CS4414 Recitation 5
C++ memory management and functions

02/24/2023

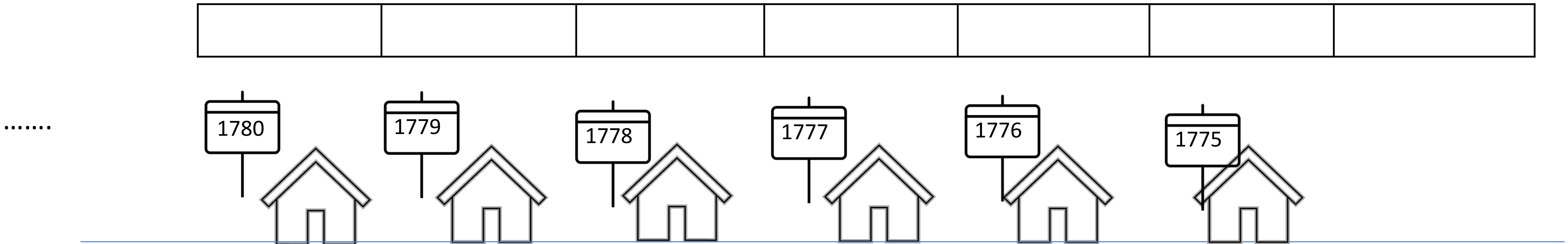
Alicia Yang

C++ Pointers and Reference

- What are C++ Pointer and Reference? Why do we have them?
- How to use C++ pointers and allocate memory for my program?

Pointers

- A pointer is a variable that stores the memory address of an object. Give programmer the ability to manipulate data directly from the computer's memory
- Why use pointers?
  - Save memory: More fine-grained object's life-time control
  - Improve the processing speed.
  - Reduces the length and complexity of a program
  - Provide reference semantics, allow the passing objects to function more efficiently.



Pointers

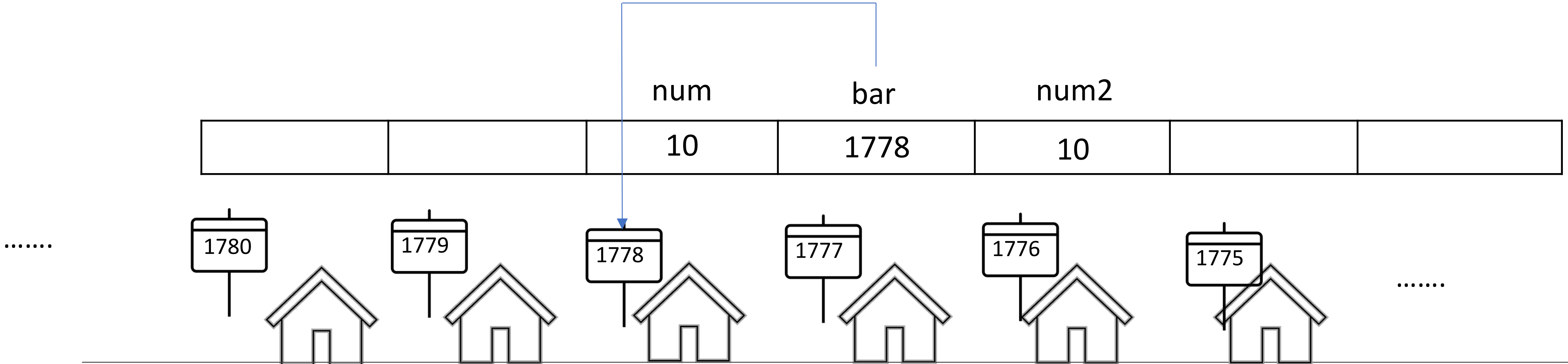
--- Address-of(&) and Dereference(*__) operators

- A pointer is a variable that stores the memory address of an object.
- Example:

```
int num = 10;
int* bar = &num;
int num2 = (* bar);
```

Hey, what IS your memory address?

Hey, what IS stored IN your memory address?

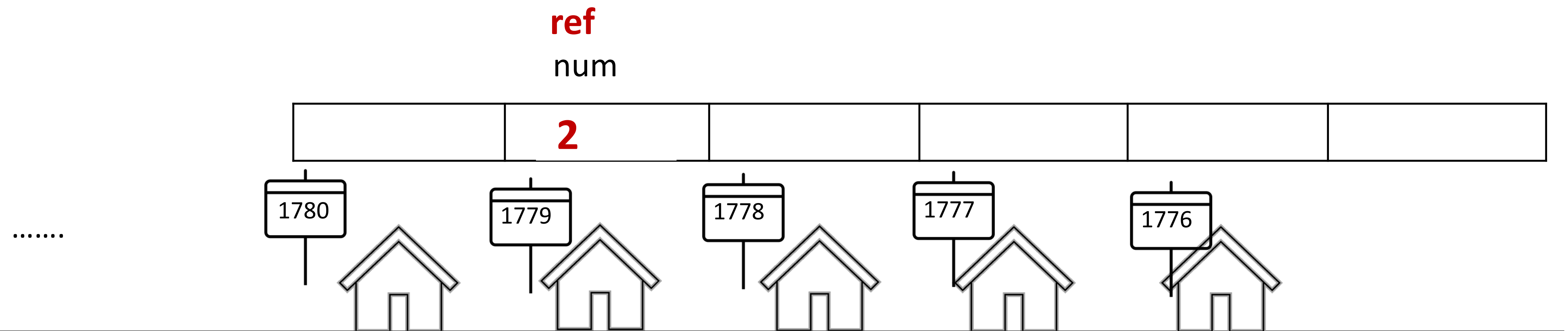


References

- Reference, is an alias, is another name for an already existing variable.
- Changes to the reference are reflected on the original object

```
int num = 10;
int& ref = num;
ref = 2;
```

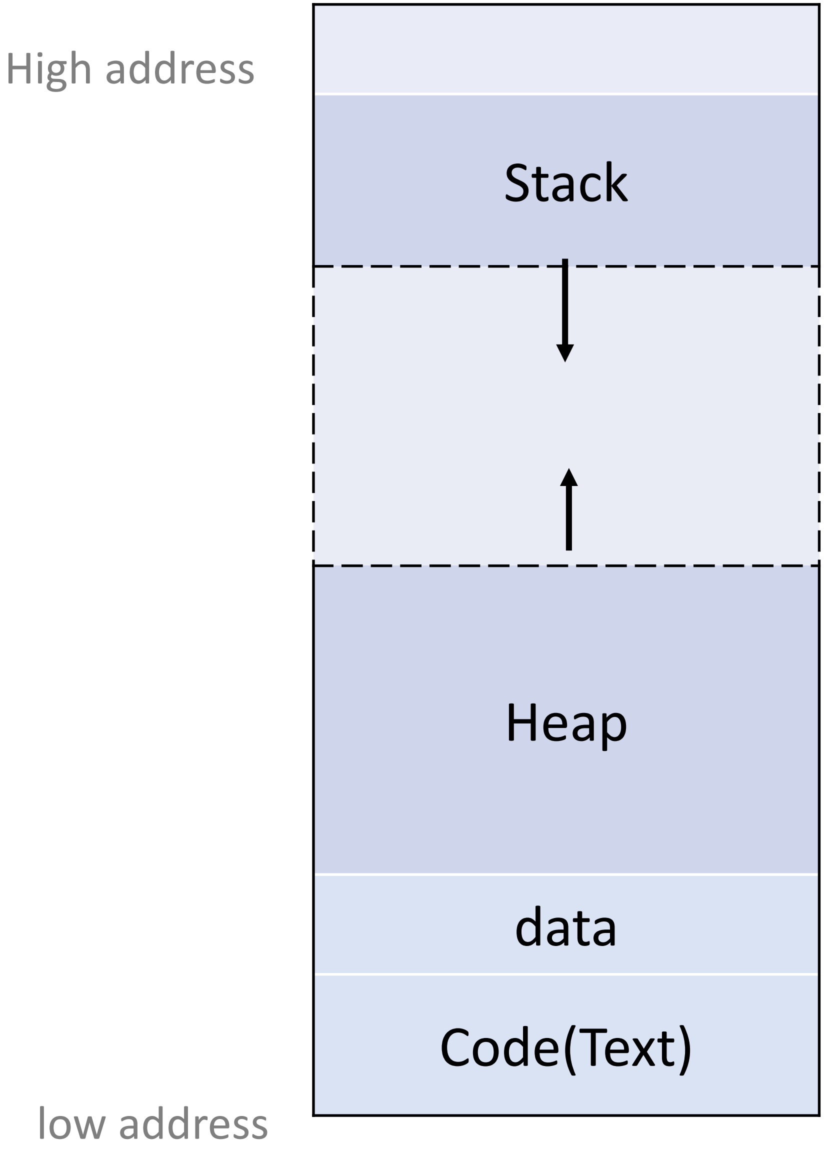
I'm a reference



C++ Memory

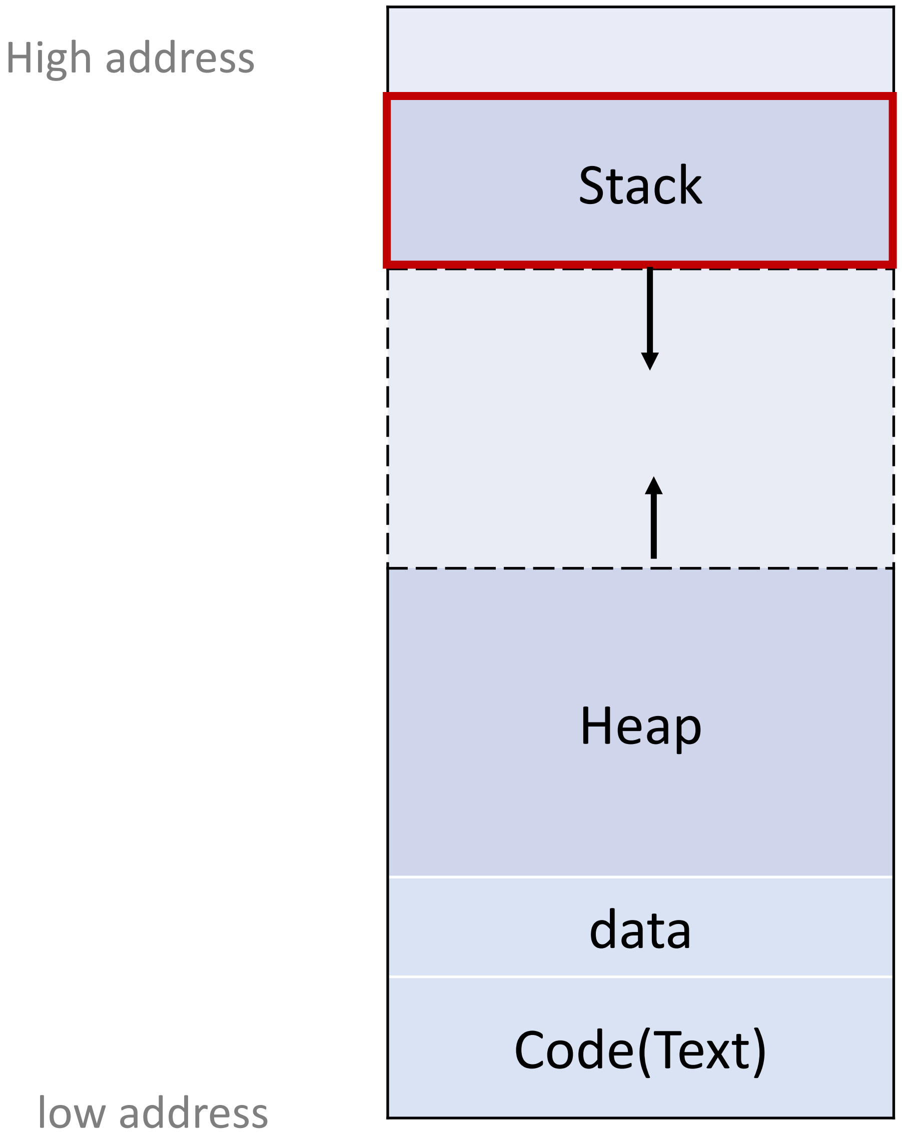
Memory

- Memory for C/C++/Java program
- **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



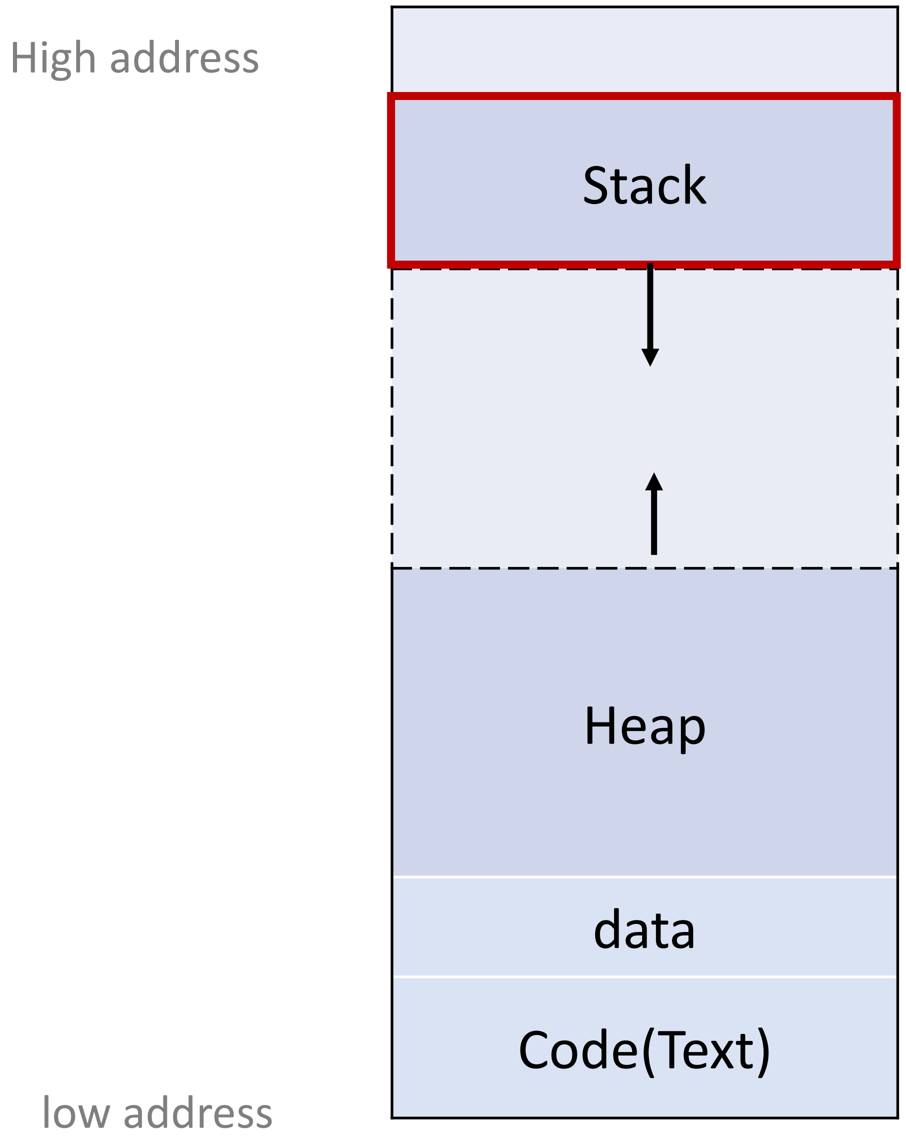
Stack Memory

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



Stack Memory

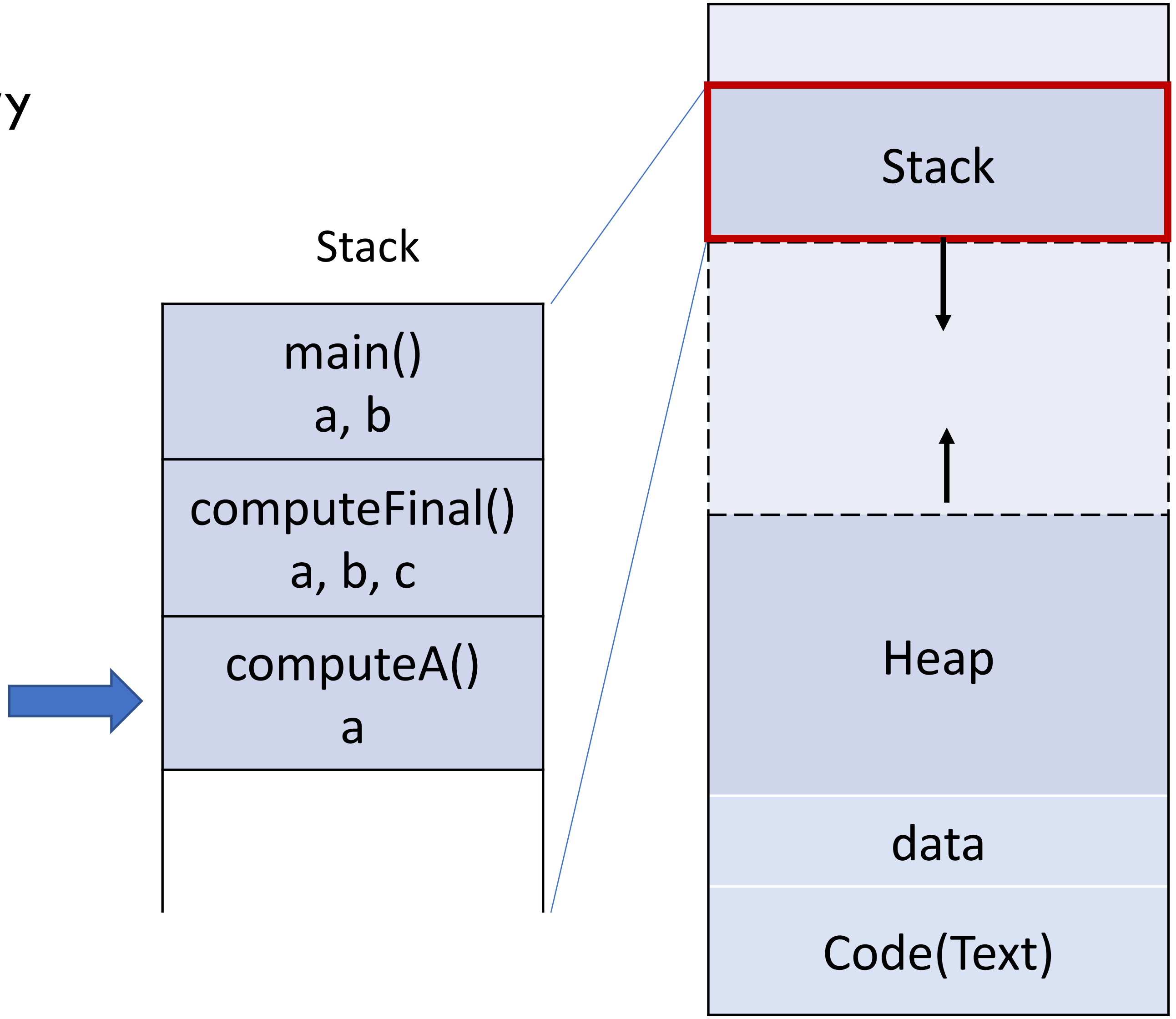
- Stack Allocation (Temporary memory allocation):
  - Allocate on contiguous blocks of memory, in a fixed size
  - Allocation happens in function call stack
  - When a function called, its variables got allocated on stack; when the function call is over, the memory for the variables is deallocated. (scope)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



Stack Memory

- Stack Allocation (Temporary memory allocation):

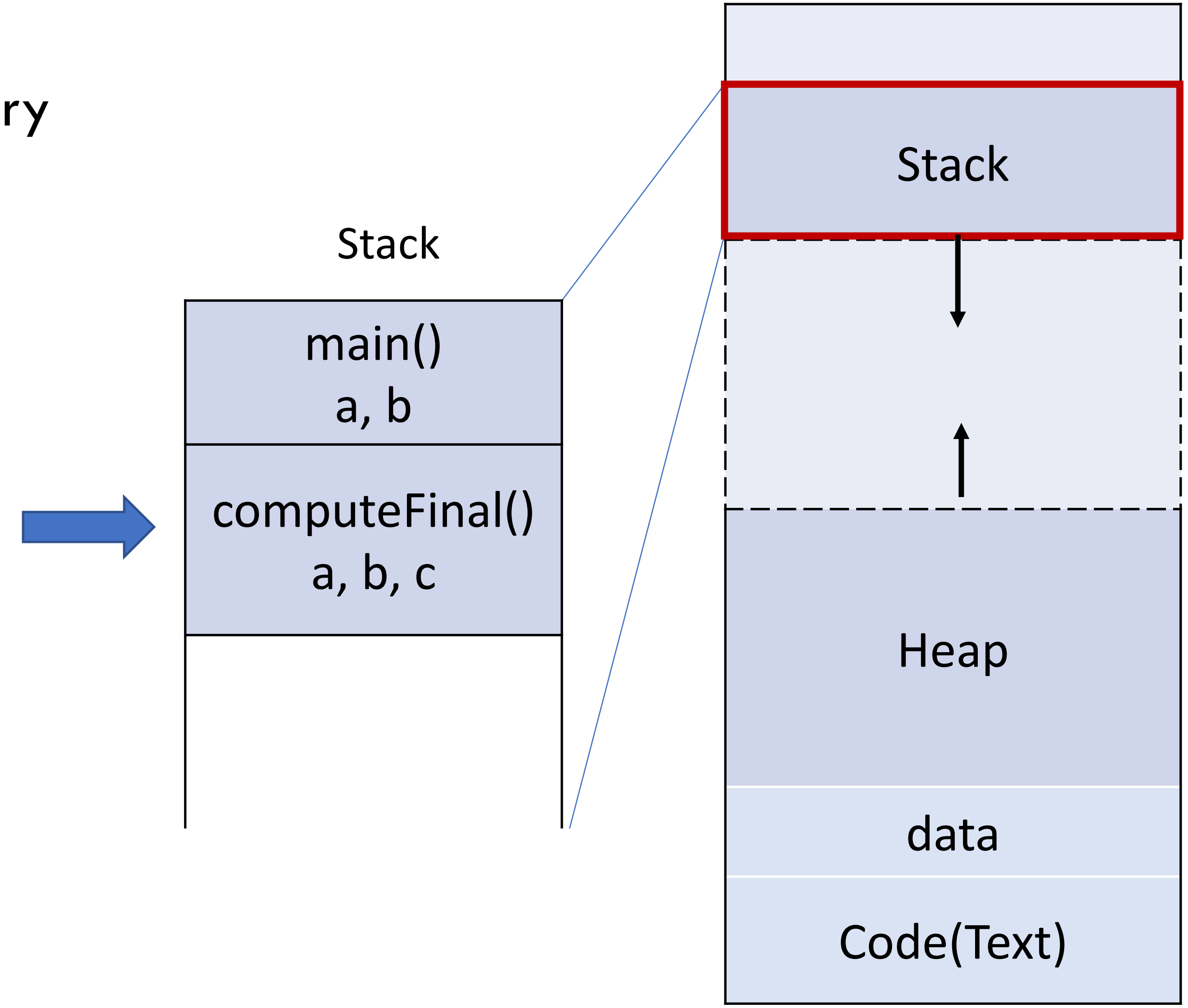
```
int computeA(int a){ return a*a; }

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

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



Stack Memory

- 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;
    total = computeFinal(a, b);
    ...
}
```

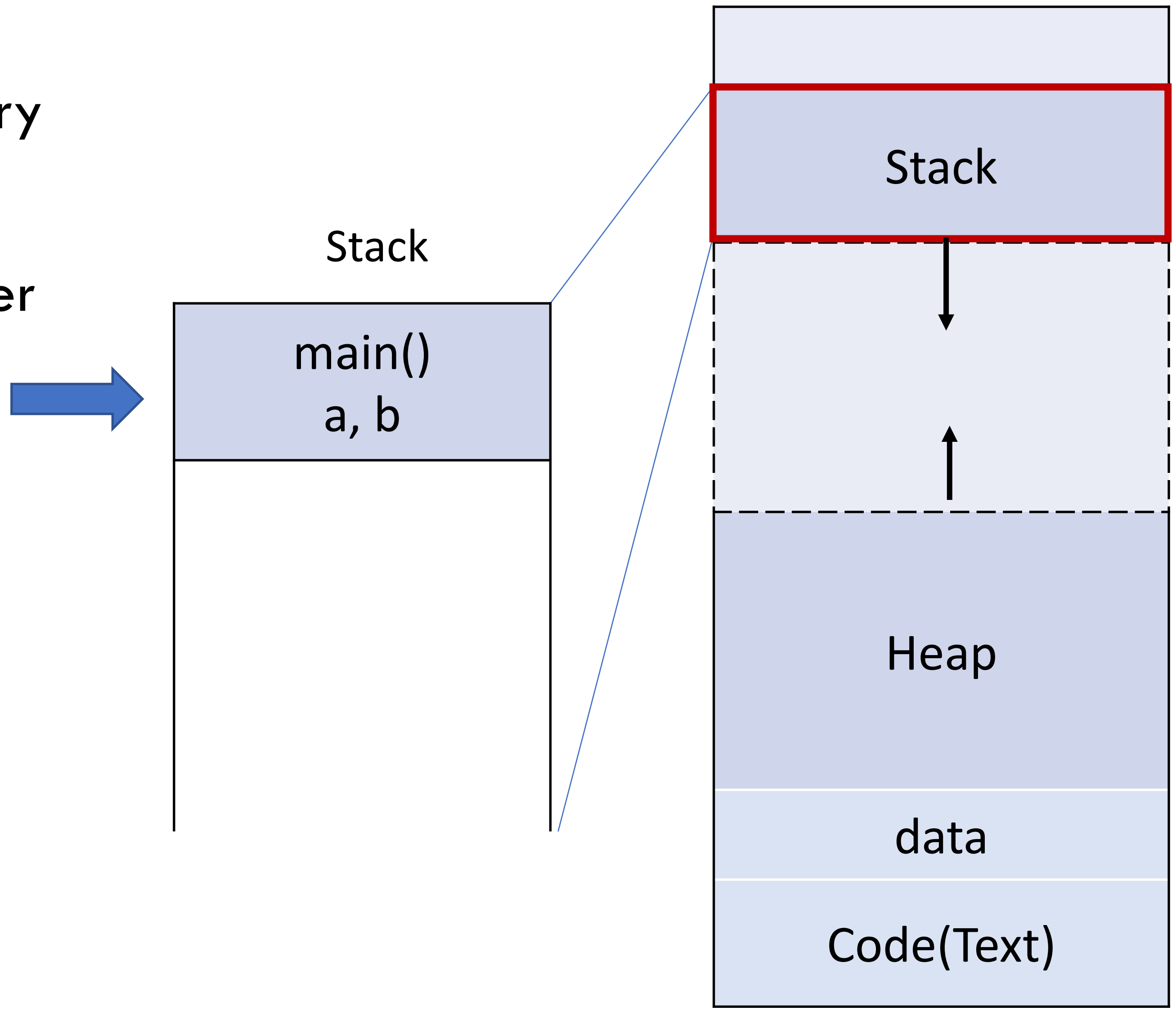


Stack Memory

- Stack Allocation (Temporary memory allocation):

Stack free memory via stack pointer

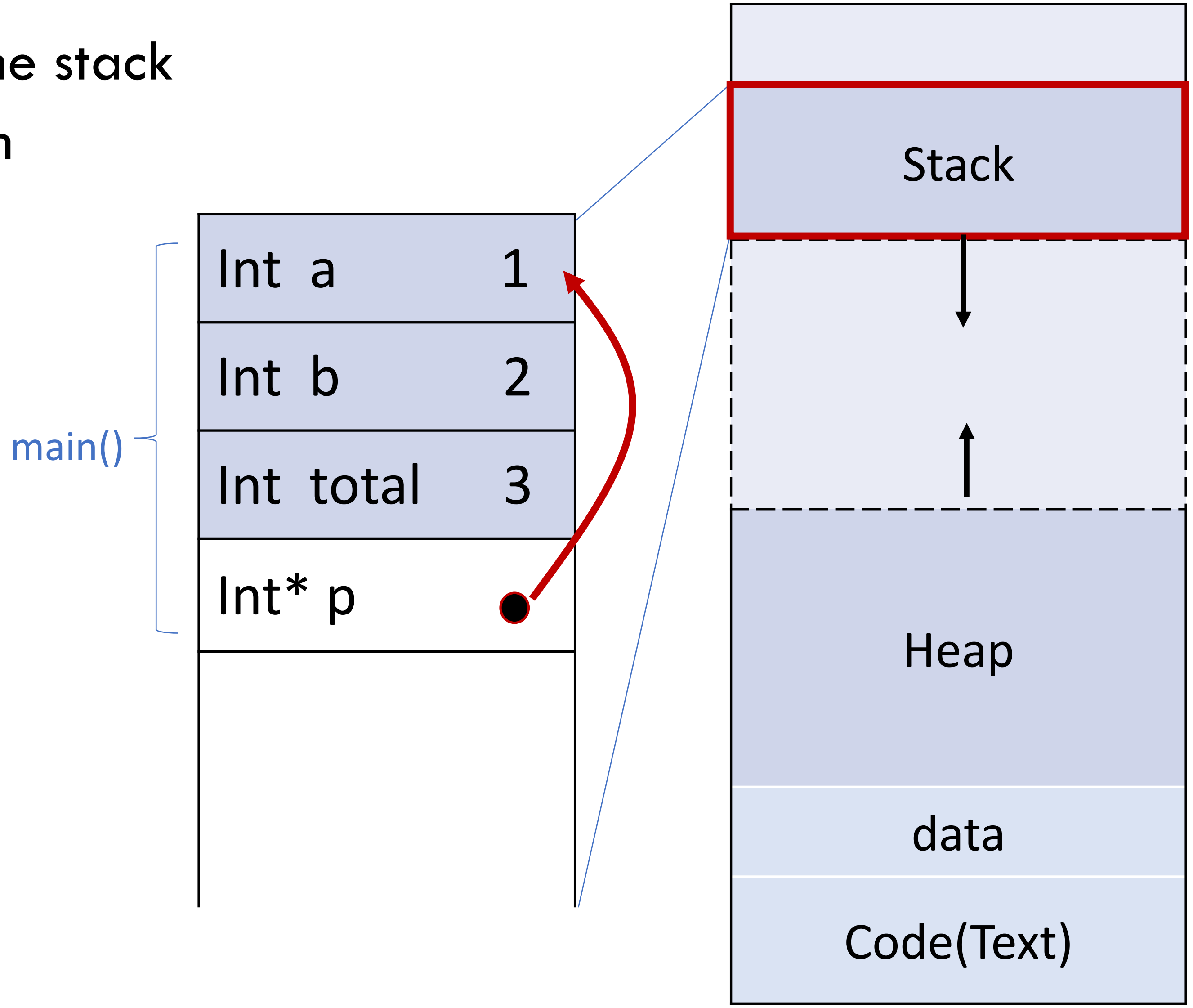
```
int computeA(int a){ return a*a; }

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

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



Data pointer in Stack Memory

- Now, if we take a closer look on the stack memory segment of main() function

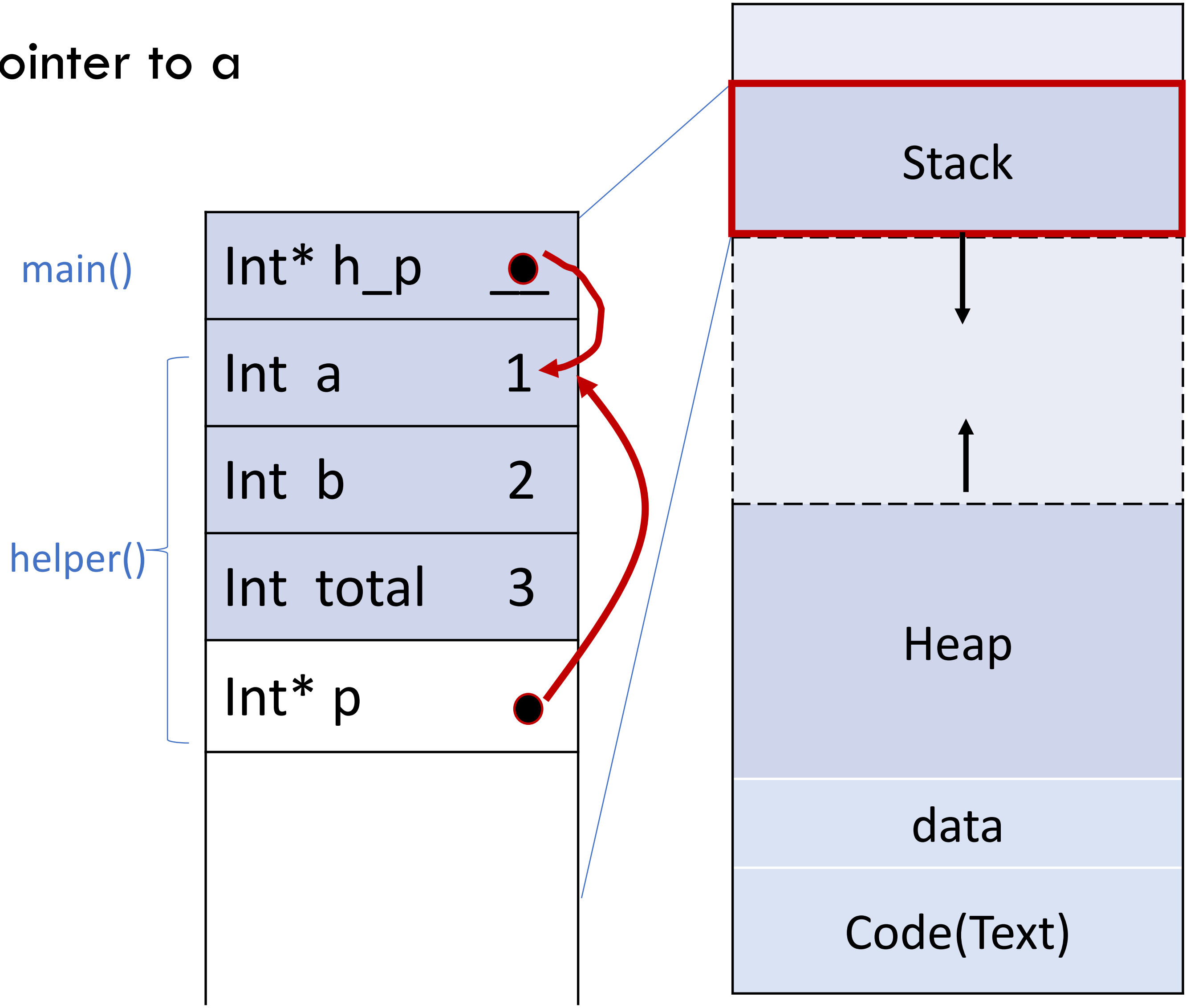
```
int main()
{
    int a = 1, int b = 2;
    total = computeFinal(a, b);
    int * p = a;
    ...
}
```



Common mistake with stack memory

- A common mistake is to return a pointer to a stack variable in a helper function

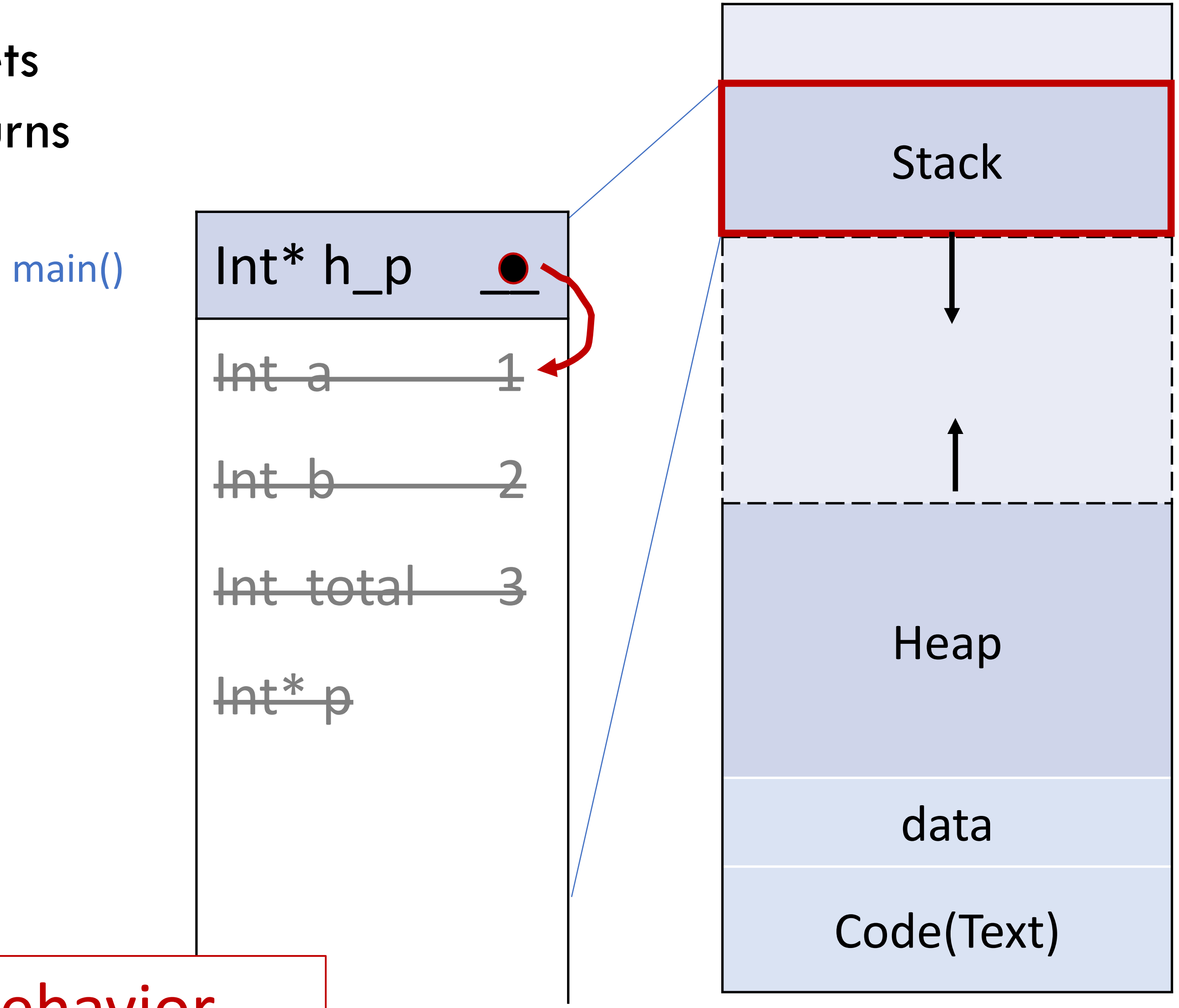
```
int* helper()
{
    int a = 1, int b = 2;
    int * p = a;
    return p;
}

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



Common mistake with stack memory

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

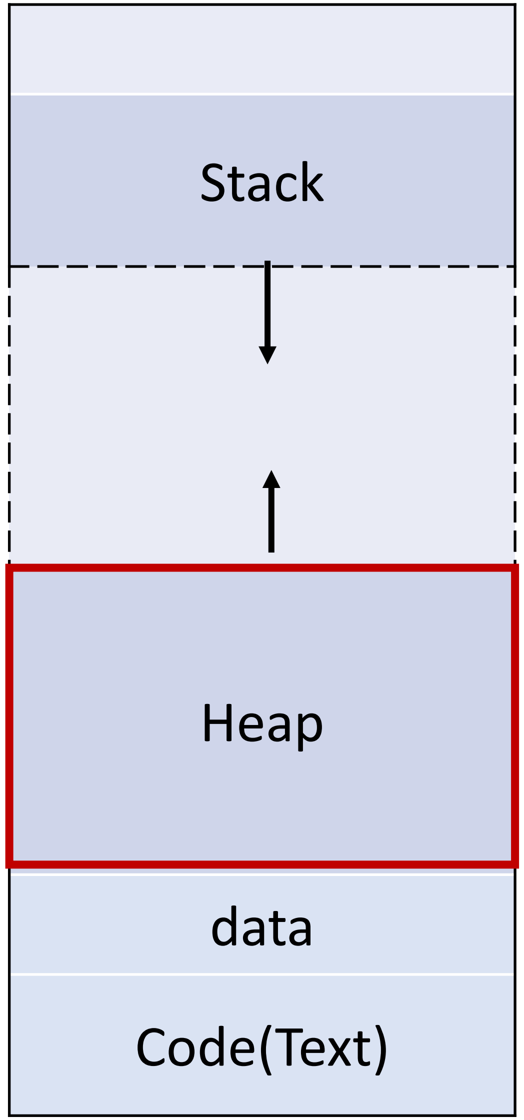
```
int* helper()
{
    int a = 1, int b = 2;
    int * p = a;
    return p;
}

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



Undefined behavior

Heap Memory

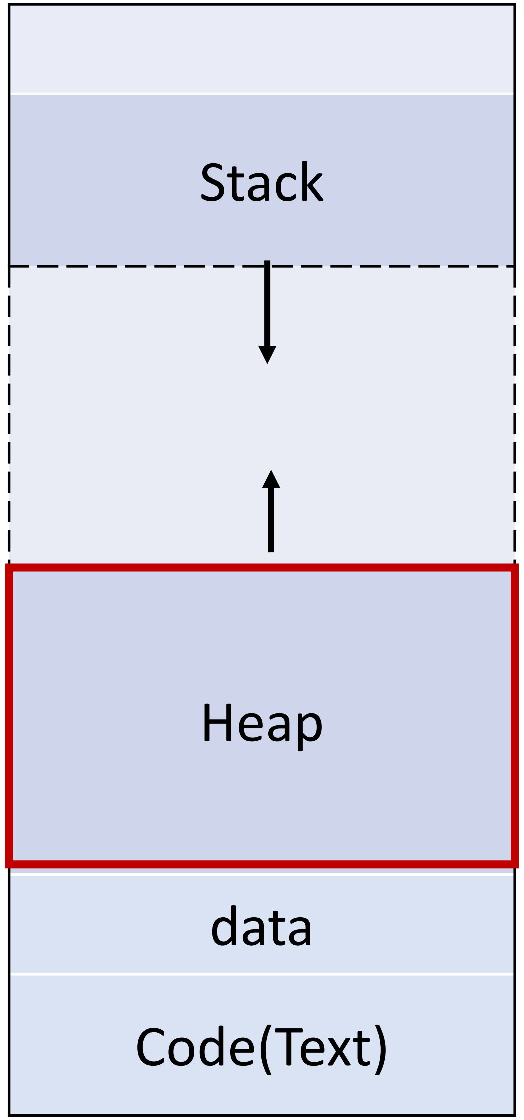
- Heap Allocation
  - Allocated during the execution of instructions written by programmers. (Variables allocated by heap could last longer than the span of the function)




```
int *ptr  = new int[10];          // This memory for 10 integers is
                                    allocated on heap
                                  //  new key word calls malloc()
```

Heap Memory

- Heap Allocation
  - Allocated during the execution of instructions written by programmers.
  - **No automatic de-allocation** feature is provided. Need to use a Garbage collector to remove the old unused objects

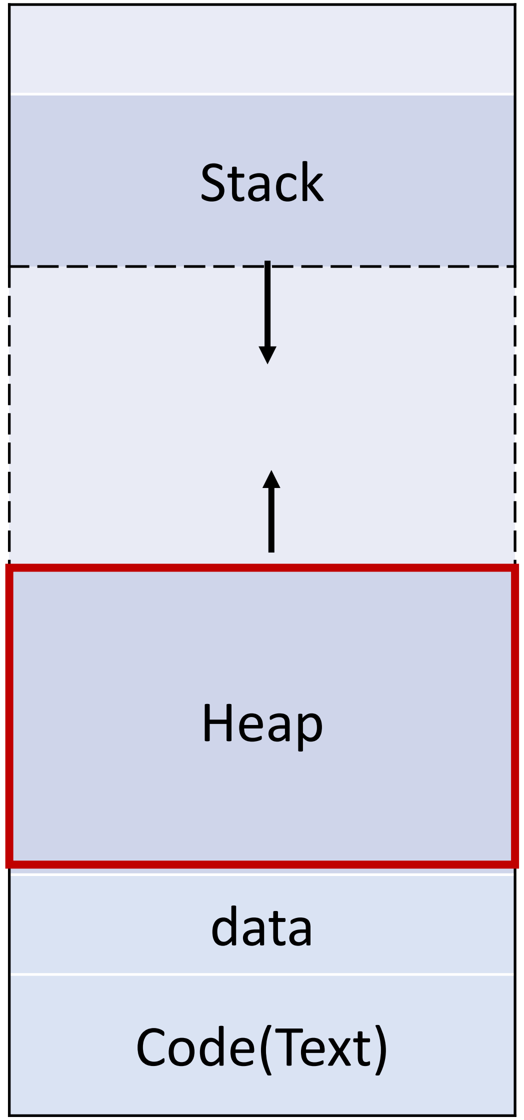



```
int *ptr  = new int[10];
Delete[] ptr;                         //  release the memory
```

Heap Memory

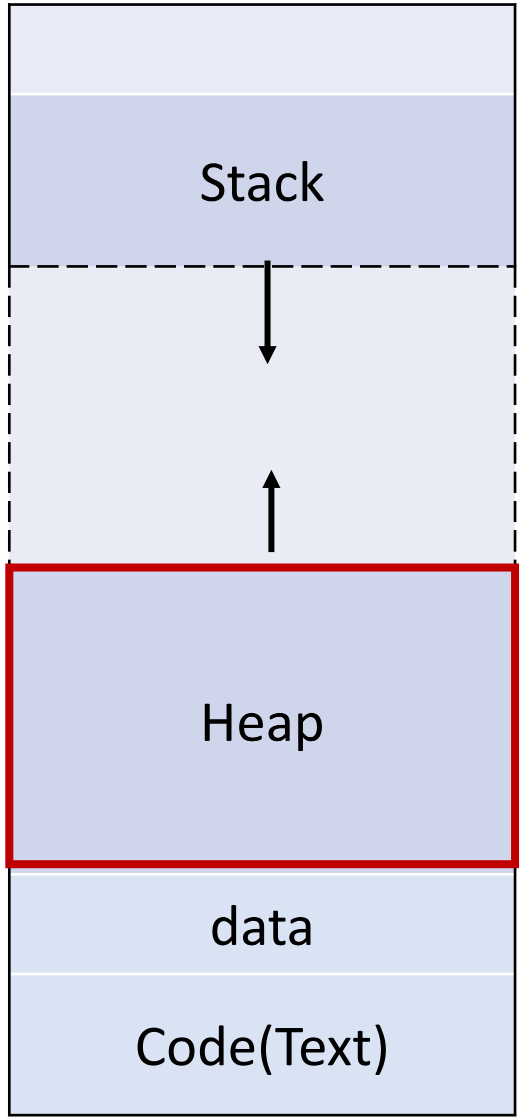
- Heap Allocation
  - Allocated during the execution of instructions written by programmers.
  - No automatic de-allocation feature is provided. Need to use a Garbage collector to remove the old unused objects
  - If you try to use the pointers to the memory after you free them, it will cause **undefined behavior**. (A good practice to set the value of freed pointers to nullptr immediately after delete)

```
int *ptr  = new int[10];
Delete[] ptr;
ptr = nullptr;                    //  set the value of the freed pointer
```



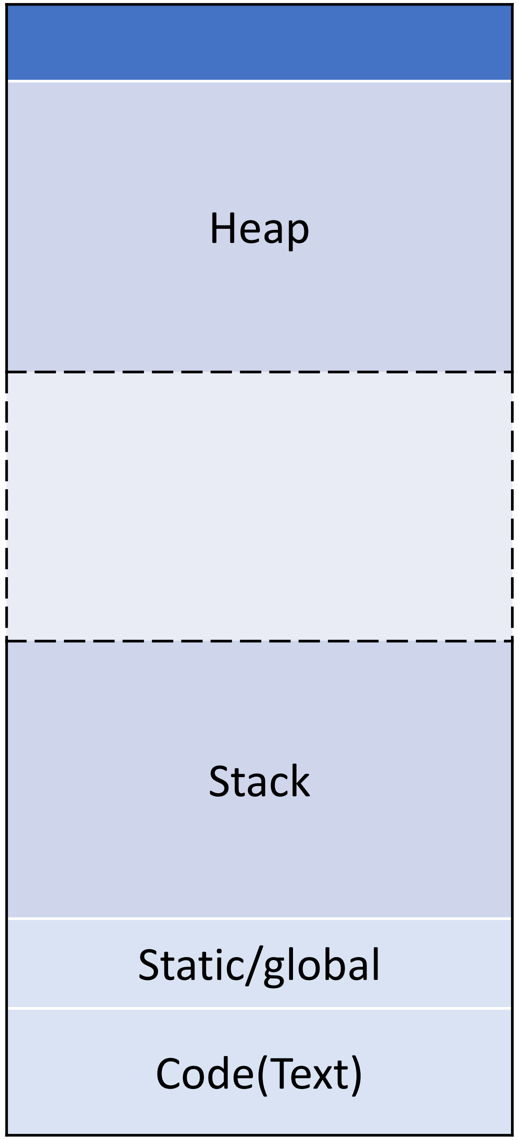
Heap Memory

- Heap Allocation
  - Allocated during the execution of instructions written by programmers. (Variables allocated by heap could last longer than the span of the function)
  - No automatic de-allocation feature is provided. Need to use a Garbage collector to remove the old unused objects
  - If you try to use the pointers to those memory after you free them, it will cause undefined behavior.
  - Unlike stack, memory allocated on heap is **not** necessarily **contiguous**



Memory

- Example demo code of objects allocate memory on Stack, Heap



C++ Pointers and memory

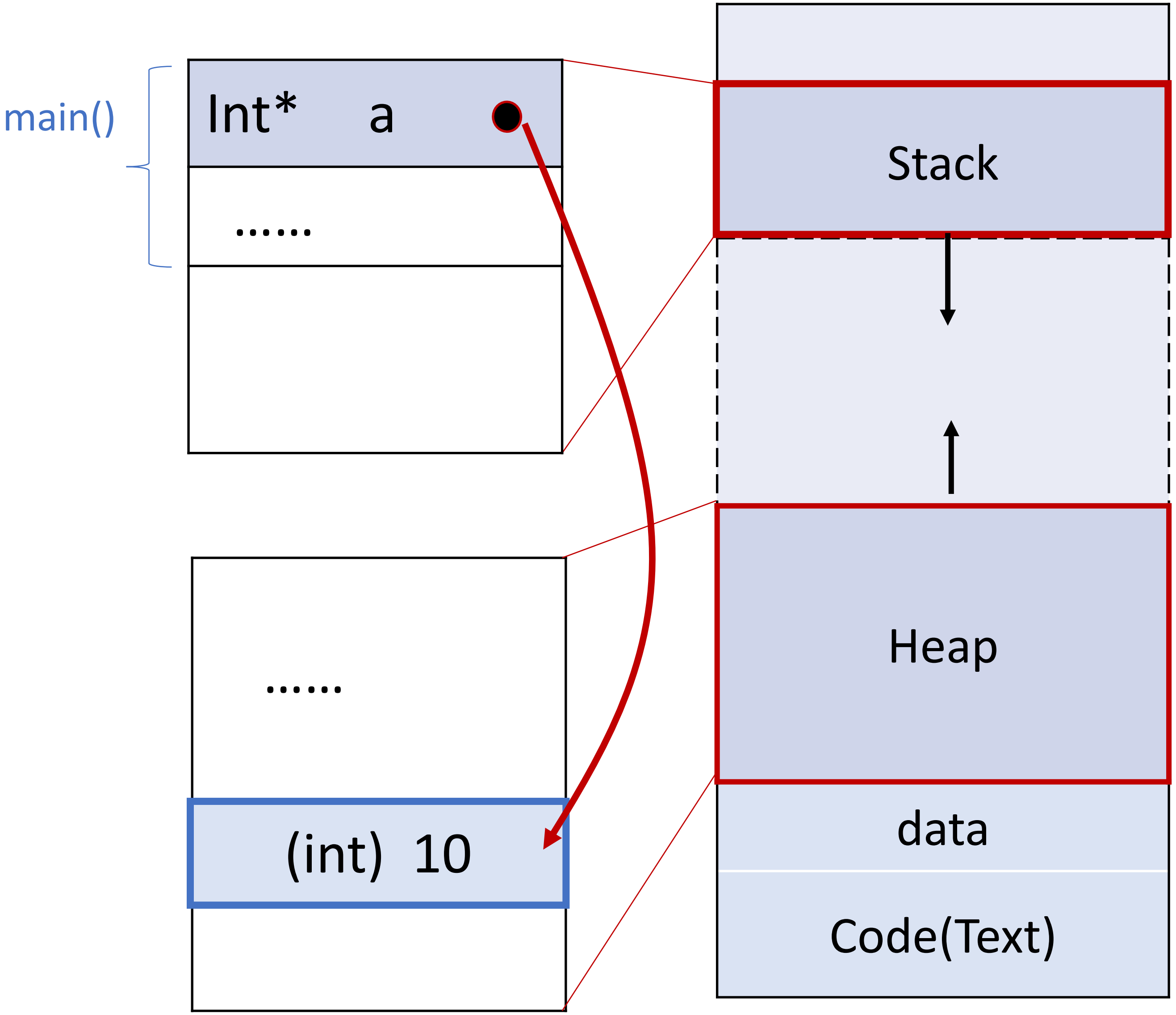
- What are C++ Pointer and Reference? Why do we have them?
- How to use C++ pointers and allocate memory for my program?

Types of Pointers

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

C++ raw pointer with heap-based memory allocation

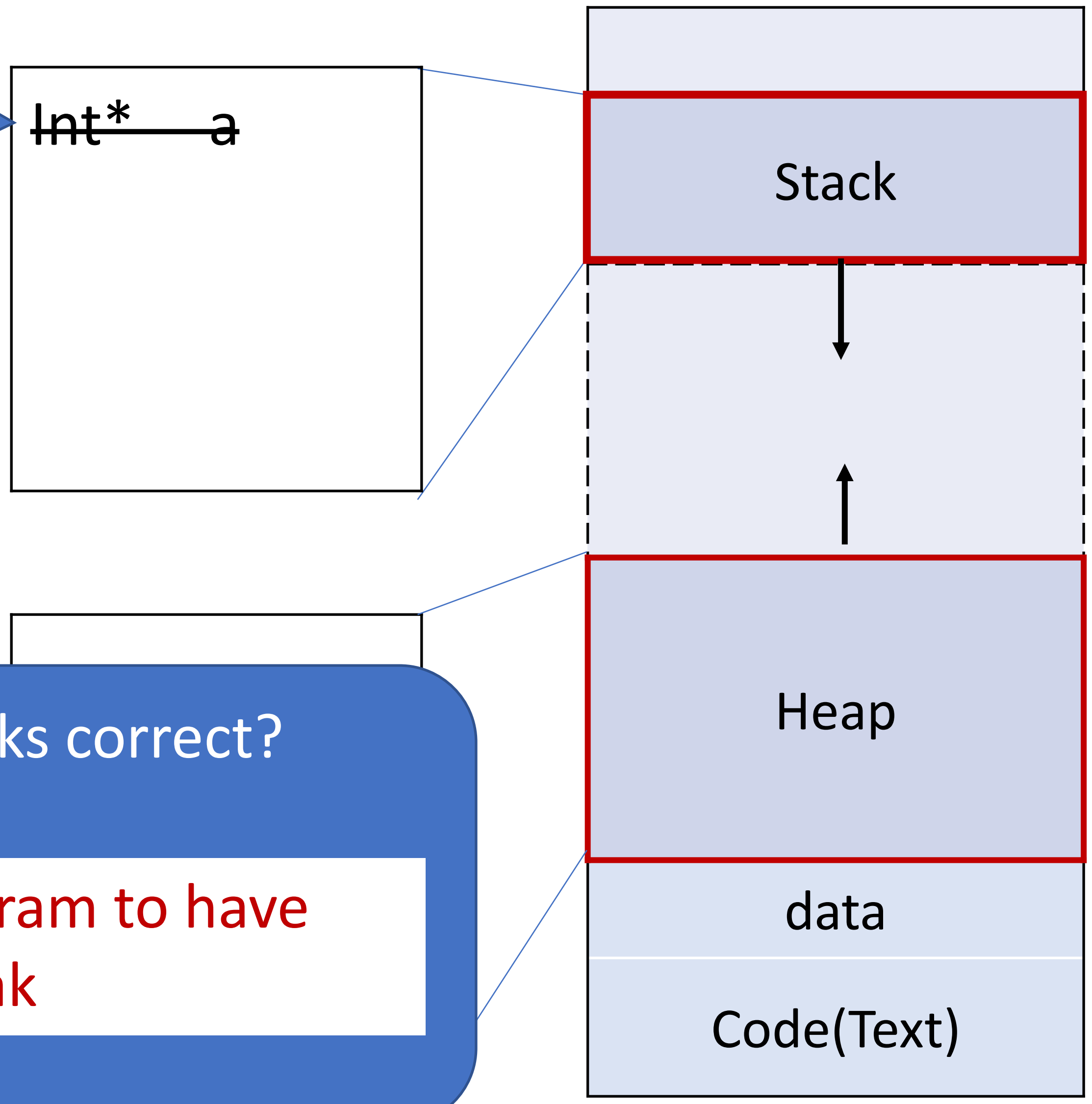
```
#include <iostream>

int main(){
    int* a = new int(10);
    ...


    return 0;
}
```



C++ raw pointer with heap-based memory allocation

```
#include <iostream>

int main(){
    int* a = new int(10);

    return 0;
}
```

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



Does this function looks correct?

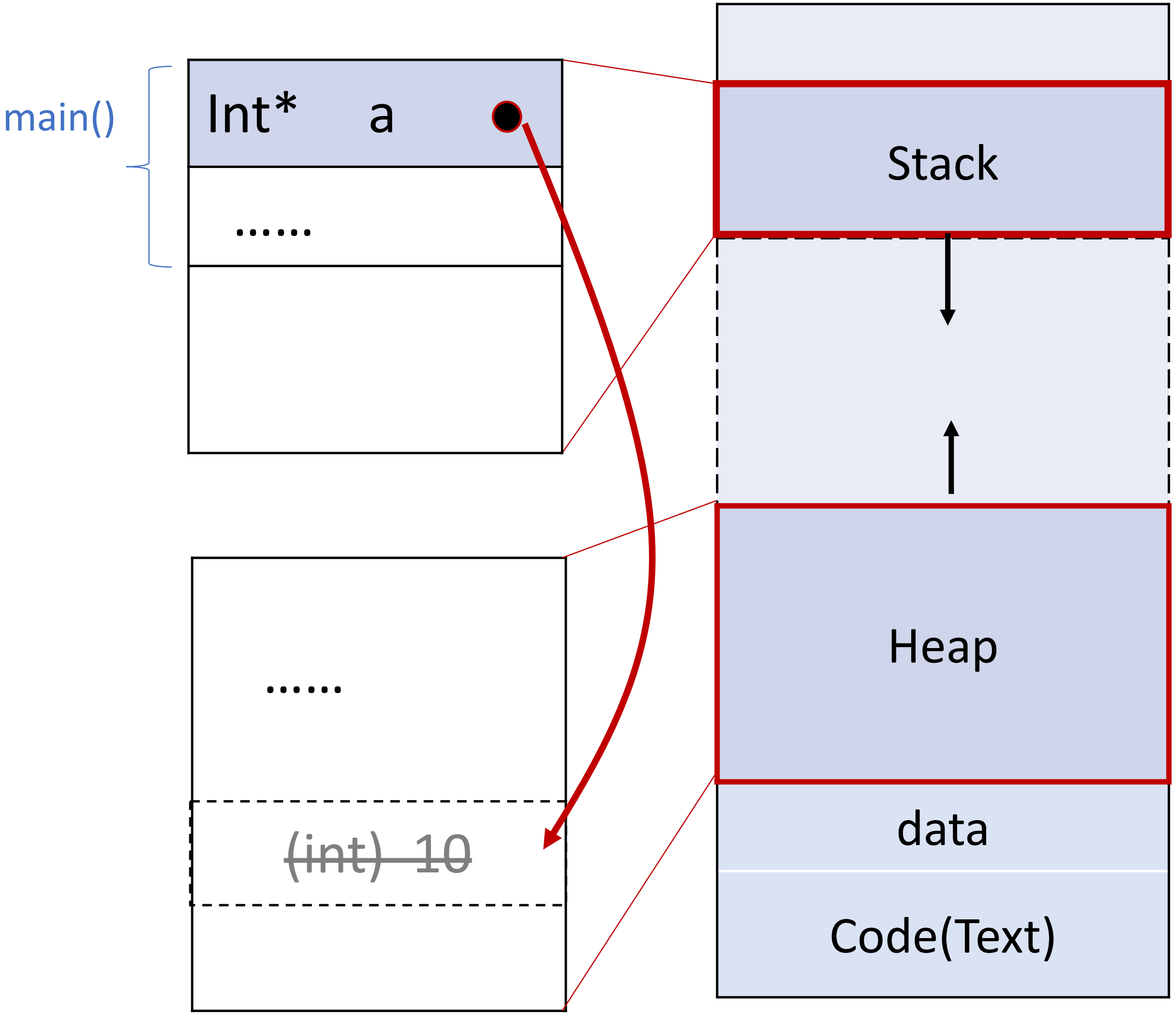
No. It causes the program to have memory leak

C++ raw pointers with heap-based memory allocation

```
#include <iostream>


int main(){
    int* a = new int(10);         // Use the * operator to declare a pointer type
                                  // Use new to allocate and initialize memory on heap

    ...
    delete a;                     // release memory
                                  // anything allocate with new, should delete the memory to
                                  // prevent memory leak
    return 0;
}
```

C++ raw pointer with heap-based memory allocation

```
#include <iostream>

int main(){
    int* a = new int(10);

    ...
    delete a;
    return 0;
}
```



C++ Raw Pointer

```
Example* example = new Example();
```

// Use the * operator to **declare a pointer type**
// Use new to allocate and initialize memory

What if never call delete example?

It will cause the program to have memory leak

```
delete example;
```

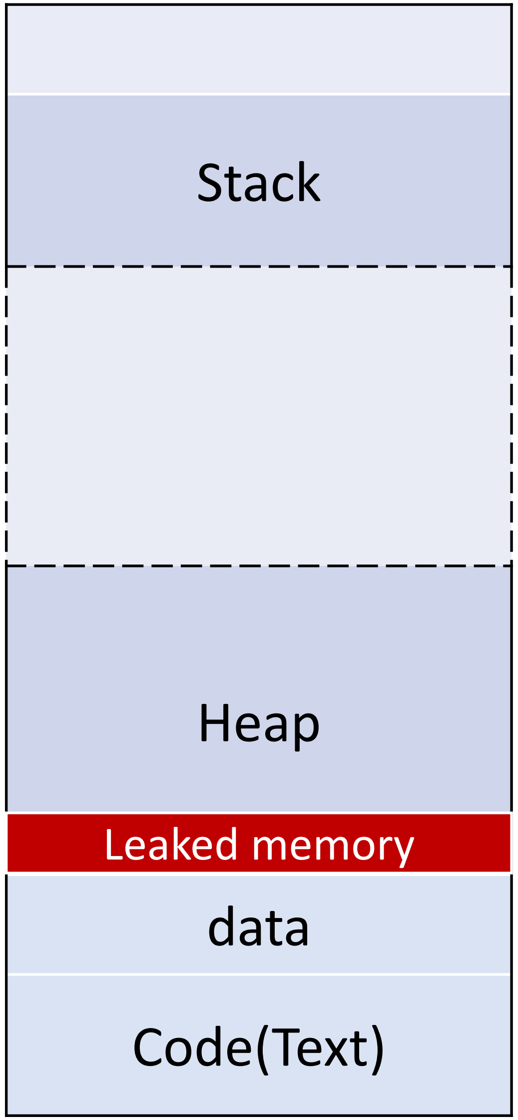
// **release memory** back to OS
// anything allocate with new, should delete the memory to prevent memory leak

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
  - The problem with memory leaks is that they accumulate over time and, if left unchecked, may cripple or even crash a program

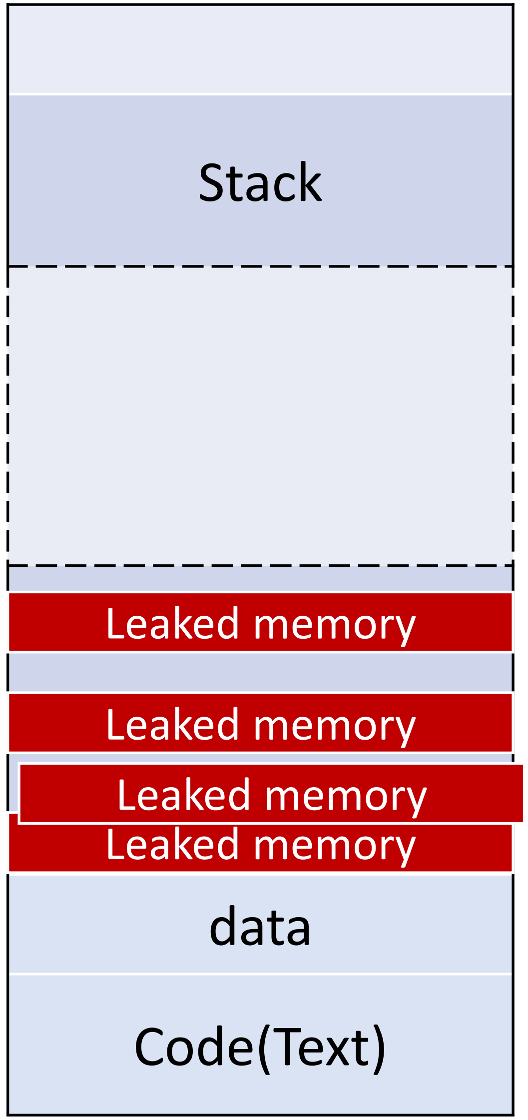
Memory Leak

- 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
  - The problem with memory leaks is that they accumulate over time and, if left unchecked, may cripple or even crash a program



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
  - The problem with memory leaks is that they accumulate over time and, if left unchecked, may cripple or even crash a program



Memory Leak

- What is memory leak in C++?
- How to avoid memory leak in my program?
  - Follow **RAII principle**(Resource acquisition is initialization): resource acquisition must succeed for initialization to succeed. The resource is guaranteed to be held between when initialization finishes and finalization starts, and be released when not used.
  - Use **smart pointers** instead of raw pointers

Memory Leak

demo

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

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

C++ Raw Pointer

```
Example* example2 = new Example();
```

// Use the * operator to **declare a pointer type**
// Use new to allocate and initialize memory

```
Example* ecopy = example2;
```

// **Declare a pointer that points to an object** using the address of operator

Undefined behavior

```
ecopy->print();
```

// **Accessing filed/function** of an object's pointer using ->

```
delete example2;
```

// **Dangerous behavior**, leaving a dangling pointer ecopy

C++ Raw Pointer

```
Example* example2 = new Example();

Example* ecopy = example2;

ecopy->print();

delete example2;

ecopy->print();
```

What happen if
I try to access example2 later in my code?

Undefined behavior

// **Dangerous behavior,** leaving a dangling pointer ecopy

// **Undefined behaviour,** the object pointed by ecopy is deleted

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
- Iterators

Ownership of Pointers

- For C++ ownership is the responsibility for cleanup.
- The three types of pointers:
  - int * : 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.
  - std::unique_ptr<int>: represents the simplest form of ownership (sole owner of resource and will get destroyed and cleaned up correctly)
  - std::shared_ptr<int> : one of a group of friends who are collectively responsible for the resource. The last of them to get destroyed will clean it up.

Types of Pointers

--- smart pointer: unique_ptr

- a smart pointer that owns and manages an object through a pointer and disposes of that object when the unique_ptr goes out of scope.

```
std::unique_ptr<Example> example = new Example();
```

✗ Unique_ptr needs to call the constructor explicitly

```
std::unique_ptr<Example> example(new Example());
std::unique_ptr<Example> example = std::make_unique<Example>();
```

✓ ✓

```
std::unique_ptr<Example> example2 = example;
```

✗ unique_ptr class doesn't allow copy of unique_ptr

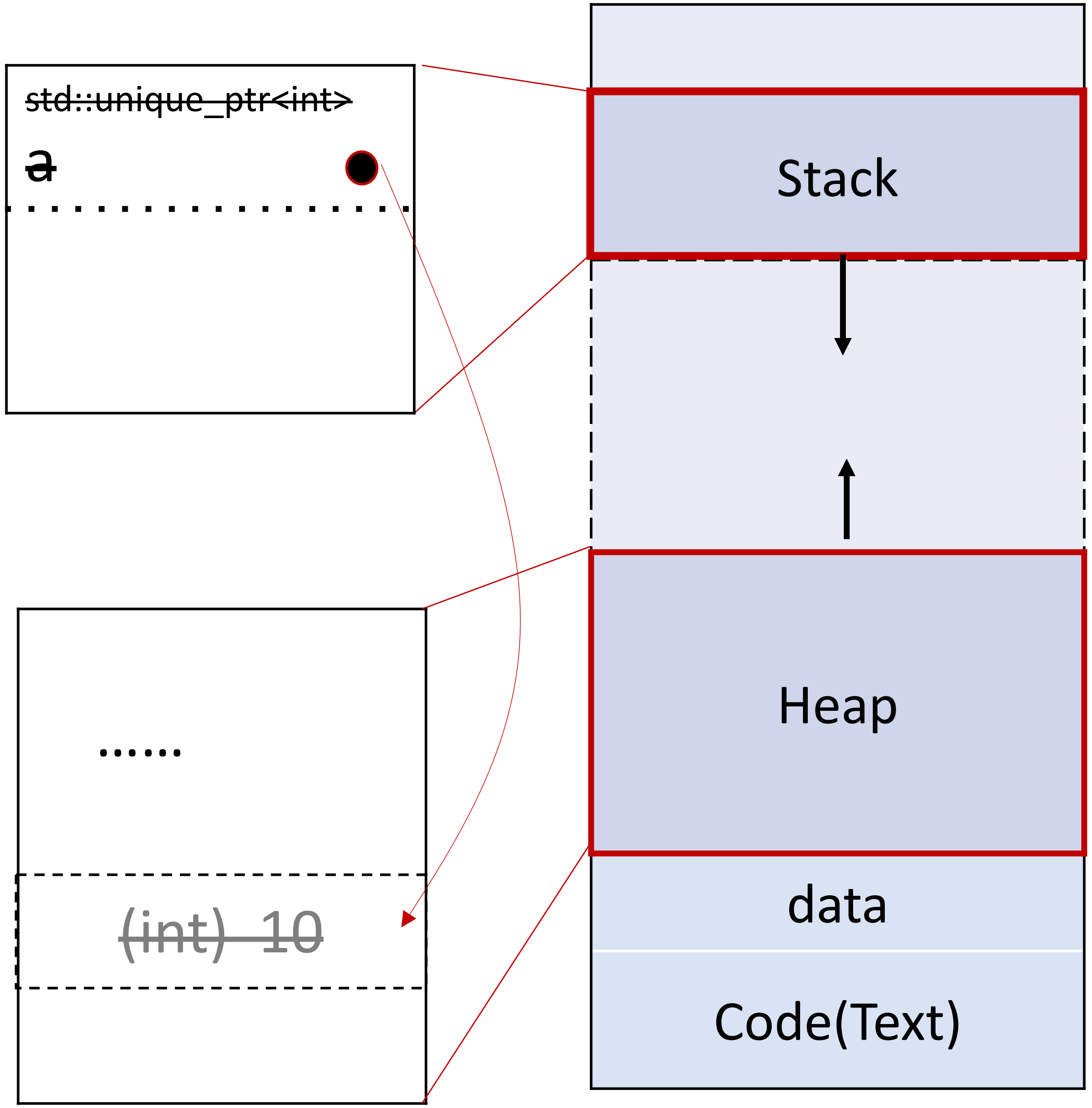
```
std::unique_ptr<Example> example2 = std::move(example);
```

✓

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

Exercise: std::vector of pointers

```
#include <iostream>

int main(){
    std::unique_ptr<int> a =
std::make_unique<int>(10);
    …
    return 0;
}
```



Types of Pointers

--- 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<Example> example = std::make_shared<Example>();    ✓

std::shared_ptr<Example> example(new Example());    ✓

std::shared_ptr<Example> example2 = example;
```

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 : prefer, low overhead
  - shared_ptr
- Array Pointer, Iterators

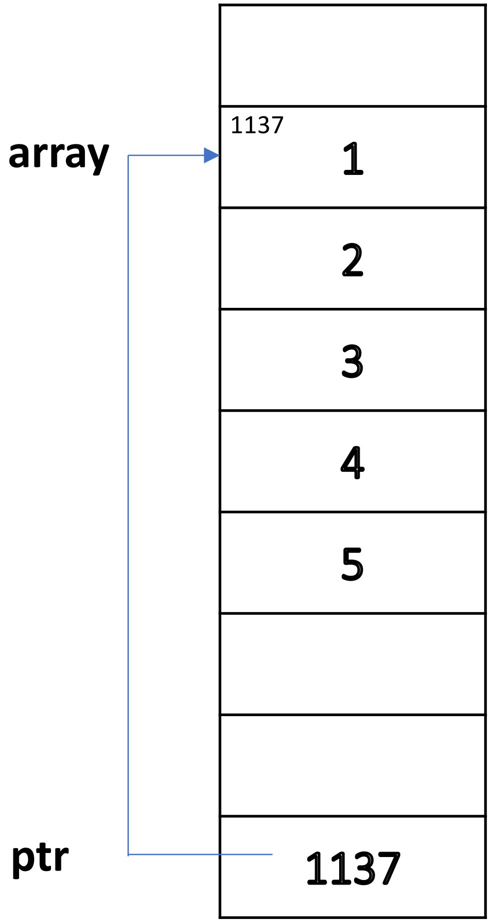
Types of Pointers

--- array pointer

- An array name is a pointer to the first element of the array
- *(array + ind) is equivalent to array[ind]

```
int array[5] = {1, 2, 3, 4, 5};
int* ptr;
ptr = array;
cout << *(array + 3) << endl;
cout << *(ptr + 3) << endl;
```

What are the print outs?



Types of Pointers

--- vector pointer

- **Vector pointer:** a direct pointer to the memory array by the vector to store its elements.
- Buggy code example:

```
std::vector<int> intVector;

intVector.push_back(1);

int* pointerToInt = &intVector[0];                // We get the pointer to the first element from our vector.
```

?

Types of Pointers

--- vector pointer

- **Vector pointer:** a direct pointer to the memory array by the vector to store its elements.
- Buggy code example:

```
std::vector<int> intVector;

intVector.push_back(1);

int* pointerToInt = &intVector[0];          // We get the pointer to the first element from our vector.

 intVector.push_back(2);
                                             // Add two more elements to trigger vector resize. During
                                             // resize the internal array is deleted causing our pointer
 intVector.push_back(3);                     // to point to an invalid location.

std::cout << "The value of our int is: " << *pointerToInt << std::endl;
```

?

Types of Pointers --- vector iterator

- **Iterator:** An iterator is an object (like a pointer) that points to an element inside the container.
- **Container**: A container is a holder object that stores a collection of other objects (its elements). Like array, vector, dequeue, list …
- **Difference** between pointer and iterator:
  - An iterator may hold a pointer, but it may be something much more complex. (e.g. iterator can iterate over data that's on file system, spread across many machines.)
  - An iterator is more restricted, can only refer to object inside a container (e.g. vector, array) . A pointer of type T* can point to any type T object.

Types of Pointers

--- vector pointer and iterator

- vector<T>::iterator i: create an iterator for a vector of type T
- begin() : return the beginning position of the container
- end() : return the after end position of the container
- To access the elements in the sequence container by i++

```
std::vector<int> myvector;

For(int i=1; i<5 ; i ==) myvect.push_back(i) ;

for (std::vector<int>::iterator it = myvector.begin() ; it != myvector.end(); ++it)

   std::cout << ' ' << *it << std::endl;
```

C++ Functions

- What are C++ Pointer and Reference? Why do we have them?
- How to use C++ memory resources for my program?

Function Parameter

- Pass by value : passing the **copy of the value**

```
void fun(X x) { std::cout << x << std::endl; };      // declare a function
X x;                                                 // create a variable
fun(x);                                              // call the function
```

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

```
void fun(X *x);
X x;
fun(&x);                                   // & means get the address_of
```

- Pass by reference : passing a **reference**

```
void fun(X &x);                    // & means the parameter type is reference
X x;
fun(x);
```

Function Parameter

--- Passing vector

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

```
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);
}
```

← Passing a vector value to a function:

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

Function Parameter

--- Passing vector

demo

- Pass by reference

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

--- Passing vector

demo

- Pass by reference

  (preferred to pass by reference than pass by pointer)

```
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

--- const

demo

- Const keyword in parameter of **reference**: a promise that the variable being referenced **cannot** be changed through the reference.

```
void foo(const std::string& x) // x is a const reference
{
        x = "hello"; // compile error: a const reference cannot have its value changed!
}
```

Function Parameter

--- const

- Const keyword in parameter of **pointer**:

  const type * identifier;   // define a read-only location

  - declares the identifier as a pointer whose pointed at value is constant. This construct is used when pointer arguments to functions will not have their contents modified.

```
void fn(const int* p){
        *p = expression;      // compiler complain: here it is illegal to have
                              a const pointer's content change
}
```

Function Parameter

--- const

- Const keyword in parameter of **pointer**:

  type * const identifier;   // define a read-only parameter

  - declares the identifier as a const pointer whose memory address it points to cannot be changed.

```
void fn(int* const p){
        int a = 5;
        p = &a;
}
```

// compiler complain: here it is illegal to have a const pointer parameter changed

Const vs constexpr

- **const** declares an object as constant. This implies **a guarantee** that once initialized, the value of that object won't change.
- A **constexpr** variable or function must return a literal type. (A literal type is one whose layout can be determined at compile time. The following are the literal types:
  - void
  - scalar types
  - references
  - Arrays of void, scalar types or references
  - A class that has a trivial destructor
- **const** variable can be deferred until run time. A **constexpr** variable must be initialized at compile time. All **constexpr** variables are **const**.

```
constexpr float x = 42.0;
constexpr float y{108};
constexpr float z = exp(5, 3);
constexpr int i;             // Error! Not initialized
int j = 0;
constexpr int k = j + 1;    //Error! j not a constant expression
```

Const vs constexpr

- A **constexpr** function is one whose return value is computable at compile time when consuming code requires it.
  - A constexpr function must accept and return only literal types.
  - A constexpr function can be recursive.

- A **constexpr** function or constructor is implicitly inline.

```
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
}
```

Function Returns

- Return by value : returning a copy of the value

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

- Return by reference

```
double& getValue( int i ) {
        return vals[i];      // return a reference to the ith element
}
```

demo

Function Returns

- Return by value
- Return by reference
- Return a pointer :
  - Generally not a good idea to return a pointer to a local variable

```
class person{
public:
    std::string name;
    int id;
    std::string hobby;
    person(std::string _name, int _age, std::string
_hobby)
        : name(_name), id(_age), hobby(_hobby){}

};
```

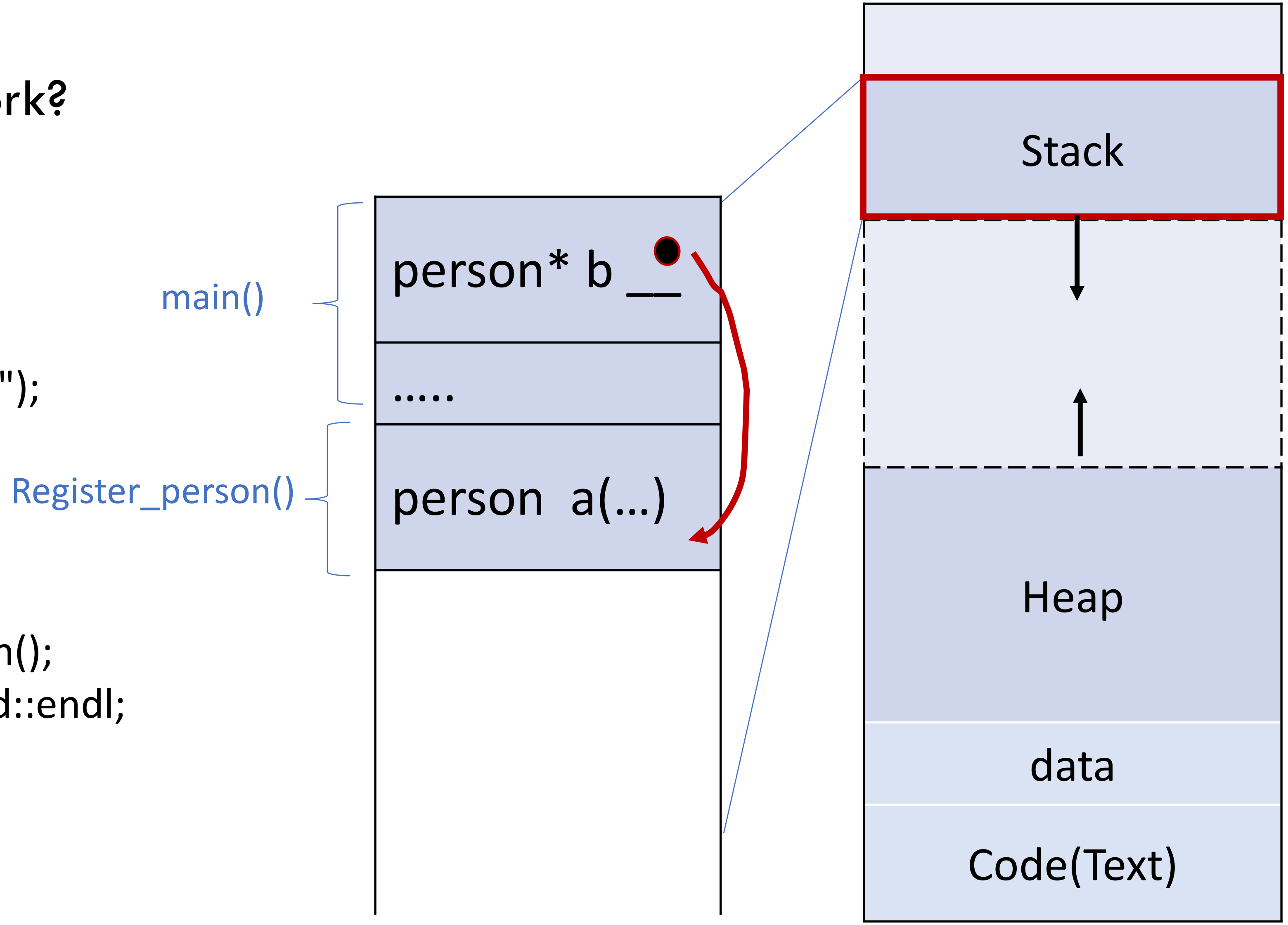
```
person* register_person(){
    person a("alicia", 1, "chess");
    return &a;
}

int main(){
    person* b = register_person();
    std::cout << b->name << std::endl;
    delete b;
...
}
```

✗

Memory

- Why this code doesn't work?

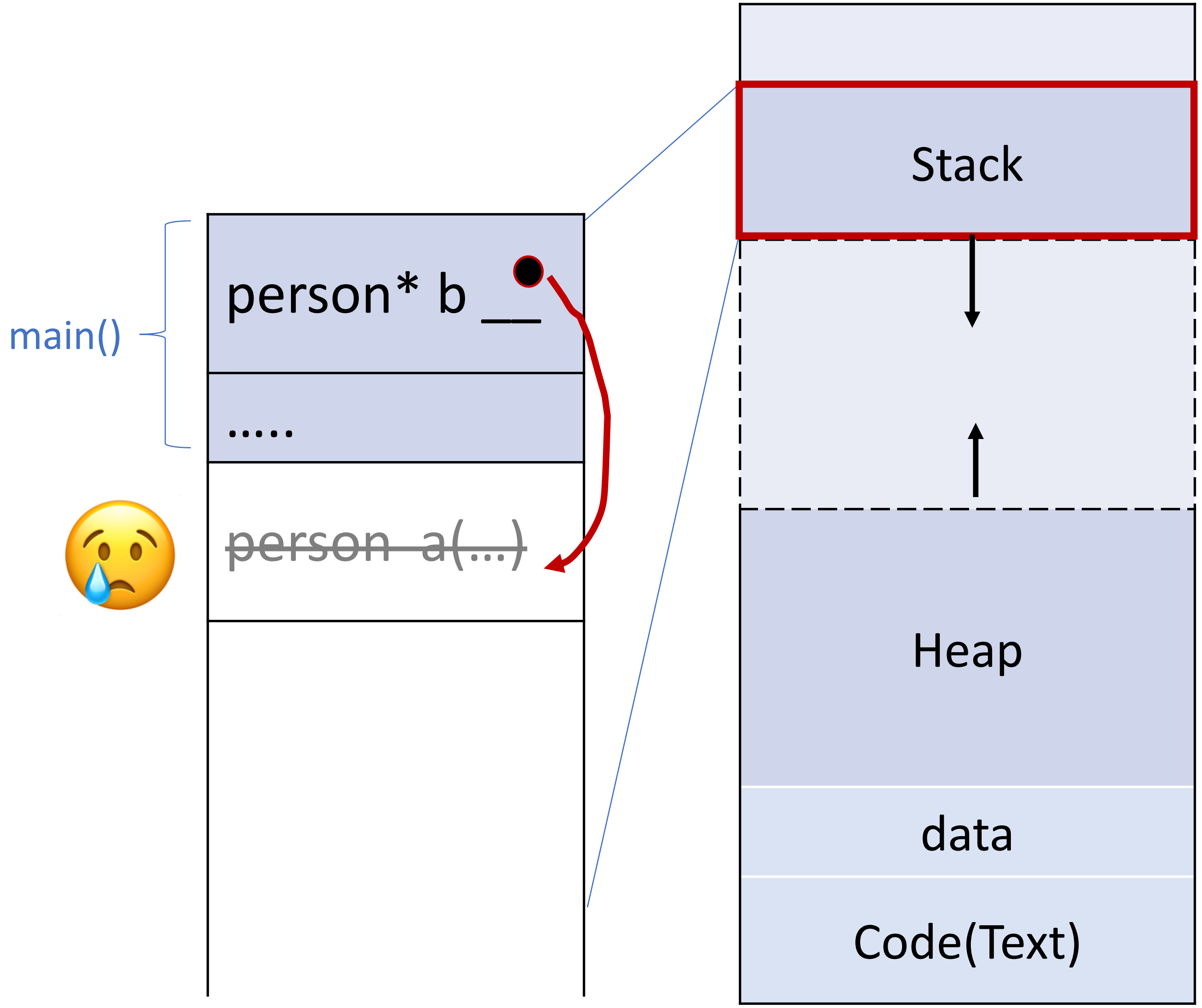
```
person* register_person(){
     person a("alicia", 1, "chess");
     return &a;
}

int main(){
    person* b = register_person();
    std::cout << b->name << std::endl;
    delete b;
...
}
```



Memory

- Why this code doesn't work?

```
person* register_person(){
      person a("alicia", 1, "chess");
      return &a;
}

int main(){
     person* b = register_person();
     std::cout << b->name << std::endl;
     delete b;
…
}
```



Function Returns

--- array

demo

- Return by value
- Return by reference
- Return a pointer
  - Generally not a good idea to return a raw pointer

Can you think of better ways?

Fix1. return by value

```
person register_person(){
    person a("alicia", 1, "chess");
    return a;
}
```

Fix2. use heap (not suggested)

```
person* register_person(){
    person* a = new person("alicia", 1, "chess");
    return a;
}
// (need the caller to release the memory of the returned pointer)
```

Exercise

From the demo examples in fn_return_example.cpp file.

What are some better solutions to return an object that is allocated on heap memory?

Try it out and explain why it works

Where to find the resources?

- Memory Heap and Stack: https://courses.engr.illinois.edu/cs225/fa2022/resources/stack-heap/
- Pointers: https://docs.microsoft.com/en-us/cpp/cpp/pointers-cpp?view=msvc-160 , https://www.cplusplus.com/doc/tutorial/pointers/
- Variable linking at compiler: https://www.cs.csub.edu/~melissa/cs350-f15/notes/notes05.html
- Move semantics: https://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html
- Iterators: https://www.geeksforgeeks.org/introduction-iterators-c/
- difference between pointers: https://www.geeksforgeeks.org/difference-between-iterators-and-pointers-in-c-c-with-examples/
- Passing arguments by reference: https://www.learncpp.com/cpp-tutorial/passing-arguments-by-reference/
- Const vs constexpr: https://learn.microsoft.com/en-us/cpp/cpp/constexpr-cpp?view=msvc-170
- Effective C++: 55 specific ways to improve your programs and designs, Scott Meyers, 3rd edition
- A Tour of C++, Bjarne Stroustrup