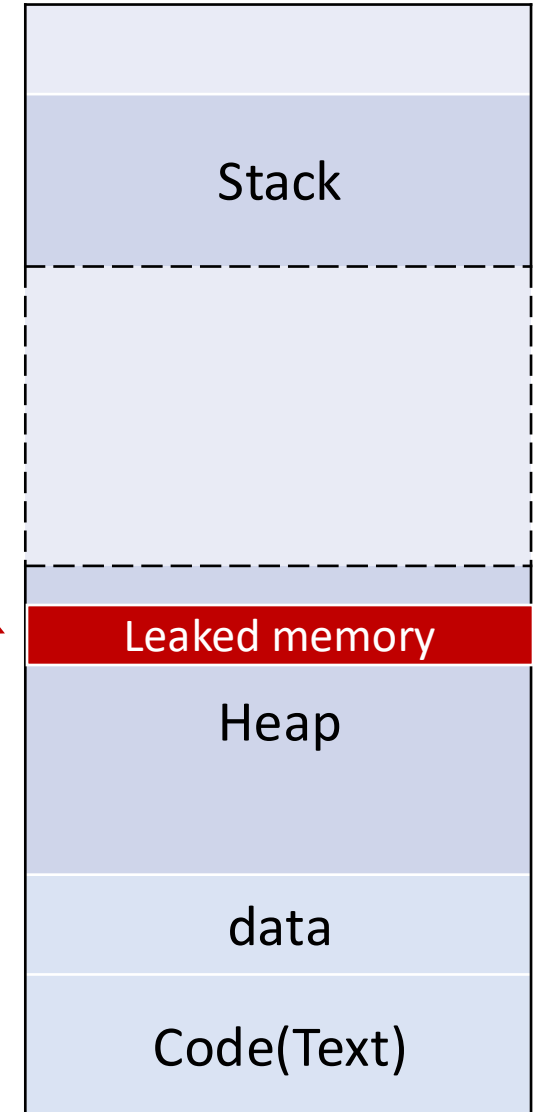
What is memory leak in C++?

- Memory leakage in C++ is when programmers allocate 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;  
}
```

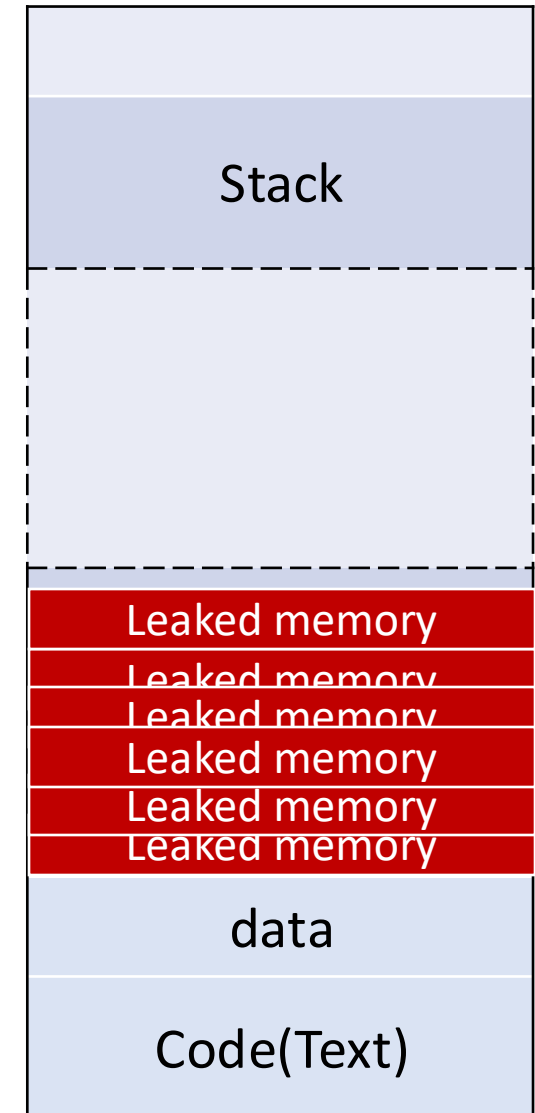
```
int main(){  
    int* r_arr = foo();  
    .....  
}
```



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

- 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
```

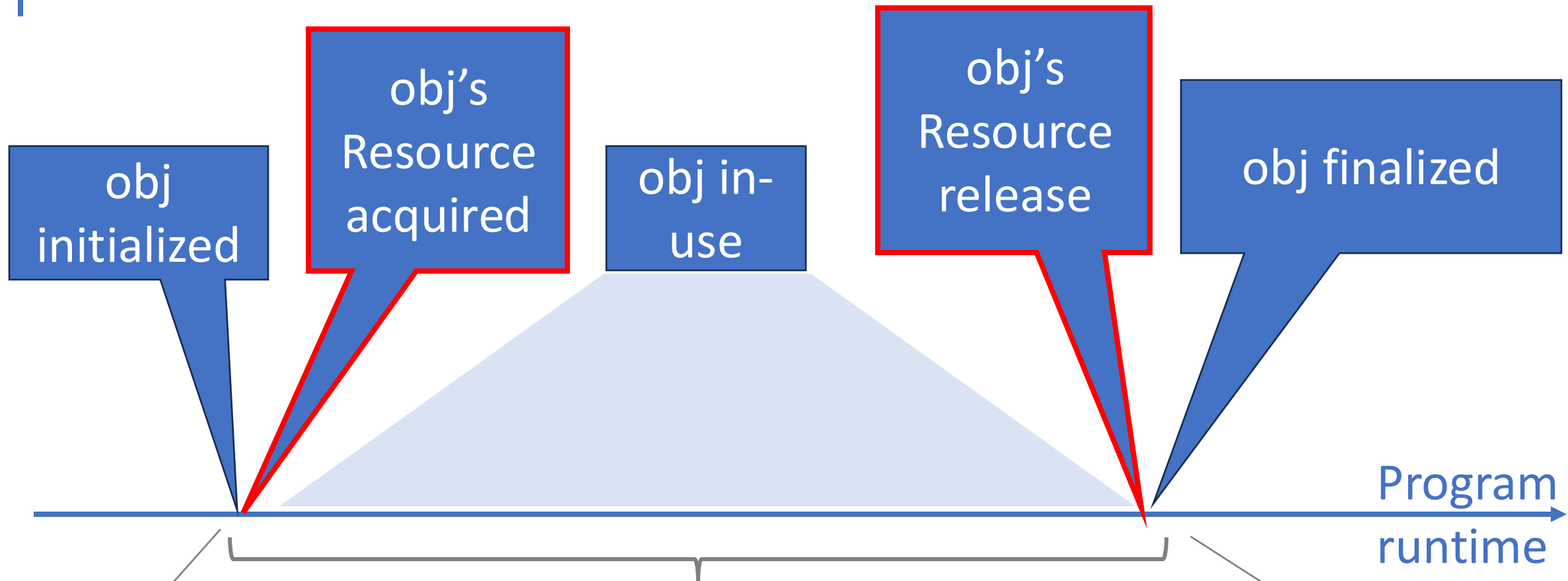
Memory Leak

- What is memory leak in C++?
- How to check if my program has memory leak?
- How to avoid memory leaks in my program?
 - Follow **RAll principles** (Resource acquisition is initialization)
 - Use **smart pointers and standard containers** whenever possible instead of using raw pointers and new/delete

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.

Example.



Begin:

- storage for its type is obtained
- initialization is complete

An object, obj's lifetime

end:

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

RAII Principle - Vector

- Even though `std::vector`'s data is on the heap, as the programmer we don't have to deal with `new` and `delete`
- When a vector is constructed, the constructor allocates space for its elements on the heap
- When a vector goes out of scope, its destructor is called and releases any underlying elements
- This prevents memory leaks and makes it simpler and safer than dealing with raw pointers

```
void foo() {  
    std::vector<int> nums = {1, 2, 3, 4};  
    // use nums safely  
} // nums goes out of scope, memory freed automatically
```

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

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

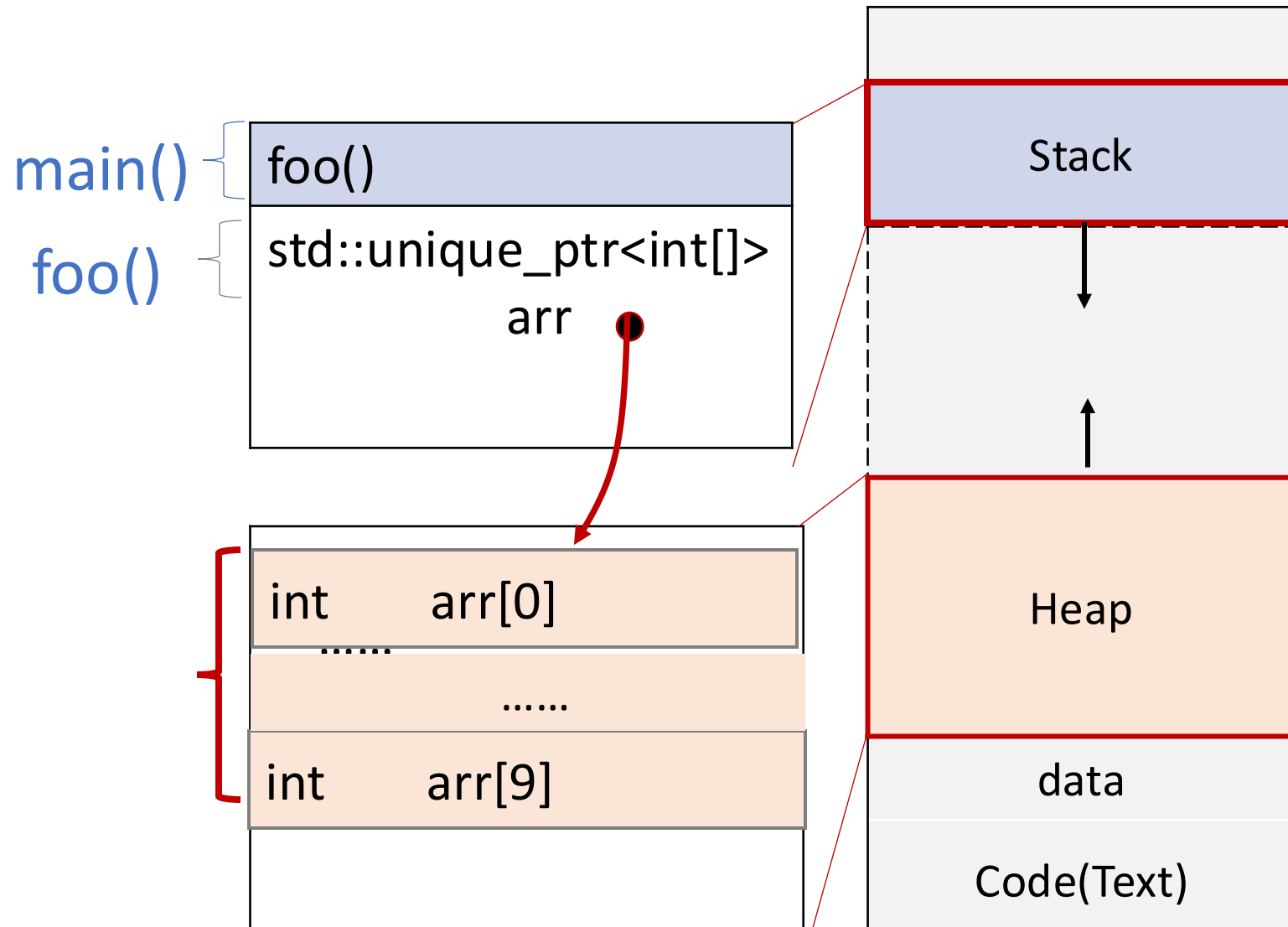
```
template <
    class T,
    class Deleter //optional
> class unique_ptr<T[], Deleter>;
```

```

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

foo()

std::unique_ptr<int[]>
arr

Stack



Heap

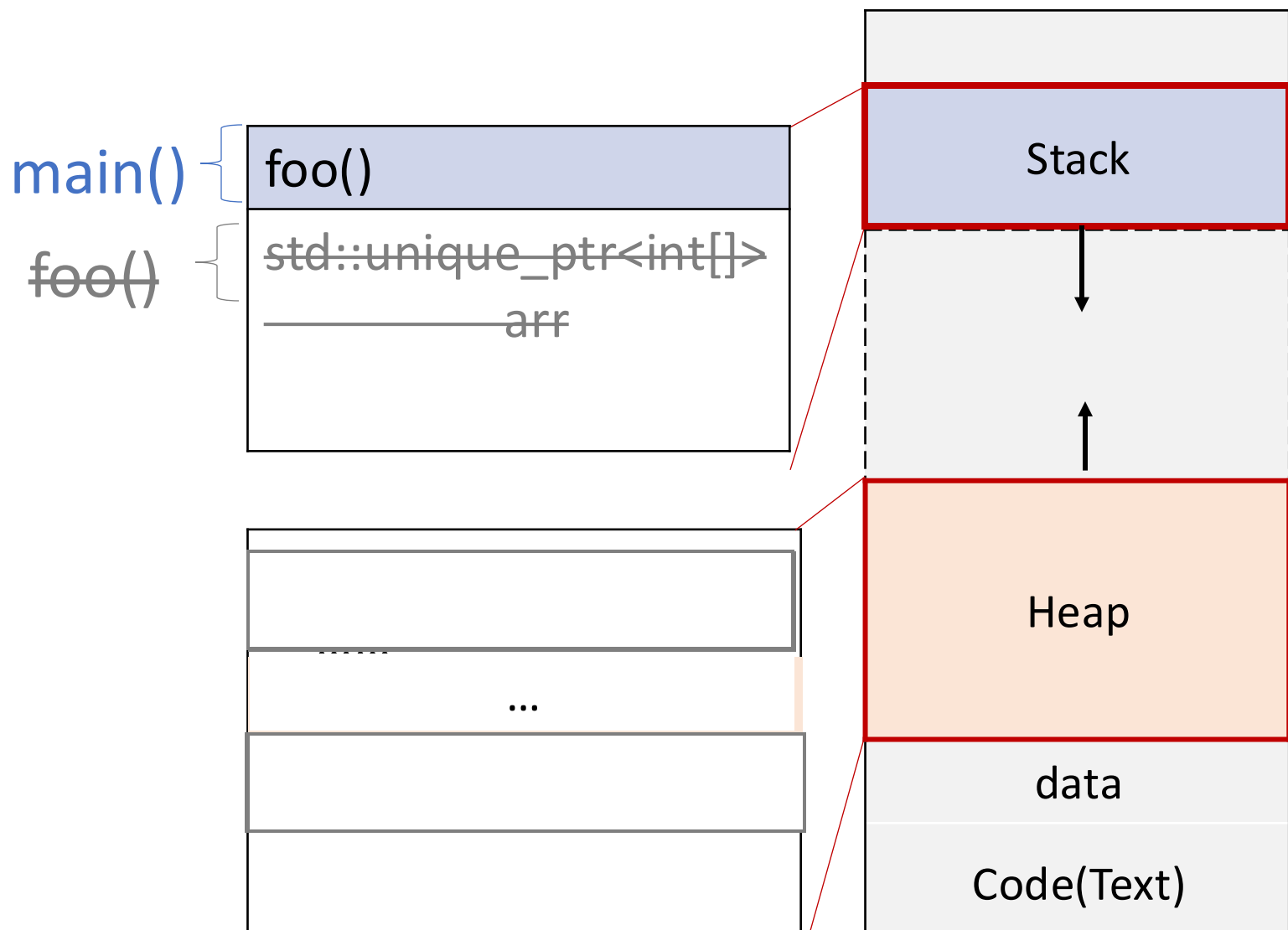
data

Code(Text)



```
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<int> a = new int(42);
```



Unique_ptr needs to call the constructor explicitly

```
std::unique_ptr<int> b(new int(42));
```



```
std::unique_ptr< int > explicit_ptr = std::make_unique< int>(42);
```



```
std::unique_ptr< int > ptr2 = explicit_ptr ;
```



unique_ptr class doesn't allow copy of unique_ptr

```
std::unique_ptr< int > ptr3 = std::move(explicit_ptr );
```



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

- `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;
```



Polls



Polls

Question:

For a function that just reads an int parameter, which way is typically *most efficient*?

- A. `void f(int x)` — pass by value
- B. `void f(int& x)` — pass by non-const reference
- C. `void f(int* x)` — pass by pointer
- D. B and C are equally efficient

Polls

Question:

Why is this code bad practice?

```
auto v = new std::vector<int>(100);
```

Where to find the resources?

- Memory Heap and Stack:
<https://courses.engr.illinois.edu/cs225/fa2022/resources/stack-heap/>
- RAll: <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