CS4414 Recitation 4

C++ smart pointer and container

09/2024

Alicia Yang

What is C++?

A federation of related languages, with four primary sublanguages

- **C**: C++ is based on C, while offering approaches superior to C. Blocks, statements, processor, built-in data types, arrays, pointers, etc., all come from C
- **Object-Oriented C++:** "C with Classes", classes including constructor, destructors, inheritance, virtual functions, etc.
- **Template C++:** generic programming language. Gives a template, define rules and pattern of computation, to be used across different classed.
- **STL(standard template library):** a special template library with conventions regarding containers, iterators, algorithms, and function objects

Overview

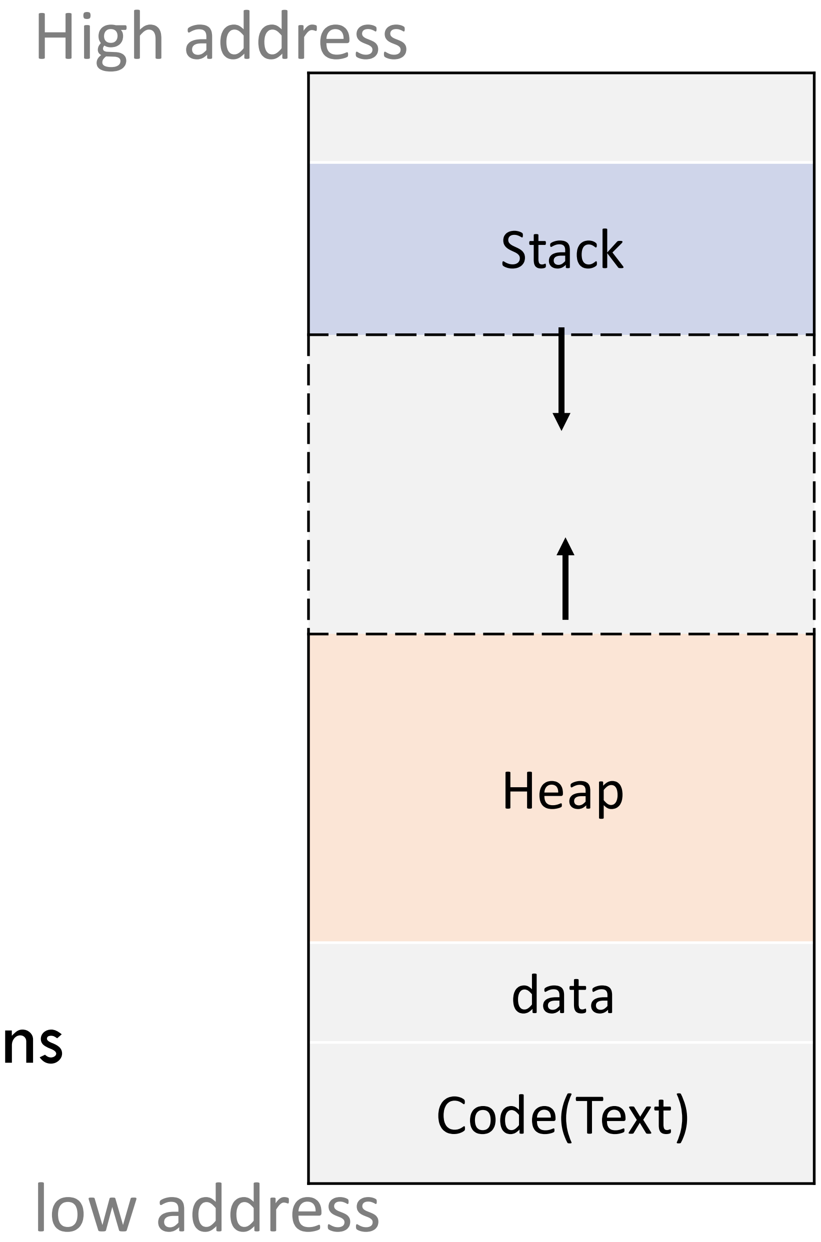
- C++ memory
  - Smart pointer
  - Applications at function and class
- Container

C++ Memory

- review
- Smart pointer
- Applications at function and classes

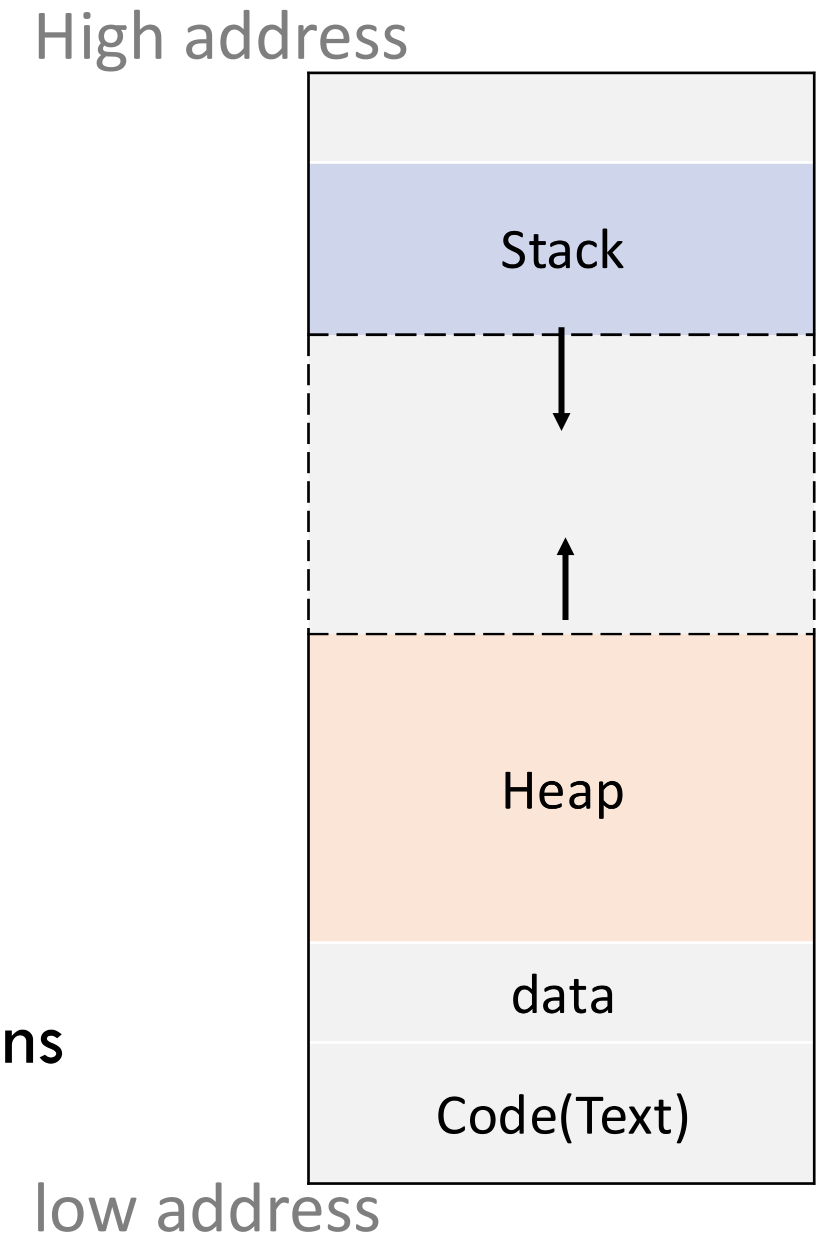
Memory

- **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



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

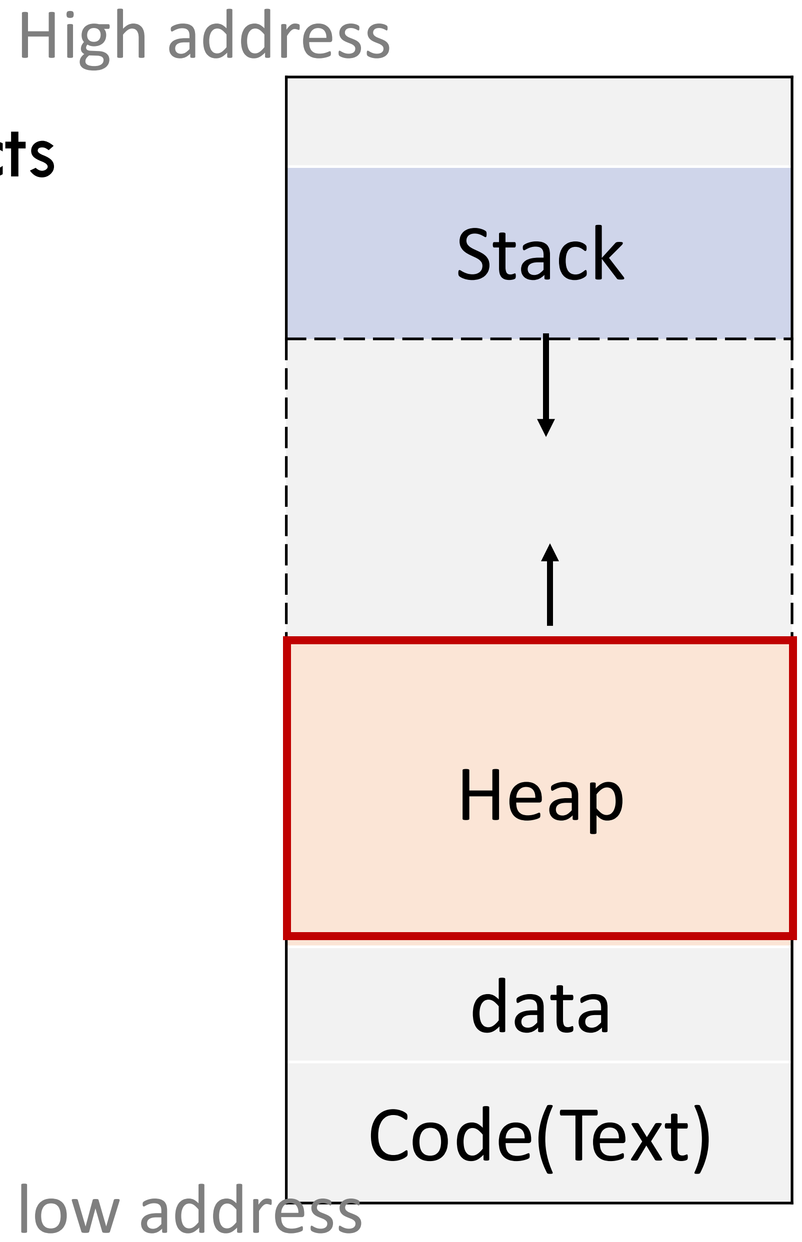


Heap Memory

new expression

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

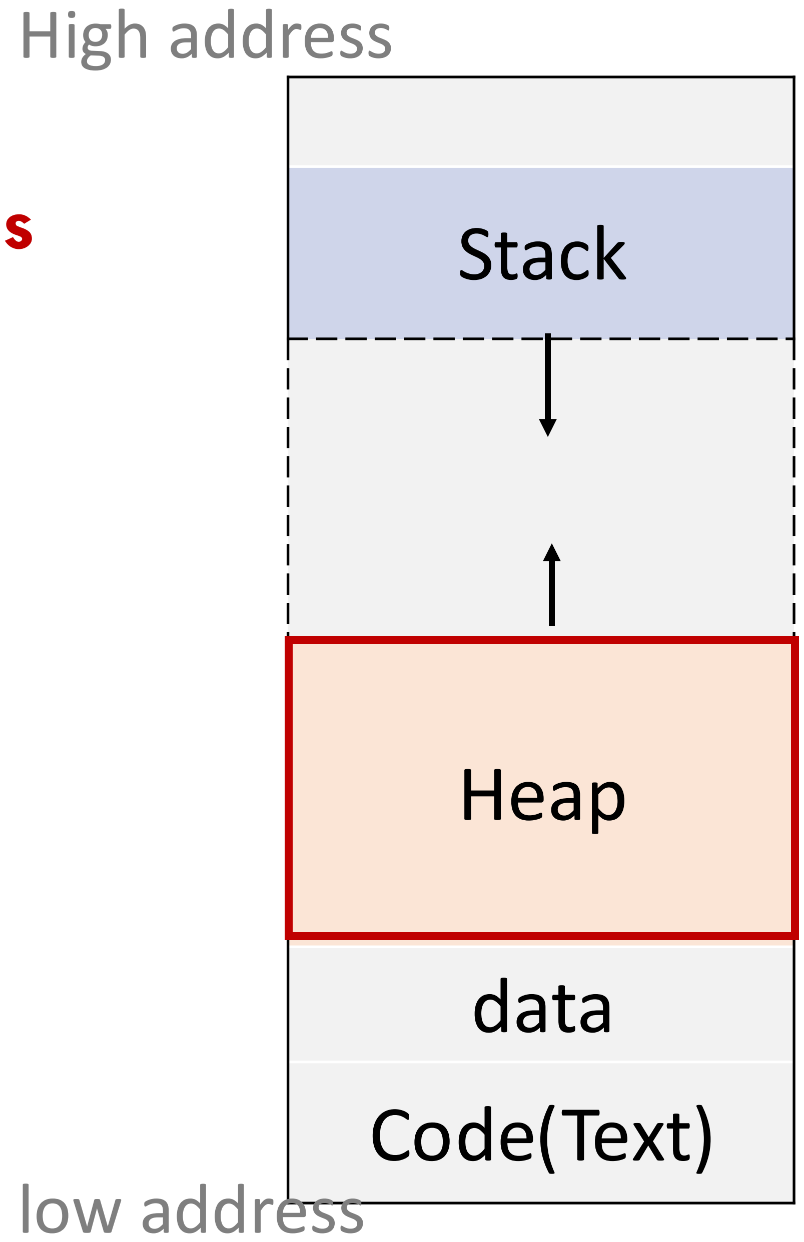
```
int* p = new int(7);

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

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



Heap Memory   delete expression

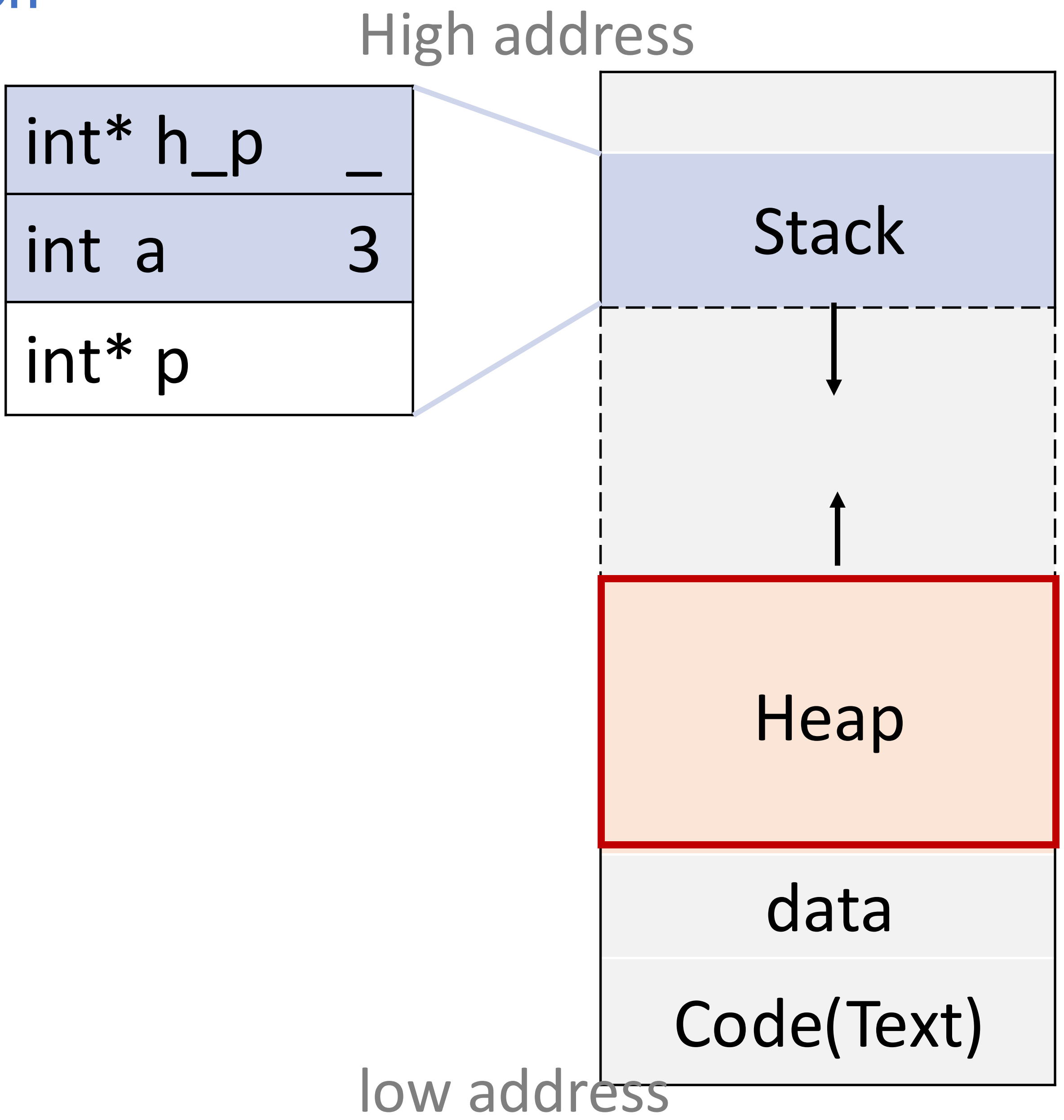
- delete expression: **destroys** object previously allocated by the new-expression and **releases** obtained memory area back to OS.

```
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 new expression

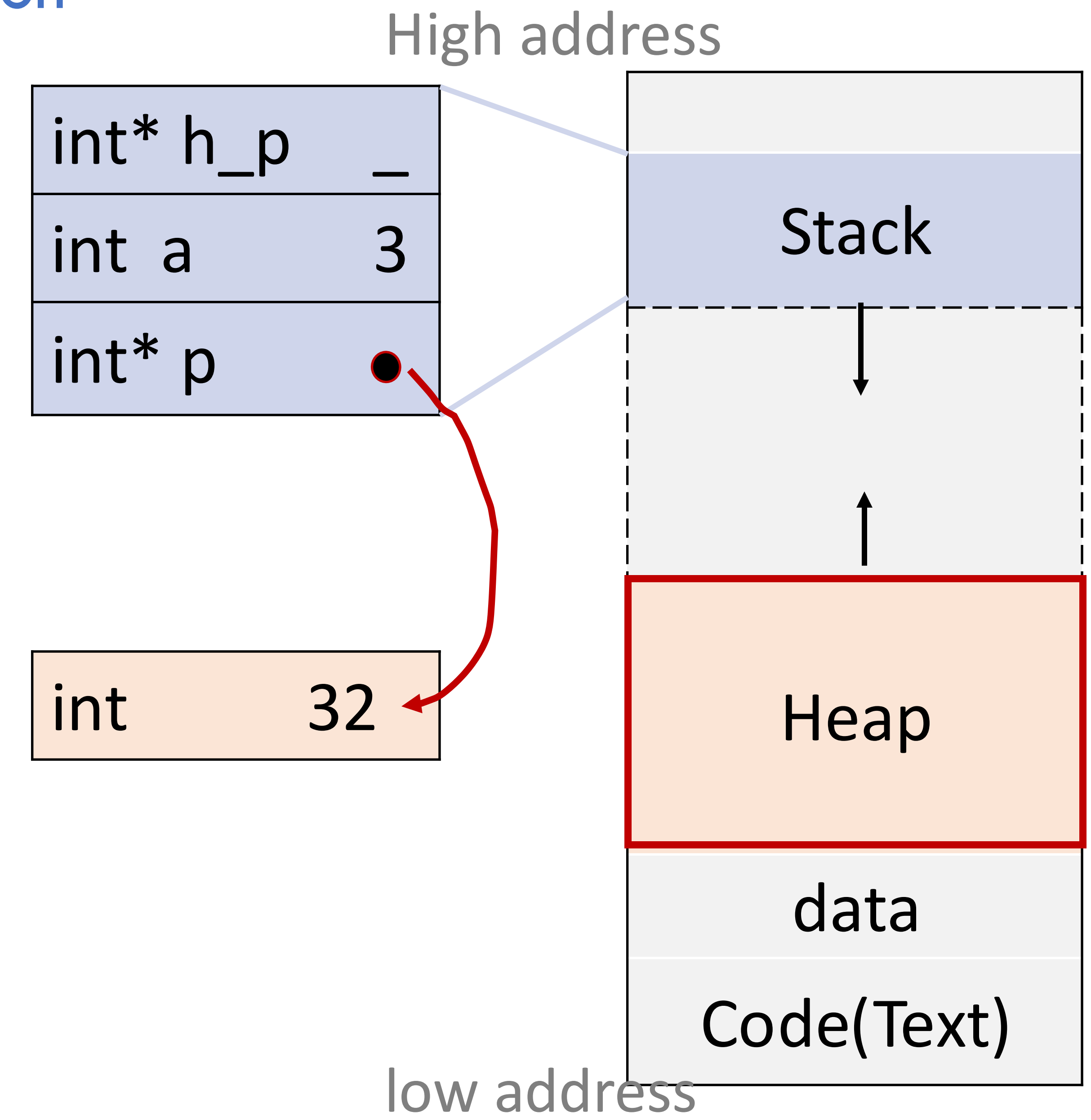
```
int* helper()
{
    int a = 3;
    int * p = new int(32);
    return p;
}
int main(){
     int* h_p = helper();
     delete h_p;
     .....
}
```



Heap Memory

new expression

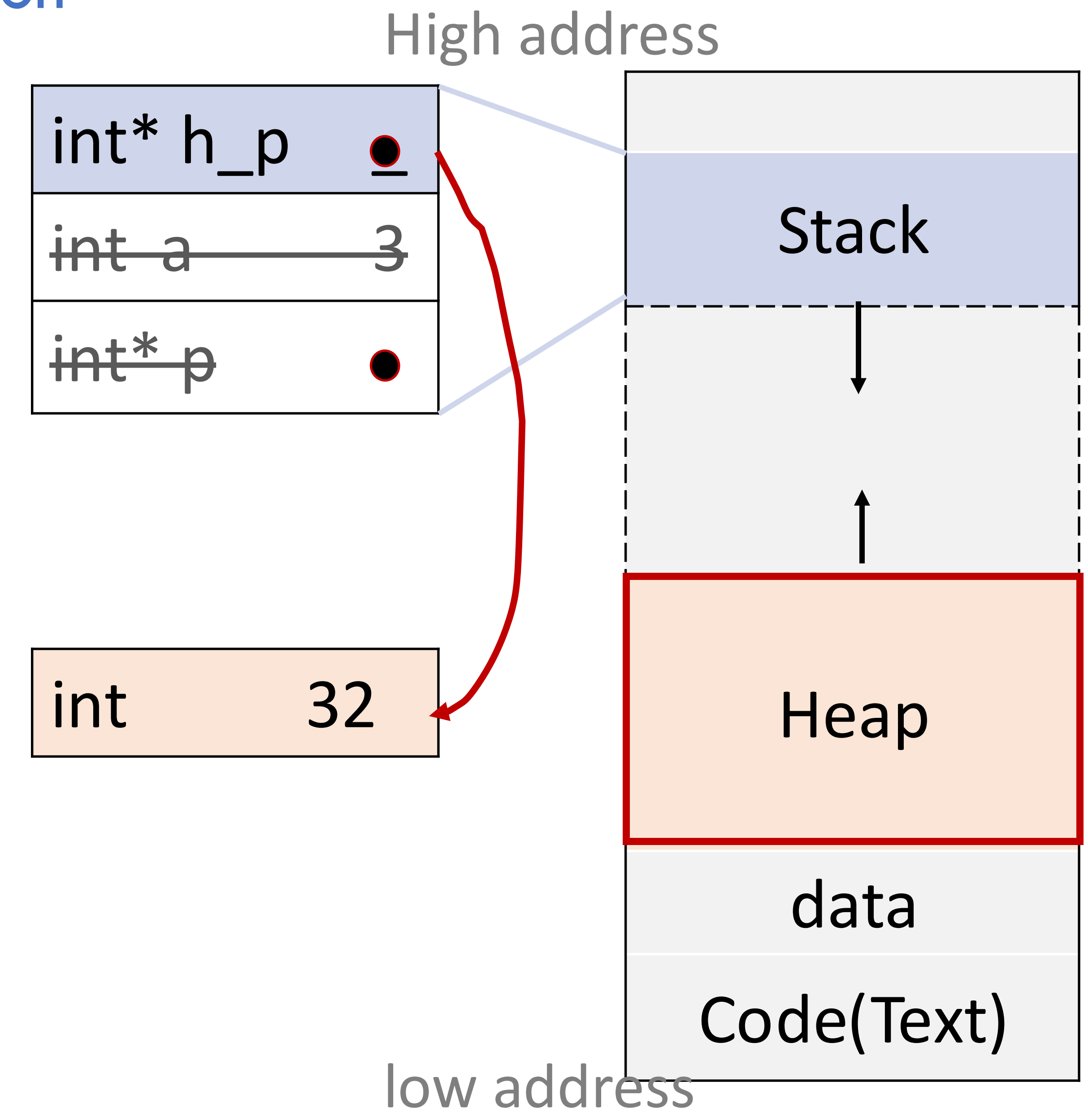
```
int* helper()
{
    int a = 3;
    int * p = new int(32);
    return p;
}
int main(){
    int* h_p = helper();
    delete h_p;
    .....
}
```



Heap Memory

new expression

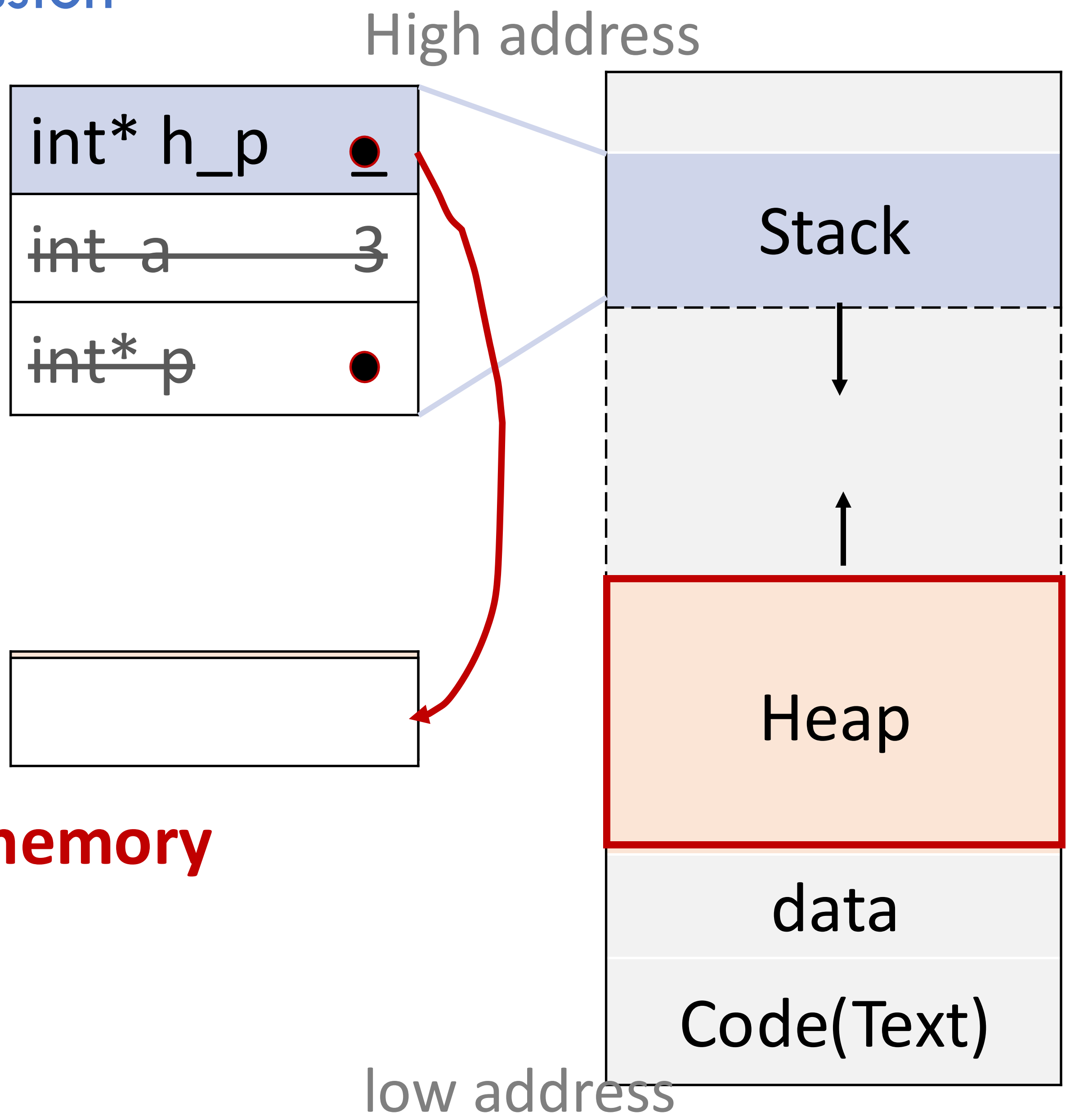
```
int* helper()
{
    int a = 3;
    int * p = new int(32);
    return p;
}
int main(){
    int* h_p = helper();
    delete h_p;
    .....
}
```



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
     .....
}
```

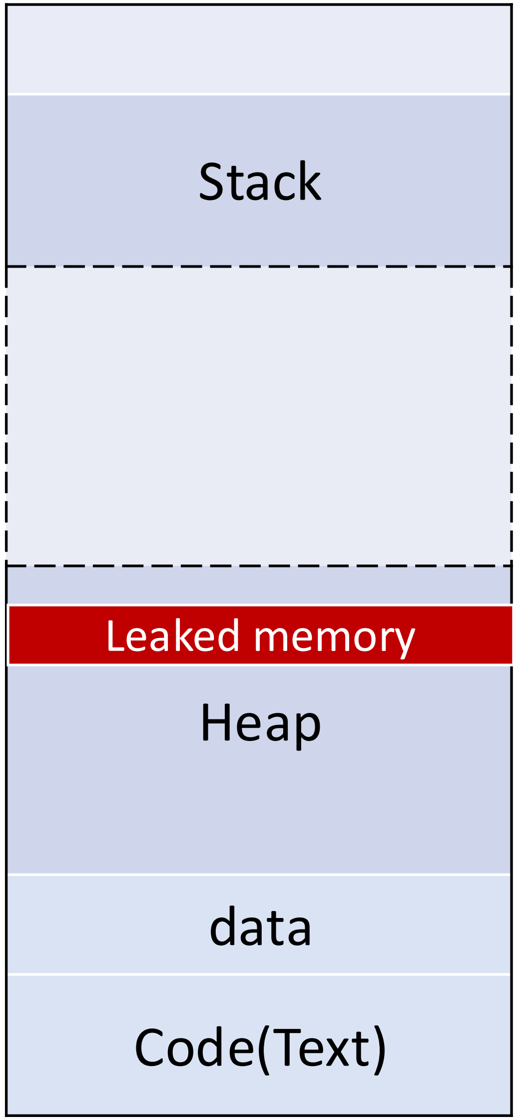


What is memory leak in C++?

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

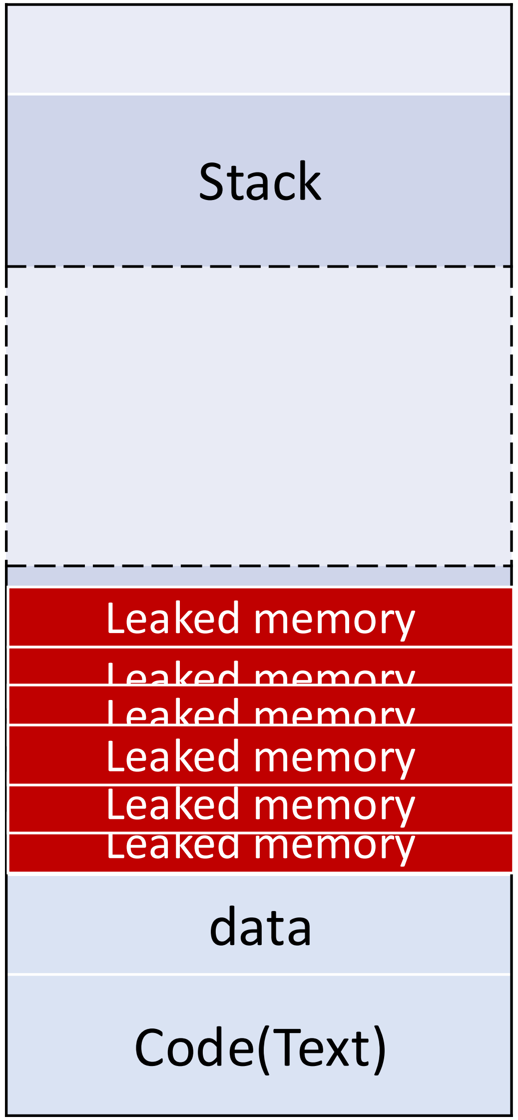
Memory Leak

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



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

C++ Memory

- review
- Smart pointer
- Applications at function and classes

Ownership

- For C++ ownership is the responsibility for cleanup.
  - C-style raw pointer : does not represents ownership
  - std::unique_ptr: sole owner of resource and will clean up the resources when it’s destroyed
  - std::shared_ptr: a group of owners who are collectively responsible for the resource. The last of them to get destroyed will clean it up

Rectangle example from last recitation

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

rectangle.hpp

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

rectangle.cpp

std::unique_ptr --- construct

std::unique_ptr could own:

- No object. Nullptr
- Single object on heap (i.e. allocated with new)
- Heap-allocated array of objects (i.e. allocated with new[])

std::unique_ptr --- construct

```
std::unique_ptr<Rectangle> rec;
```

// default-initialized unique_ptr, rec is a nullptr

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

//Name of the unique_ptr object

// Create object on heap

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

// 1. Create an unique_ptr

// 2. create the object that explicit_rec points to on heap

std::unique_ptr --- construct

```
std::unique_ptr<int[]> heap_arr(new int[5]);
```

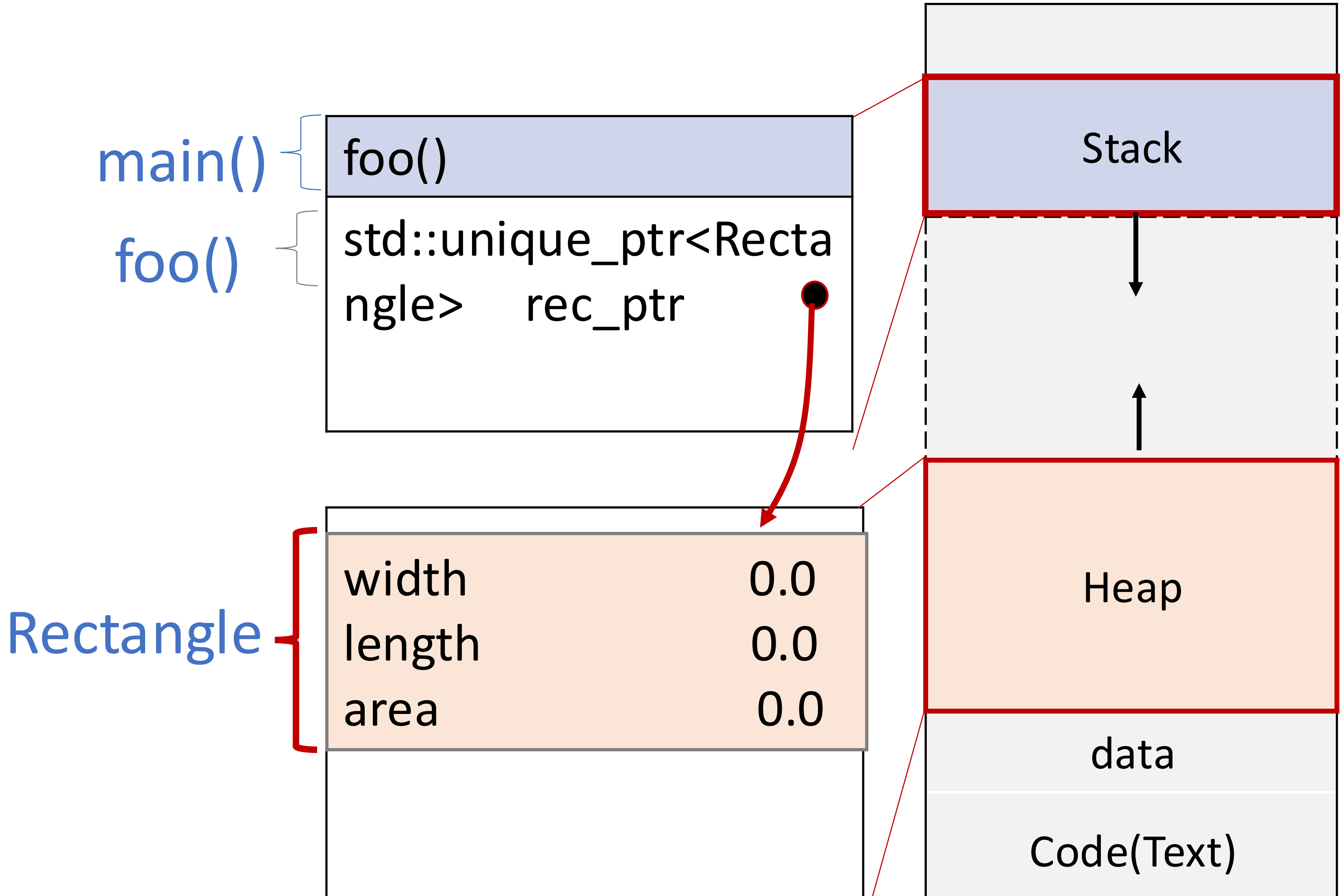
✓

```
std::unique_ptr<int[]> stack_arr(int[5]);
```

✗

```
// int[5] creates an array on stack
// unique_ptr only owns heap-allocated object/array
```

```
void foo(){
    std::unique_ptr<Rectangle> rec_ptr = std::make_unique< Rectangle>();
}

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



std::unique_ptr --- disposed

std::unique_ptr is disposed of when either of the following happens:

- The unique_ptr object is destroyed. (Out of scope)
- The unique_ptr object is assigned to another pointer via = or reset()

std::unique_ptr --- disposed

When destroyed, std::unique_ptr uses the user-supplied or default delete to release the resources

```
template<
    class T,
    class Deleter = std::default_delete<T>
> class unique_ptr;
```

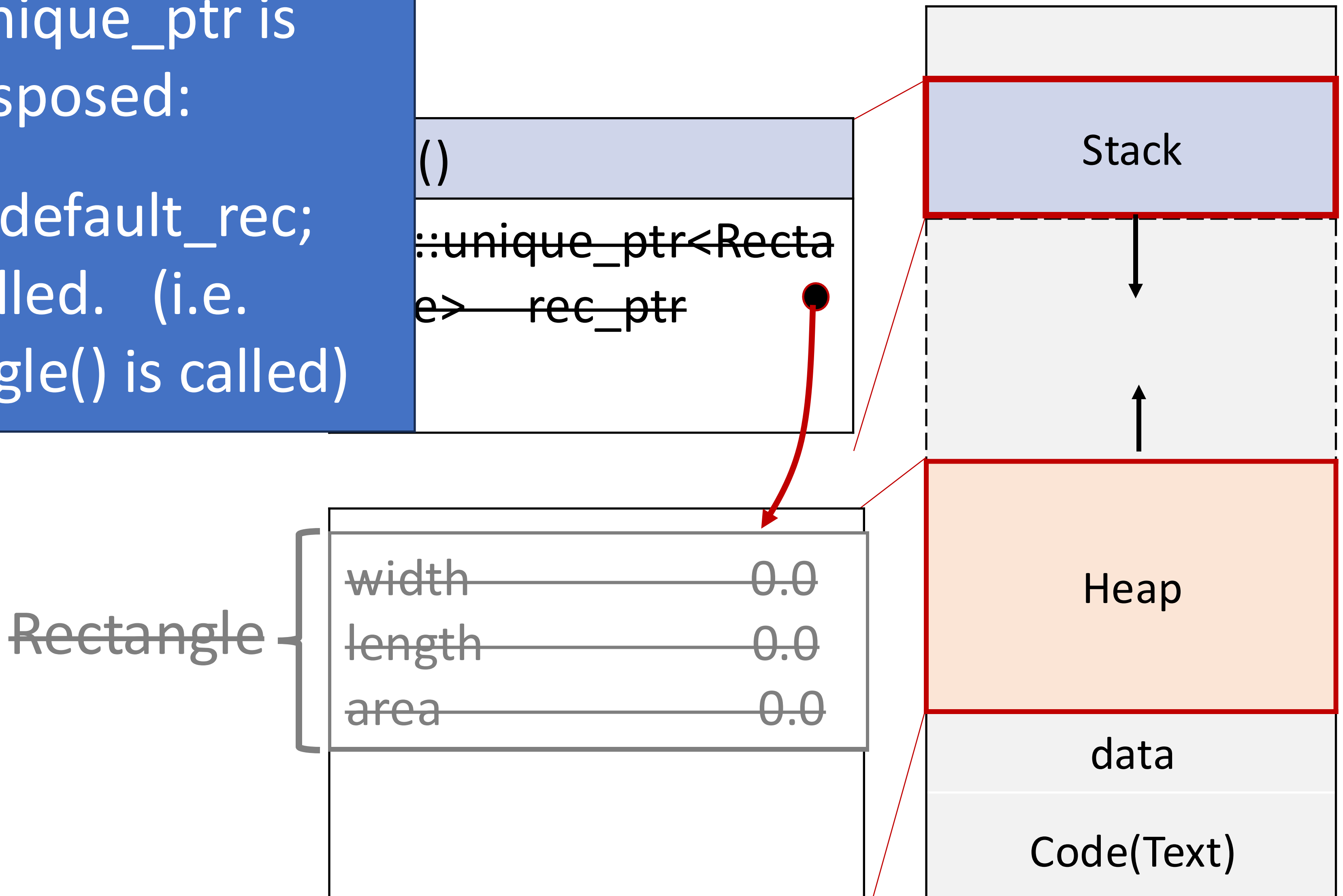
```
template <
    class T,
    class Deleter
> class unique_ptr<T[], Deleter>;
```

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

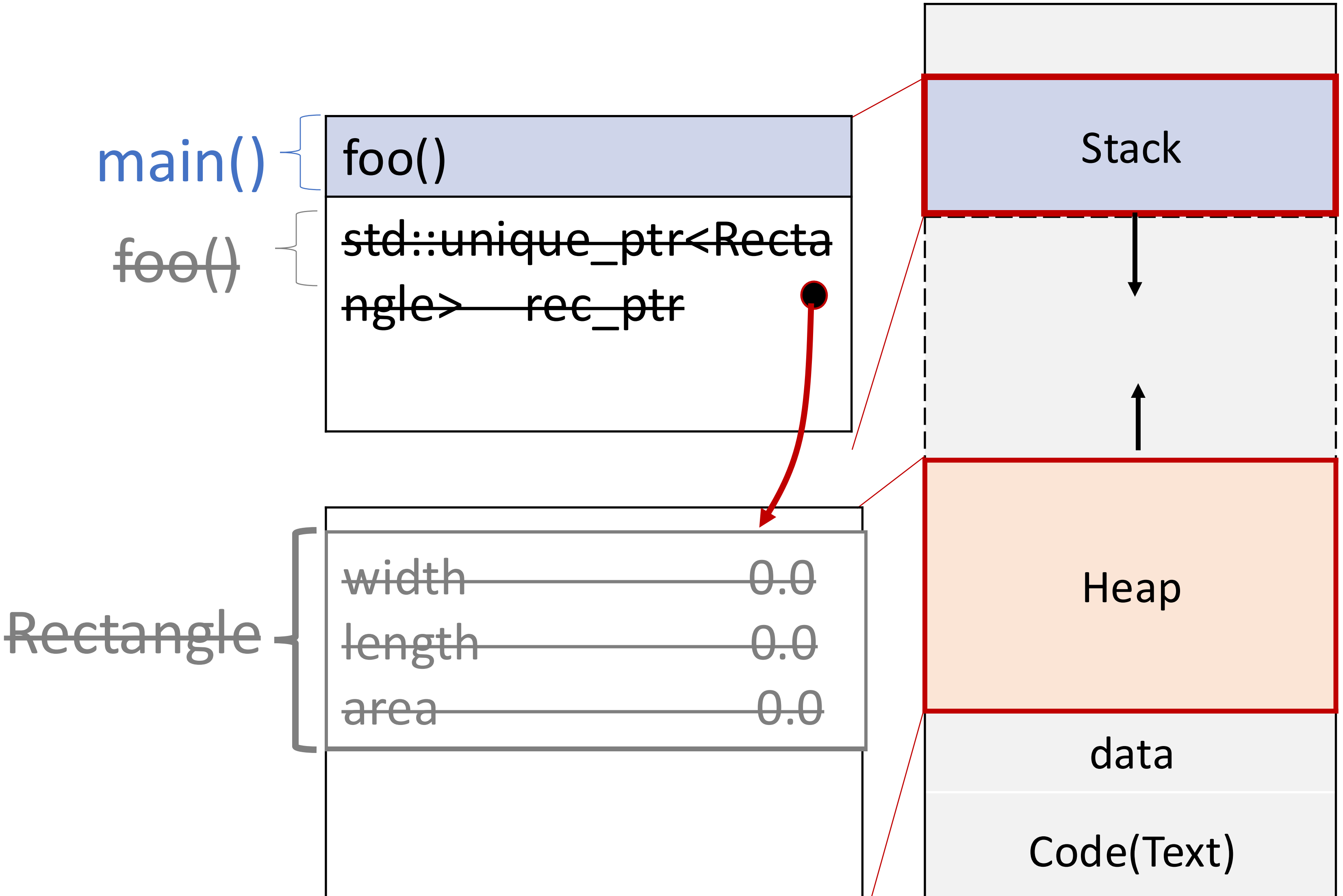
```
void foo(){
    std::unique_ptr<Rectangle> rec_ptr= std::make_unique< Rectangle>();
}

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

std::unique_ptr is disposed:

delete default_rec; is called. (i.e. ~Rectangle() is called)



```
void foo(){
    std::unique_ptr<Rectangle> rec_ptr =std::make_unique< Rectangle>()
}

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



std::unique_ptr --- transfer ownership

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

std::unique_ptr< Rectangle > rec2 = std::move(explicit_rec );
        // ownership of Rectangle managed by explicit_rec is now transferred to rec2.
        // explicit_rec is now nullptr
std::unique_ptr< Rectangle > rec3 = rec2;

        // unique_ptr cannot be copied, only moved
```

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

std::unique_ptr --- dereference

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

int rec_width = explicit_rec->width;  // operator -> accesses members of the
                                      object managed by unique_ptr

(* explicit_rec).width = 10;  // operator * dereferences the smart pointer
```

std::shared_ptr

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

std::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;
```

C++ Memory

- review
- Smart pointer
- Applications at function and classes

C++ Functions

- How to use C++ memory resources for my program?

Function Returns --- pointer

What can go wrong?

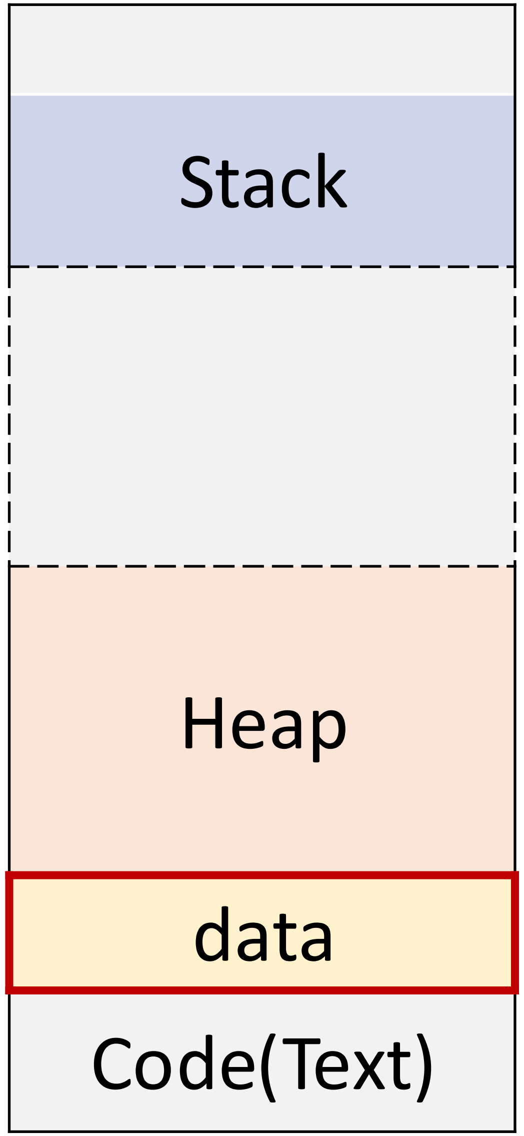
Dangling pointers

```
int* dangerousFunc() {
        int localVar = 100;
        return &localVar;
}
int main() {
        int* res = dangerousFunc();
        std::cout << *res << std::endl;
        …
}
```

Undefined behavior!

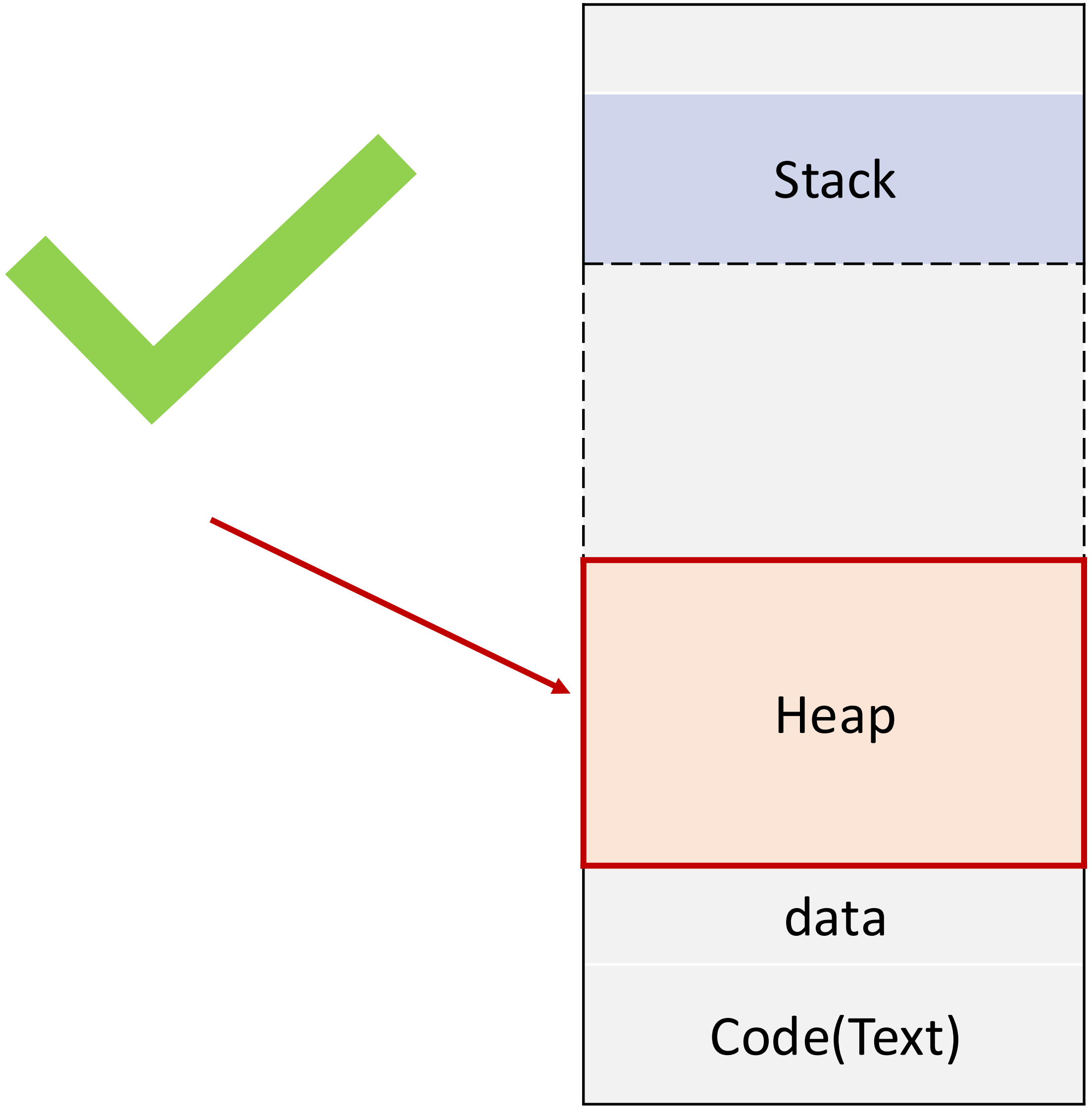
Function Returns --- Fix1 (static or global)

```
int localVar = 100;
int* safeFunc() {
        return &localVar;
}
int main() {
        int* res = safeFunc();
        std::cout << *res << std::endl;
...
}
```



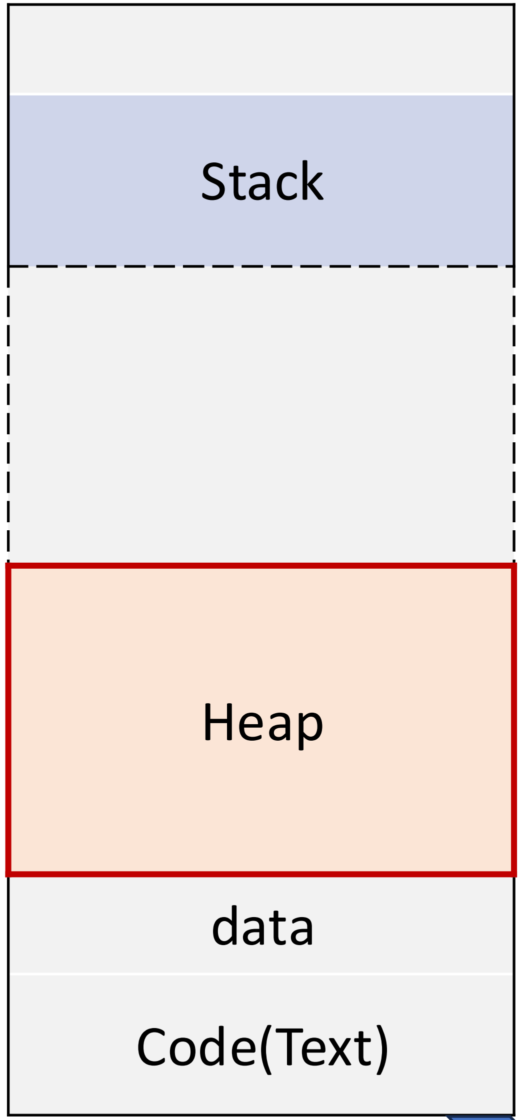
* Function Returns --- Fix2 (use heap)

```
int* safeFunc() {
        int* var_ptr = new int(100);
        return var_ptr;
}
int main() {
        int* res = safeFunc();
        std::cout << *res << std::endl;
        delete res;  ...
}
```



* Function Returns --- Fix3 (smart pointer)

```
std::unique_ptr<int> safeFunc() {
    std::unique_ptr<int> var_ptr = std::make_unique<int>(100);
    return std::move(var_ptr);
}   // return var_ptr;   //also works, because of RVO

int main() {
        std::unique_ptr<int> res = safeFunc();
        std::cout << *res << std::endl;
   return 0;
}
```



C++ Containers

C++ Container

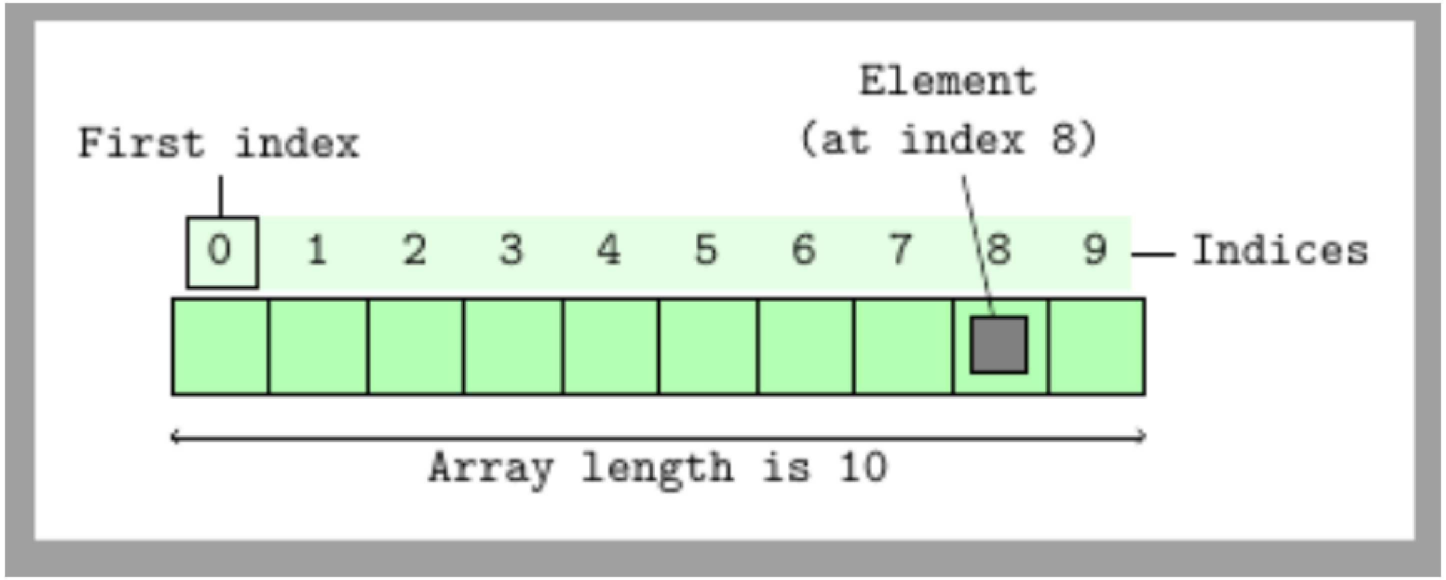
Standard Template Library

- Collection of classes and functions for general purpose use
- Provides container types (list, vector, map, ...), pair, tuple, string, thread and many other functionalities
- Available in the std namespace

C++ Container

- A Container is an object used to **store other objects** and take care of the **management of the memory** of the objects it contains.
- Containers include many commonly used structure:
  - std::array,
  - std::vector,
  - std::queues,
  - std::map,
  - std::set,
  - …

Array



- Arrays must be declared by type and size
- The size must be fixed at compile-time
- Stores elements contiguously (in continuous memory locations)
- Elements are accessed starting with position 0 (0-based indexing)
- O(1) access given the index of the element

C-style array (raw array)

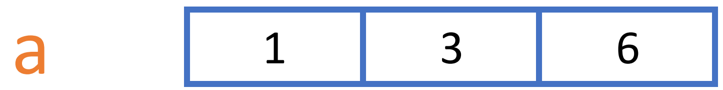
- C-style array is a block of memory that can be interpreted as an array

```
int  a[10];
```

// declare **a** as an **array object** that consist of 10 **contiguous allocated** objects of **type int**

```
int  a[3] = {1 , 3,  6} ;
```

// assignment of objects in array



std::array<T, N> ---a container that holds fixed size arrays

- Has the same semantics as a C-style array, but implemented by standard template library
- To use this container, include it at the beginning of the file

```
#include <array>
```

- T and N are template parameters: T is the type of the array, and N defines the number of elements
  - E.g., `std::array<char, 10>`, `std::array<int, 3>`

std::array<T, N> ---a container that holds fixed size arrays

- Has the same semantics as a C[...] by standard template library
- To use this conta[...]

  #include <array>

Why use std::array offered by C++ Standard Template Library(std)?

- T and N are template parameters: T is the type of the array, and N [...]es the number of elements

  [...]g., std::array<char, 10>, std::array<int, 3>

C-style array vs. std::array<T, N>

- C-style array
  - No bound check when accessing element using operator[]
    - Undefined result if access a[20] if a is an array with size 3
  - Array-to-pointer decay
    - E.g., When pass a C-style array as **a value** to a function it decays to **a pointer** of the first element in the array, losing the size information.

C-style array vs. std::array<T, N>

- C-style array characteristics
  - No bound check when accessing element using operator[]
  - Array-to-pointer decay

```
void print_array(int arr[]){
    size_t arr_size = sizeof(arr) / sizeof(int);
    for(int i = 0; i < arr_size; ++ i){
        std::cout << arr[i] << std::endl;
    }
}
```

→

```
void print_array(int * arr){
    size_t arr_size = sizeof(arr) / sizeof(int);
    for(int i = 0; i < arr_size; ++ i){
        std::cout << arr[i] << std::endl;
    }
}
```

```
yy354@en-ci-cisugcl14:~/CS4414Demo/recitation2$ g++ -fstack-protector-all array_example.cpp -o arr
array_example.cpp: In function 'void print_arr(int*)':
array_example.cpp:11:34: warning: 'sizeof' on array function parameter 'arr' will return size of 'int*' [-Wsizeof-array-argument]
   11 |     size_t arr_size = sizeof(arr) / sizeof(int);
      |                                 ^
array_example.cpp:10:20: note: declared here
   10 | void print_arr(int arr[]){
      |                ~~~~^~~~~
```


https://cppinsights.io

C-style array vs. std::array<T, N>

Std::array<T> has more functions of standard container, makes it easier to use

```
std::array<int, 3> a = {1, 2, 3};
```

- size() : get the size of the array

```
std::cout << a.size() << std::endl;
```

- at() / operator [] : access specified element with bounds checking

```
std::cout << a.at(2) << std::endl;
```

- Use iterator to access container elements

```
for(auto it = a.begin(); it < a.end(); ++it )
{....}
```

- More functionalities: https://en.cppreference.com/w/cpp/container/array

std::vector<T>

- T is a template parameter
- Std::vector<int> is a vector of integers, std::vector<char> is a vector of characters
- Same as std::array, T can be a class or other C++ container
  - E.g., std::vector<Rectangle>,

    std::vector<std::map<int, std::string>>…

std::vector<T>

- T is a template para
- Std::vector<in tor of characters
- Same as std::array, T n be a class or other C++ container

Why do I want to use std::vector<T> ?

g., std::vector<Rectangle>,

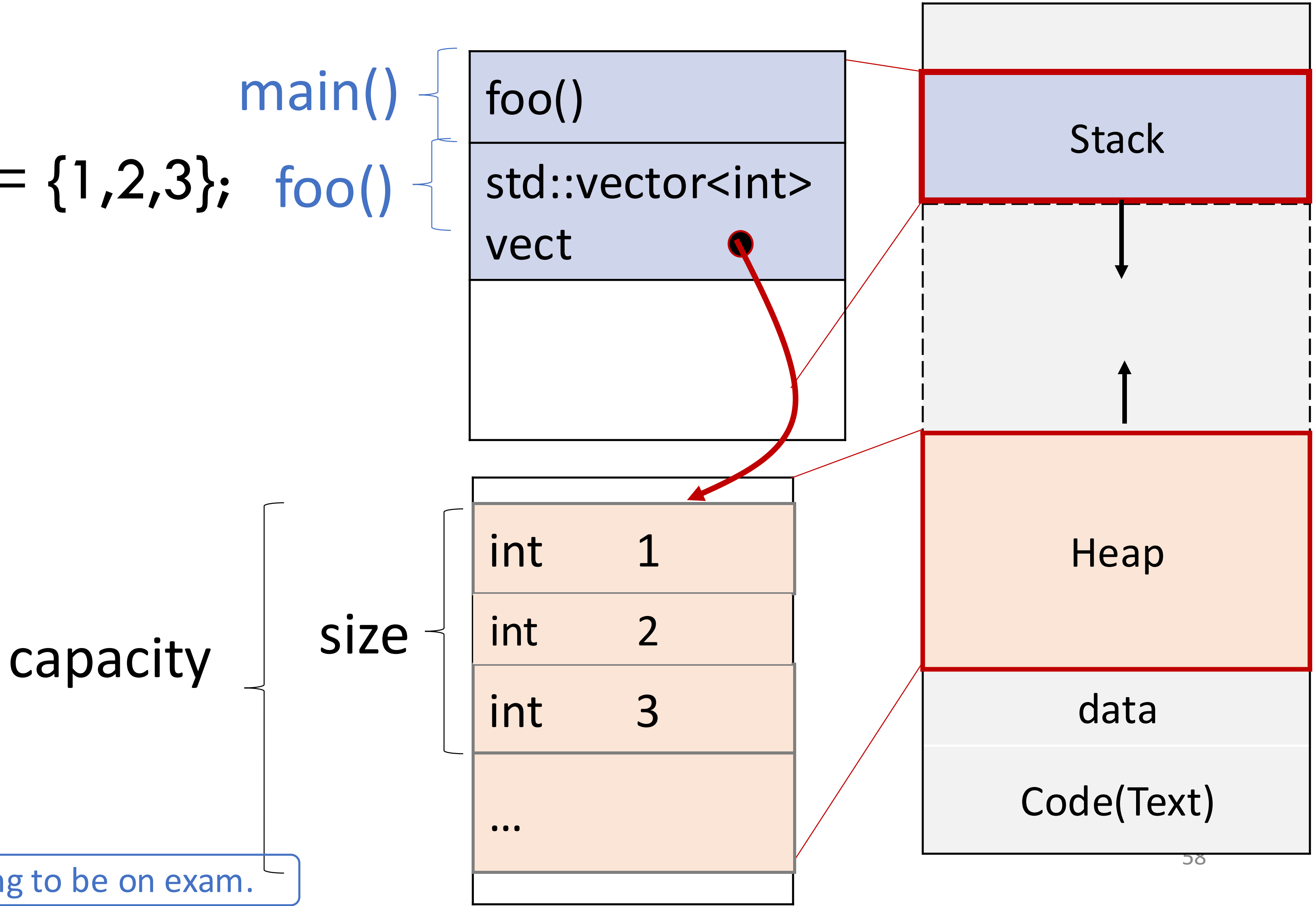
std::vector<std::map<int, std::string>>…

std::vector<T> - A dynamicly-sized array

- Main problem: How to support adding elements efficiently?
- Concept of size vs. capacity

std::vector<T> - under the hood memory structure

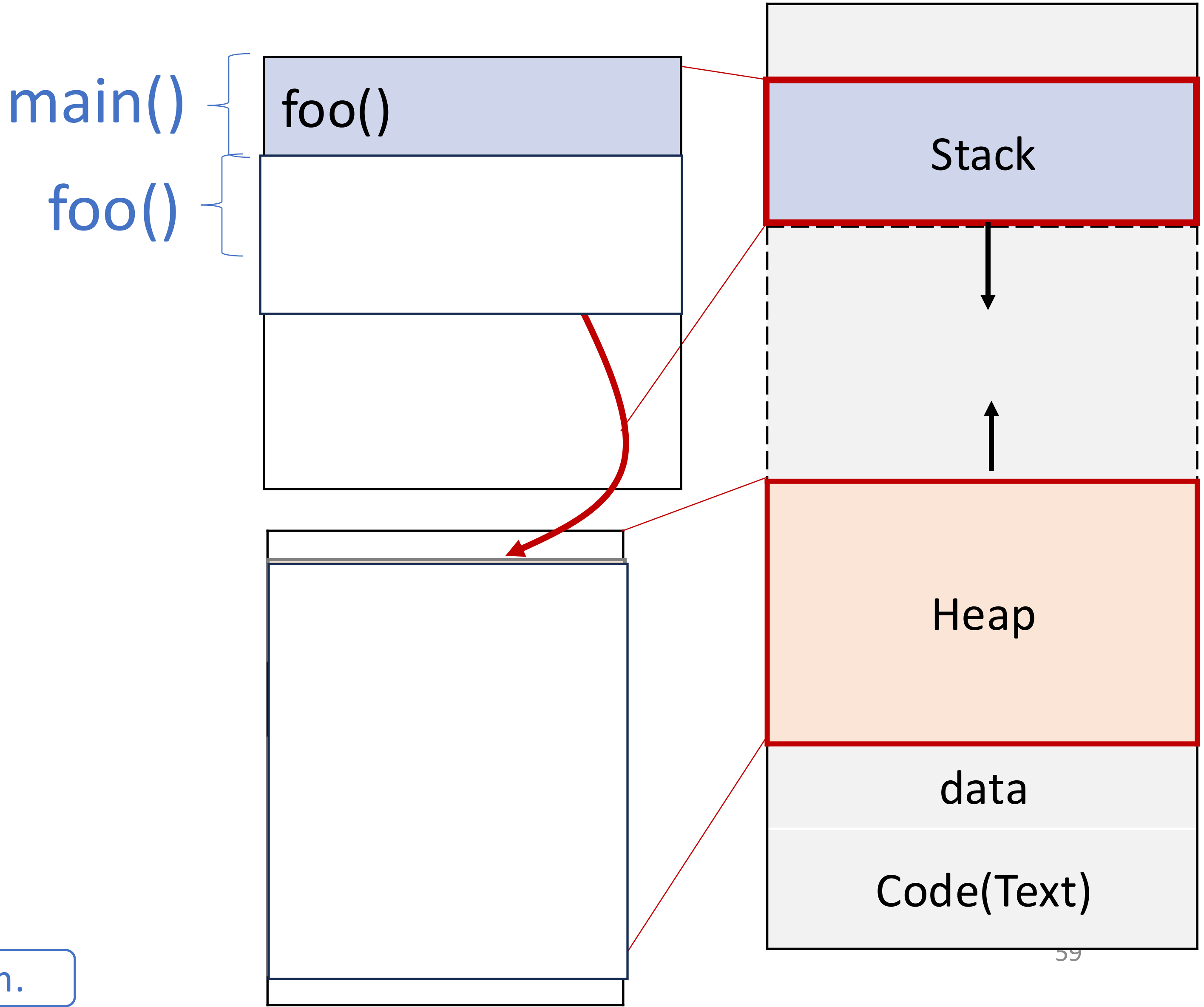
```
void foo(){
    std::vector<int> vect= {1,2,3};
}

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



This slide is for curiosity only, not going to be on exam.

std::vector<T> - under the hood memory structure

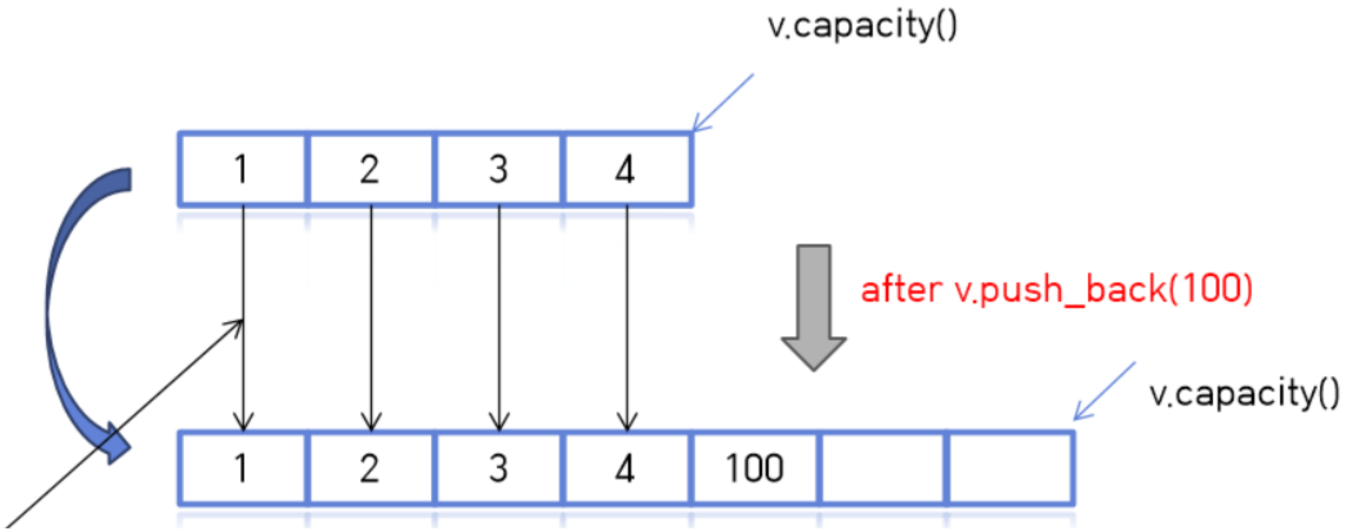
```
void foo(){
  std::vector<int> vect= {1,2,3};
}

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



This slide is for curiosity only, not going to be on exam.

std::vector<T> - A dynamic-sized array

- Main problem: How to support adding elements efficiently?
- Concept of size vs. capacity
- Reallocates elements when capacity is exceeded



std::vector<T> - functionalities

demo

- Element access: operator [], at, front, back, data
- Iterators: begin, end, rbegin, rend
- Capacity: size, capacity, reserve
- Modifiers: emplace, push_back, erase, resize

https://en.cppreference.com/w/cpp/container/vector

Complexity of std::vector<T>::push_back

- Most push_backs will be O(1) (when size < capacity)
- Some will have linear complexity (when the vector is reallocated)
- Amortized O(1) complexity with exponential growth in capacity
- What about the complexity of inserting at a random position in the vector?

C++ Functions

- How to use C++ container for my program?

Function Parameter

demo

- Pass by value

vect.size() = **3**

vect.size() = **2**

```
void func(vector<int> vect)
{
   vect.push_back(30);
}

int main()
{
    vector<int> vect;
    vect.push_back(10);
    vect.push_back(20);
    func(vect);
}
```

Function Parameter

```
void func(std::vector<int> vect)
{
   vect.push_back(30);
}
int main()
{
    std::vector<int> vect;
    vect.push_back(10);
    vect.push_back(20);

    func(vect);
}
```

← func() will not work as intended:

- changes made inside the function are not reflected outside because the function only changes the copy of vect.
- it might also take a lot of time in cases of large vectors.

demo

Function Parameter --- passing by reference

vect.size() = **3**

```
void func(vector<int>& vect)
{
   vect.push_back(30);
}

int main()
{
    vector<int> vect;
    vect.push_back(10);
    vect.push_back(20);

    func(vect);
}
```

vect.size() = **3**

Exercise

- Pick a large N (> 1 million)
- Program A: Creates a vector of N elements and assigns vec[i] = i for each i in a for-loop
- Program B: Creates an empty vector and calls vec.push_back(i) N times in a for-loop
- Program C: Creates an empty vector and calls vec.insert(vec.begin(), N-i-1) N times in a for-loop
- Measure the time taken by program A, B and C

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