CS4414 Recitation 6

C++ memory management and functions

03/03/2023

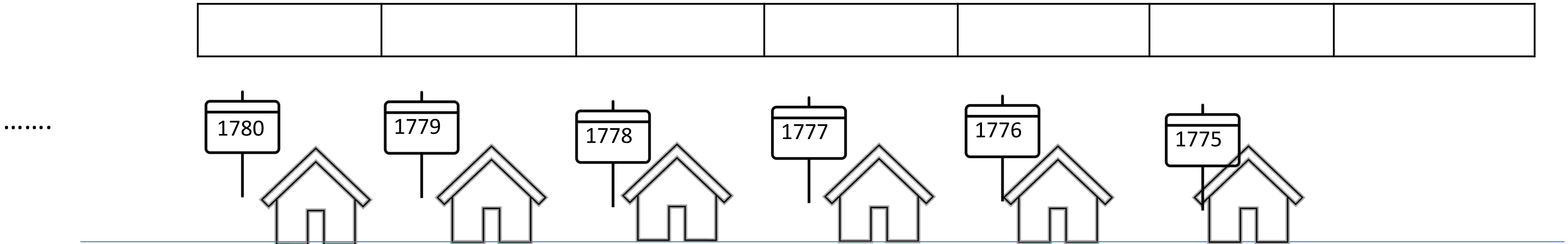
Alicia Yang

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

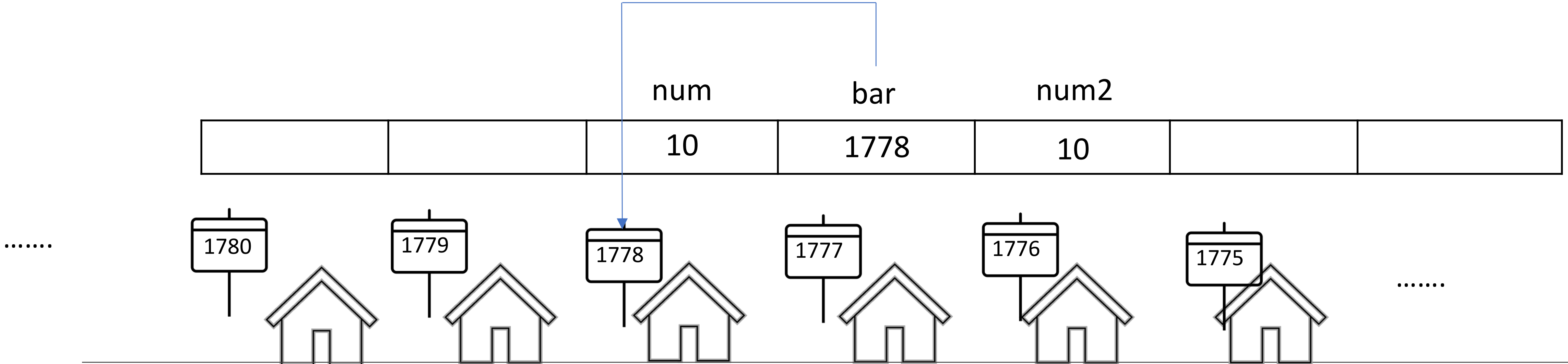
--- Address-of(&) and Dereference(*__) operators

- 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?



		num	bar	num2		
		10	1778	10		

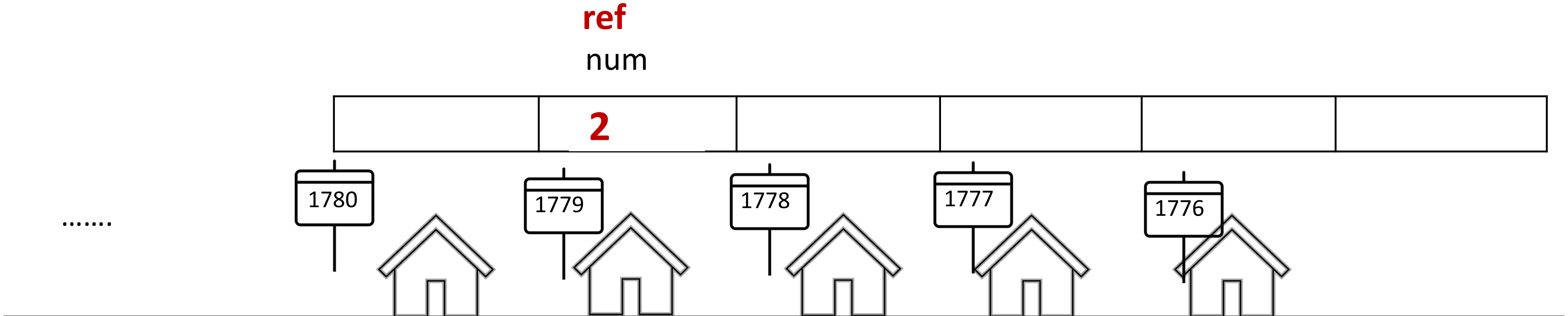
References

MAYBE IF I WEAR THIS SHIRT
NO ONE WILL BE ABLE TO TELL I'M SPIDERMAN

- 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;
```

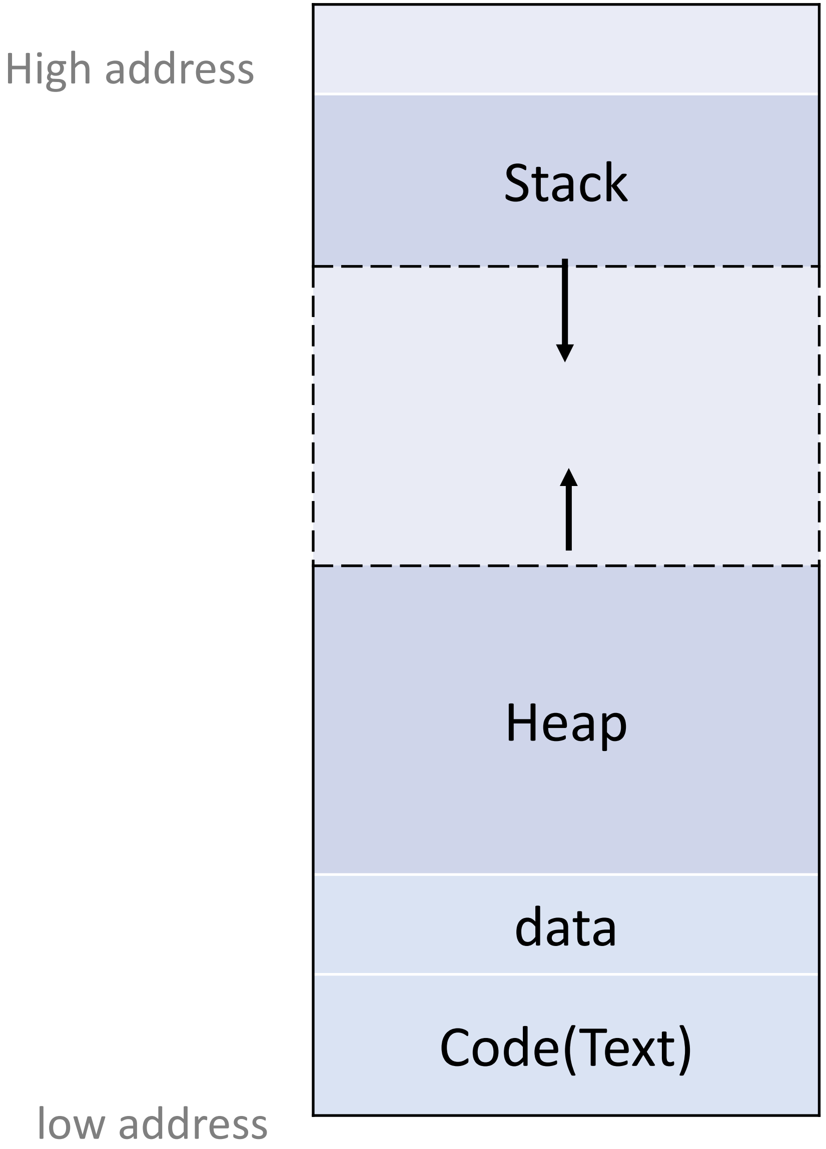
I'm a reference



C++ Memory

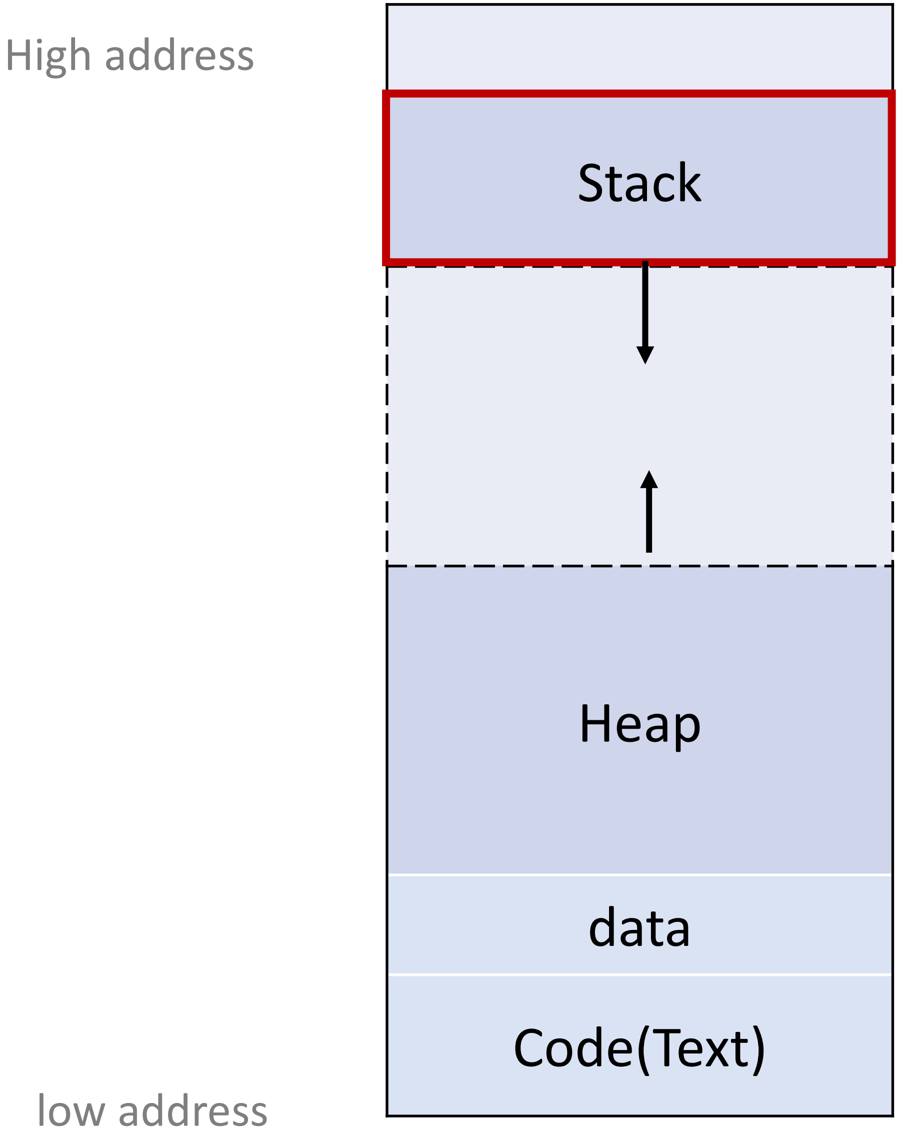
Memory

- 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

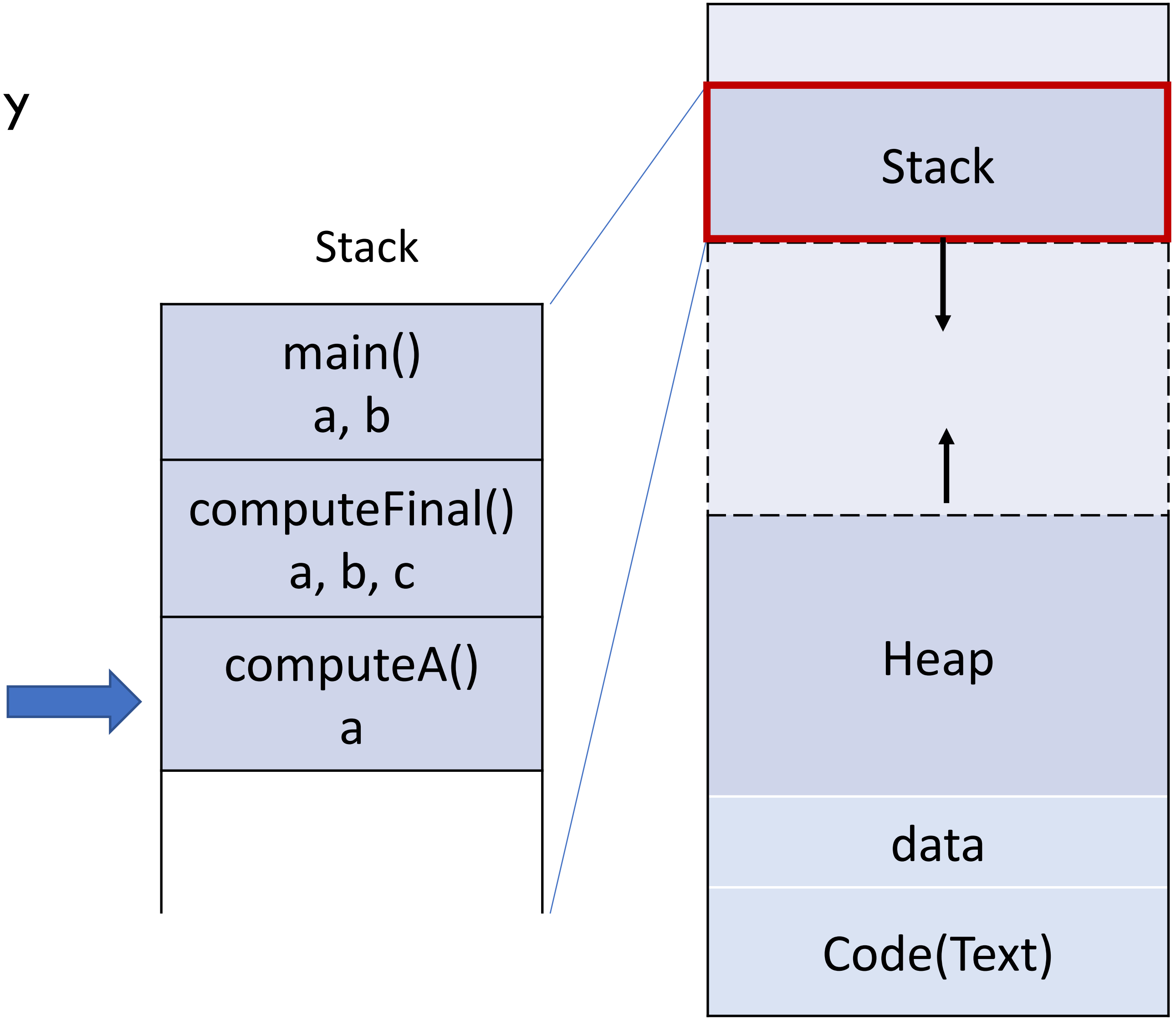
- Stack Allocation (Temporary memory allocation):
  - Allocate on contiguous blocks of memory, in a fixed size
  - Allocation happens in function call stack
  - When a function called, its variables got allocated on stack; when the function call is over, the memory for the variables is deallocated. (scope)
  - 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):

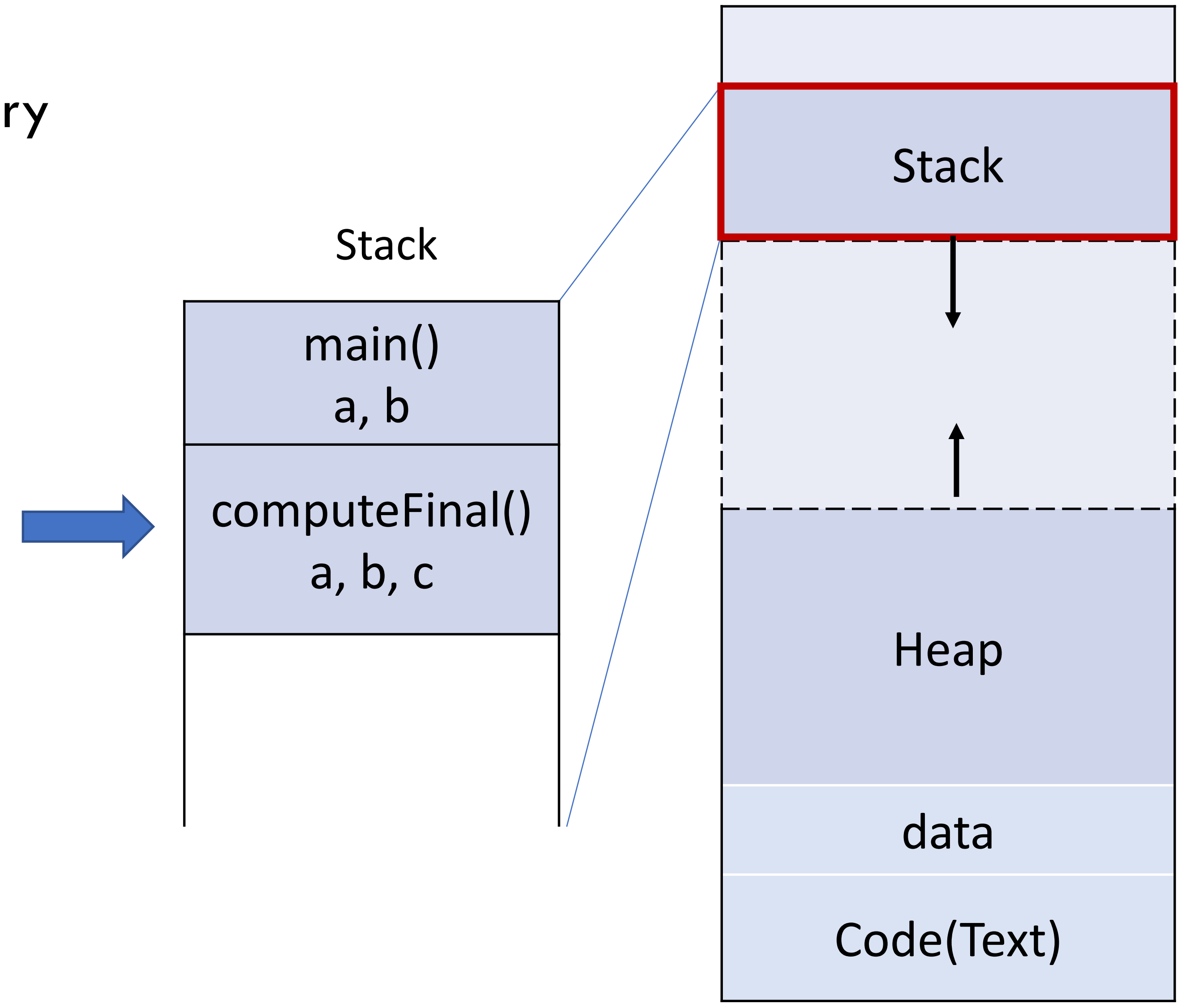
```
int computeA(int a){ return a*a; }

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);
    …
}
```

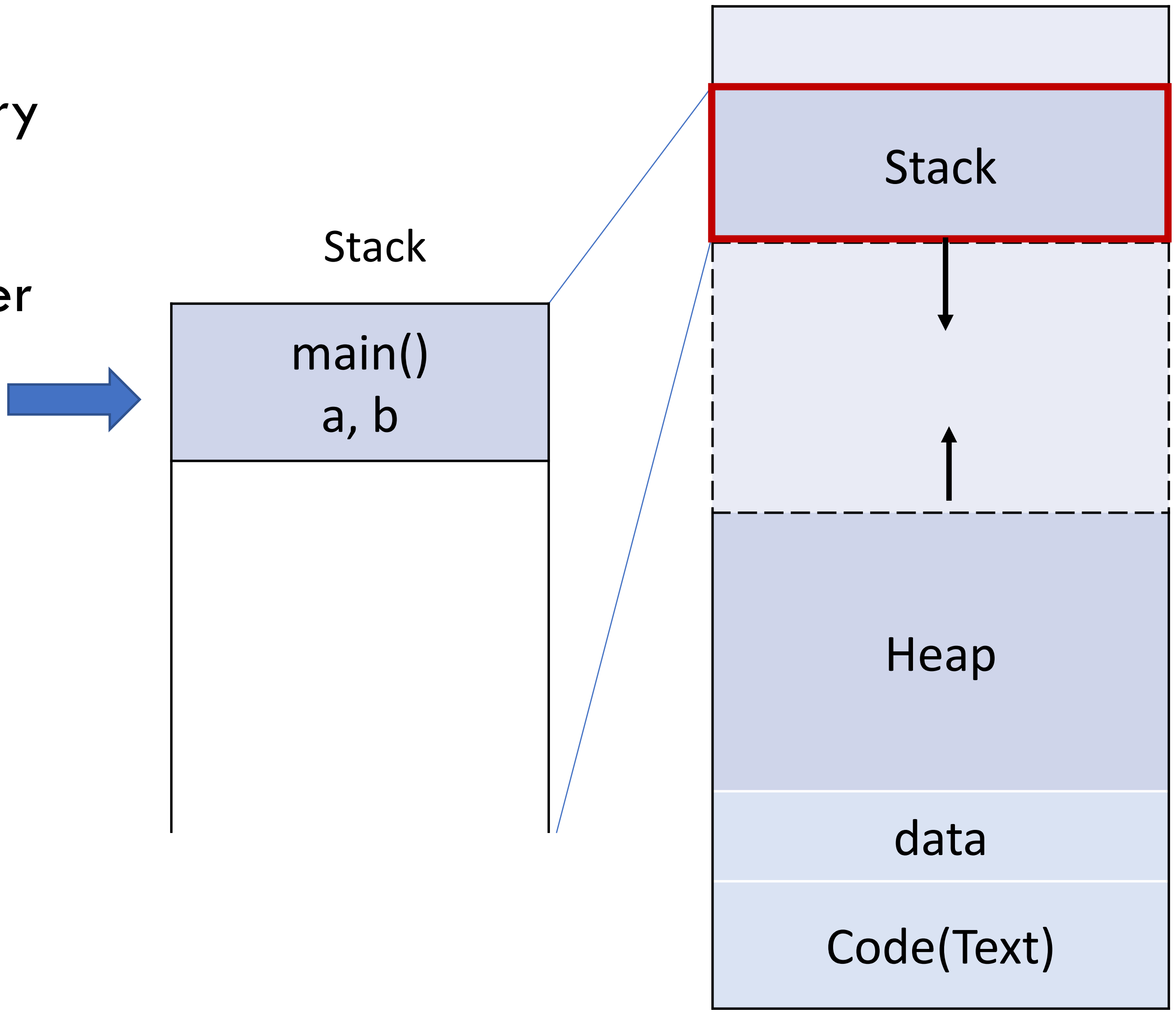


Stack Memory

- 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);
    …
}
```

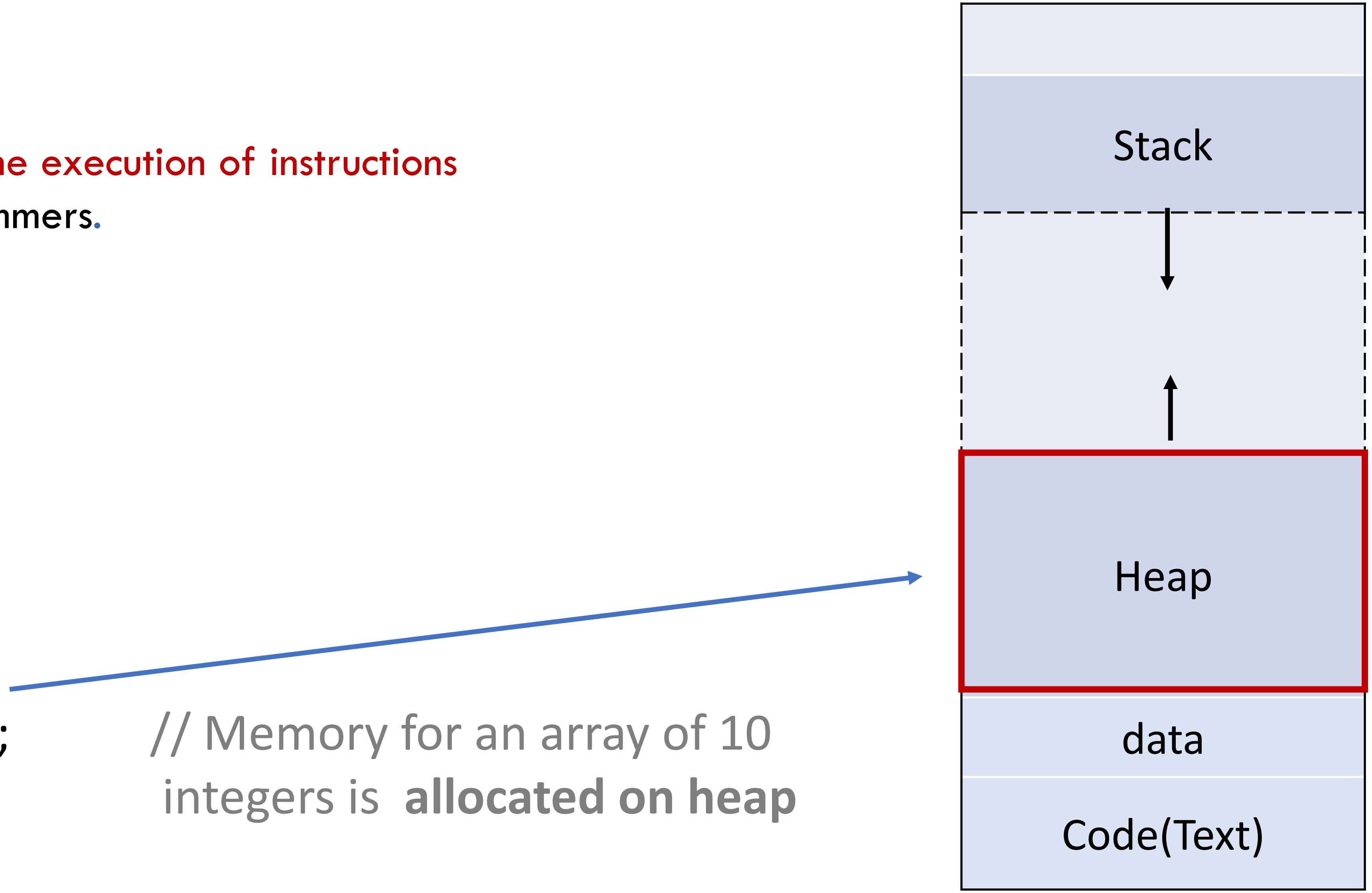


Heap Memory

- Heap Allocation
  - Allocated during the execution of instructions written by programmers.

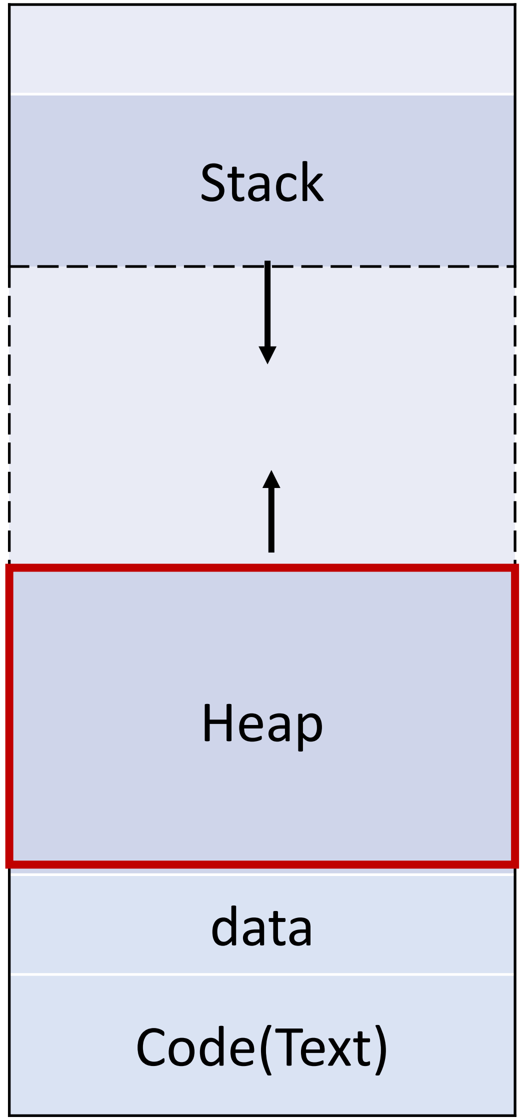
```
int *ptr  = new int[10];
```

// Memory for an array of 10 integers is **allocated on heap**



Heap Memory

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

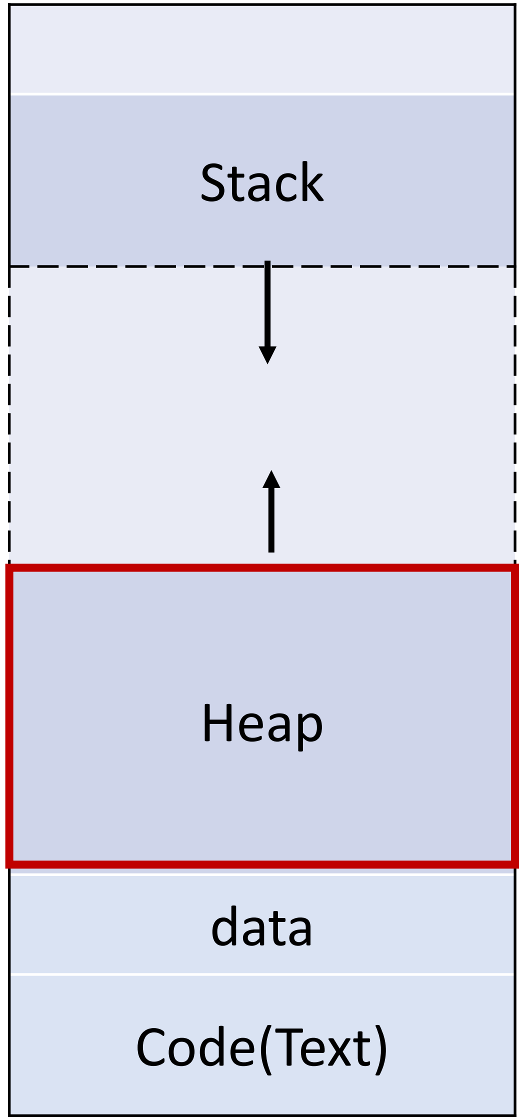


```
int *ptr  = new int[10];
Delete[] ptr;                          //  release the memory
```

Heap Memory

- Heap Allocation
  - Allocated during the execution of instructions written by programmers.
  - No automatic de-allocation feature is provided. Need to use a manually to free the memory allocated to 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
```



C++ Pointers and memory

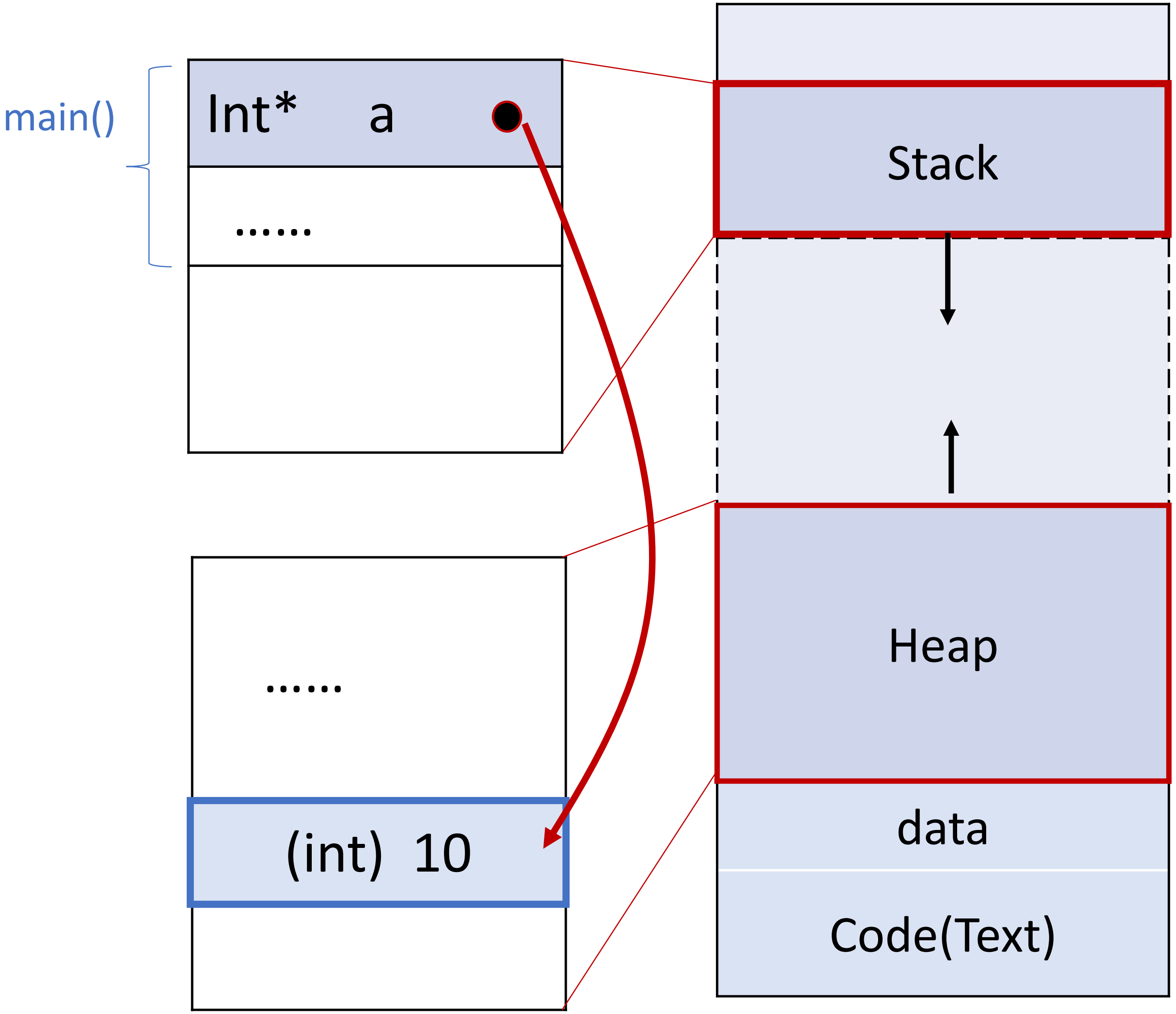
- What are C++ Pointer and Reference? Why do we have them?
- How to use C++ pointers and allocate memory for my program?

Types of Pointers

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

C++ raw pointer with heap-based memory allocation

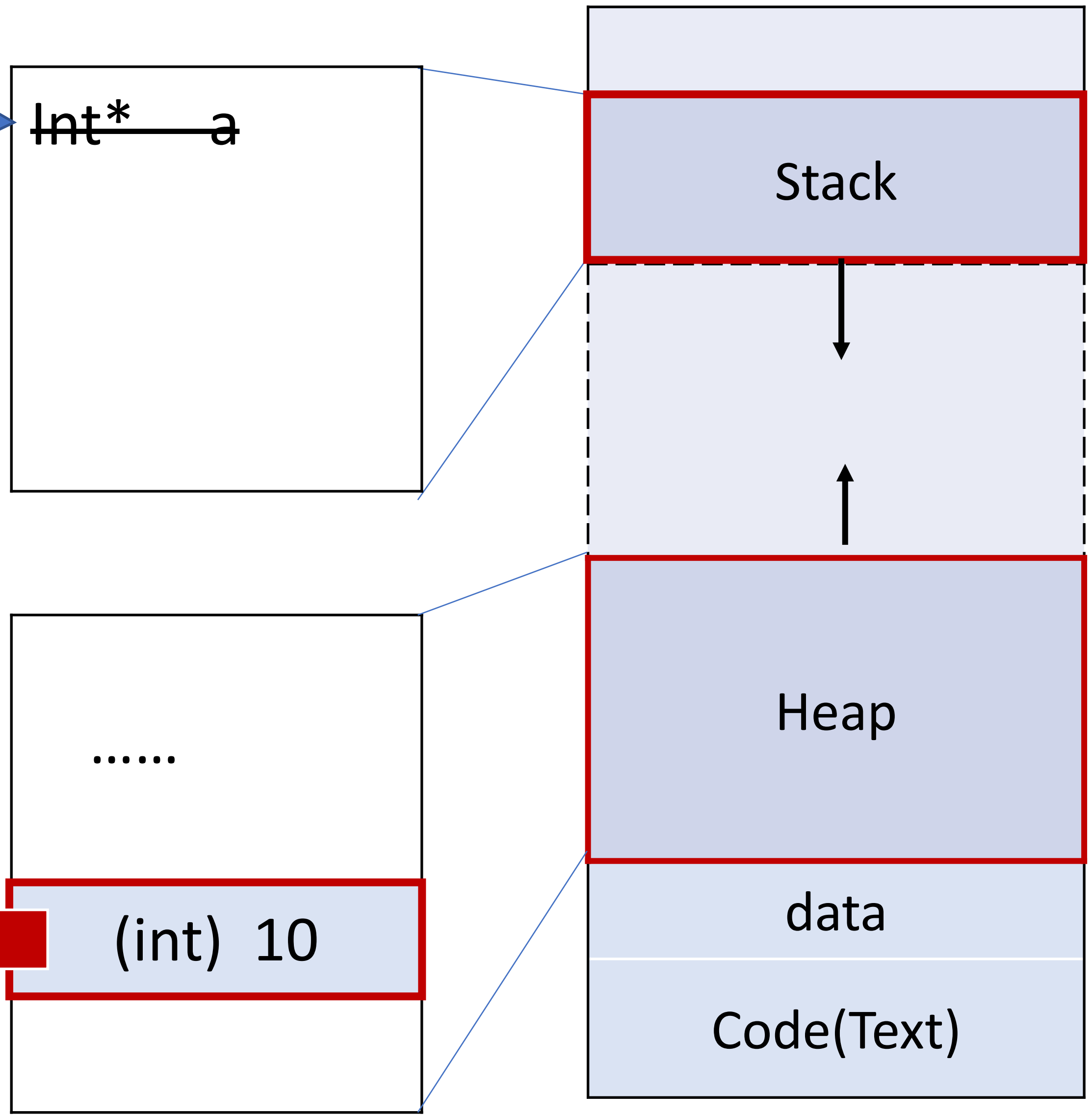
```
#include <iostream>


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



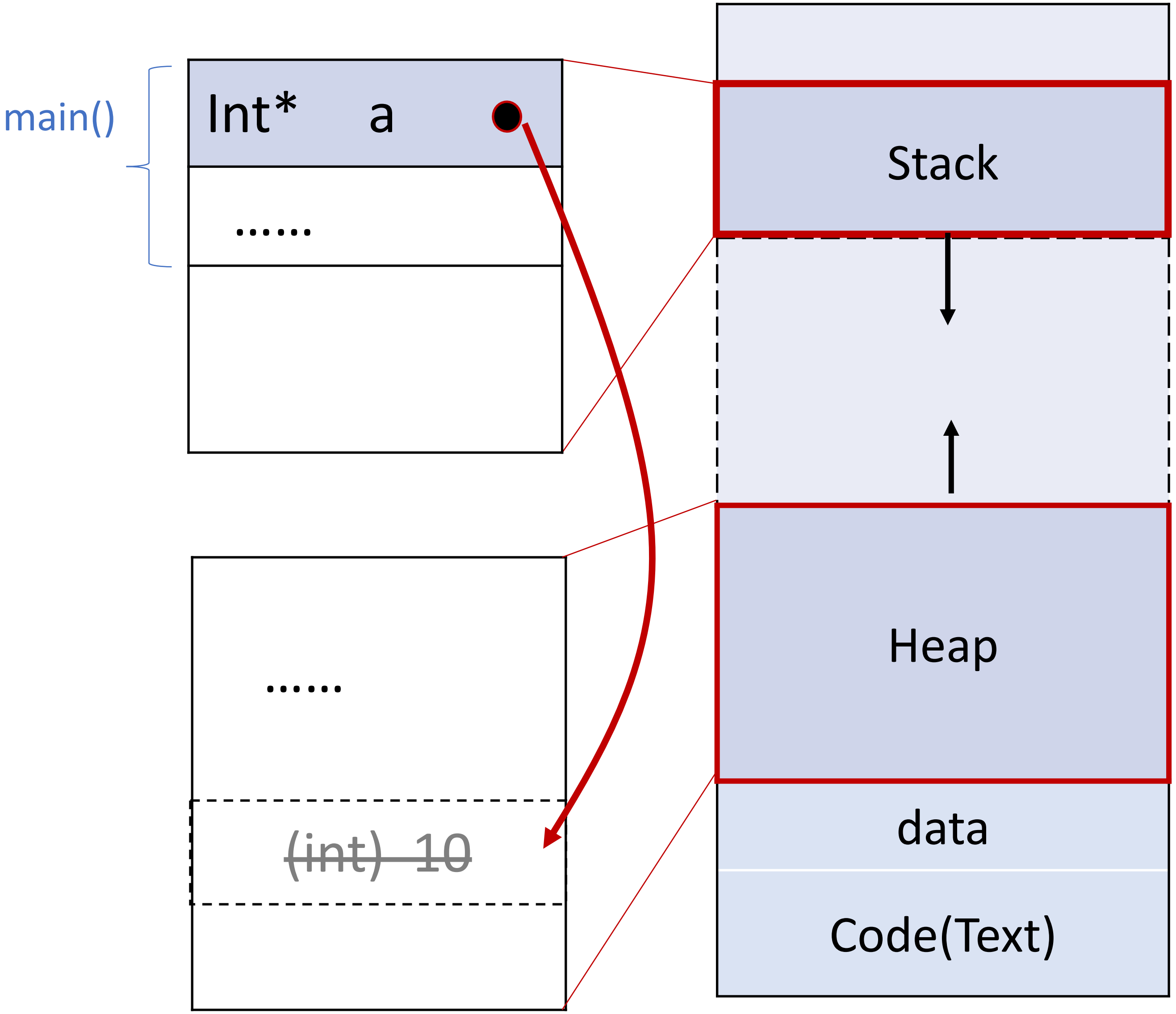
Leaked memory

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;
}
```

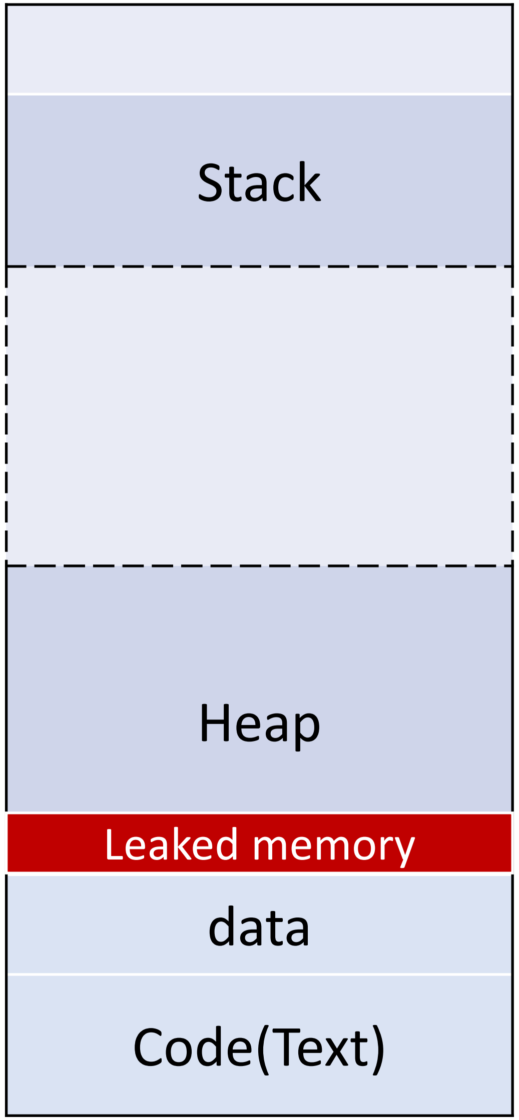


Memory Leak

- 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

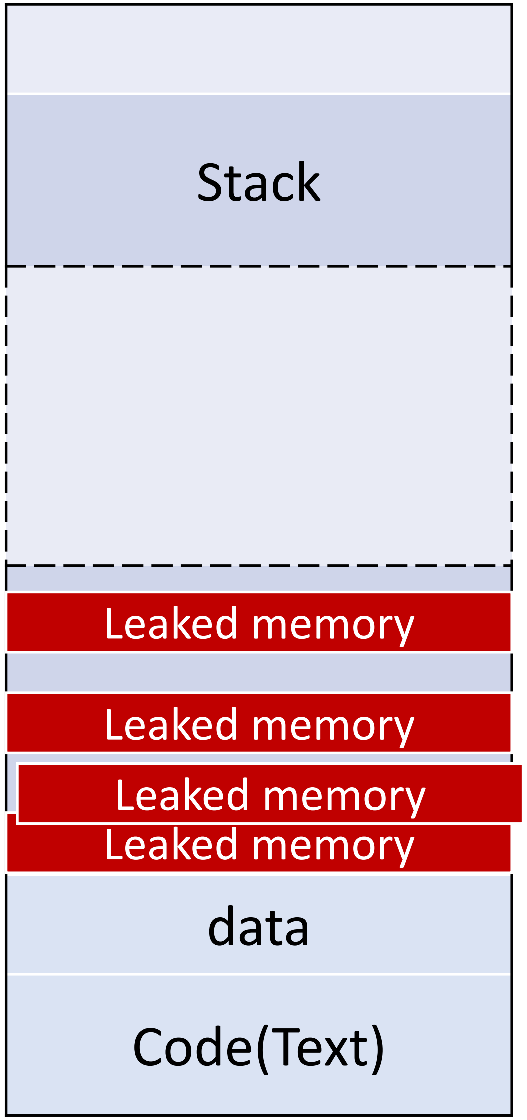
Memory Leak

- 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

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

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

demo

Exercise: std::vector of pointers

```
class Gene{

    int32_t id;

    std::string code;

public:

    Gene(): id(-1), code(""){}

    Gene(uint32_t _id, std::string _code):
id(_id), code(_code){}

};
```

```
int main(){


    std::vector<Gene*> gene_vec;

     Gene* gene1 = new Gene(0, "ITIYAY");

     gene_vec.emplace_back(gene1);


}
```

Is this code correct?

demo

Exercise: std::vector of pointers

```
int main(){


    std::vector<Gene*> gene_vec;
     Gene* gene1 = new Gene(0, "ITIYAY");
     gene_vec.emplace_back(gene1);


}
```

Memory Leak

Is this code correct? No!

demo

How to fix this?

- **Delete the allocated heap memory**
- Use stack allocation
- Use smart pointers

```
int main(){
    std::vector<Gene*> gene_vec;
    Gene* gene1 = new Gene(0,
"ITIYAY");
    gene_vec.emplace_back(gene1);
    …
    delete gene1;
// or iterate over gene_vec and call delete on
each gene_ptr
    return 0;
}
```

demo

How to fix this?

- Delete the allocated heap memory
- **Use stack allocation**
- Use smart pointers

```
int main(){
    std::vector<Gene> gene_vec;
     Gene gene1(0, "ITIYAY");
     gene_vec.emplace_back(gene1);
}
```

Types of Pointers

- C-style raw pointers
- Smart pointers: A stack-allocated object, wrapper of a raw pointer, such that when the object is destroyed, it frees the memory as well.
    - unique_ptr
    - shared_ptr

Types of Pointers

--- smart pointer: unique_ptr

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

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.
- To use smart pointers:

```
#include <memory>
```

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

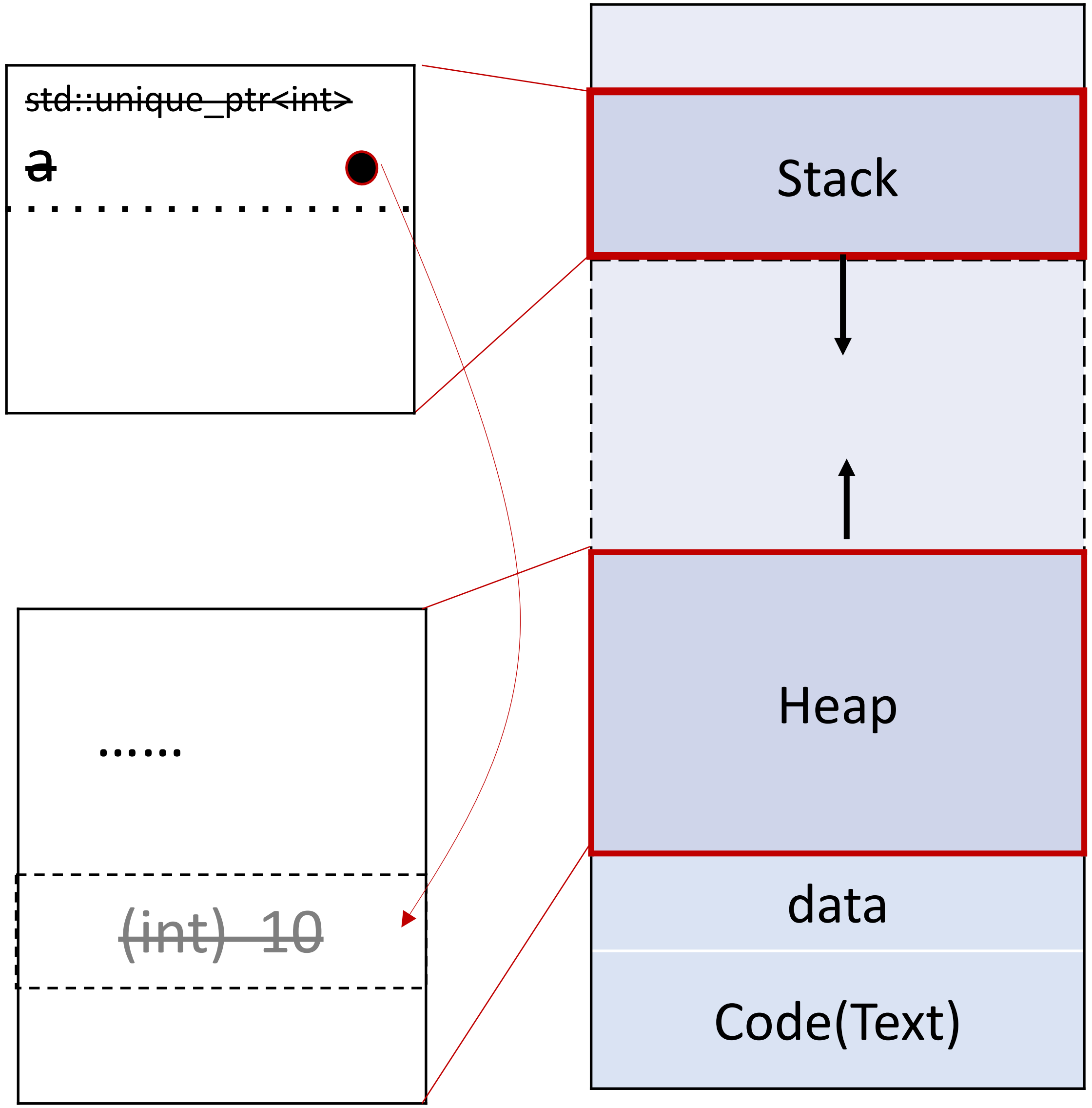
```
std::unique_ptr<Example> example2 = std::move(example);
```

✓

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;
```

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.

How to fix this?

demo

- Delete the allocated heap memory
- Use stack allocation
- **Use smart pointers**

```
int main(){
         std::vector<std::unique_ptr<Gene>> gene_vec;
        gene_vec.emplace_back(std::make_unique<Gene>(0, "ITIYAY"));
        …
}
```

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.

demo

Function Parameter

--- Passing vector

- Pass by reference

```
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

vect.size() = 2

Function Parameter

--- Passing vector

demo

- Pass by reference

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

vect.size() = 3

```
}

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

vect.size() = 3

```
}
```

Function Parameter

--- const

- 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

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){}

};
```

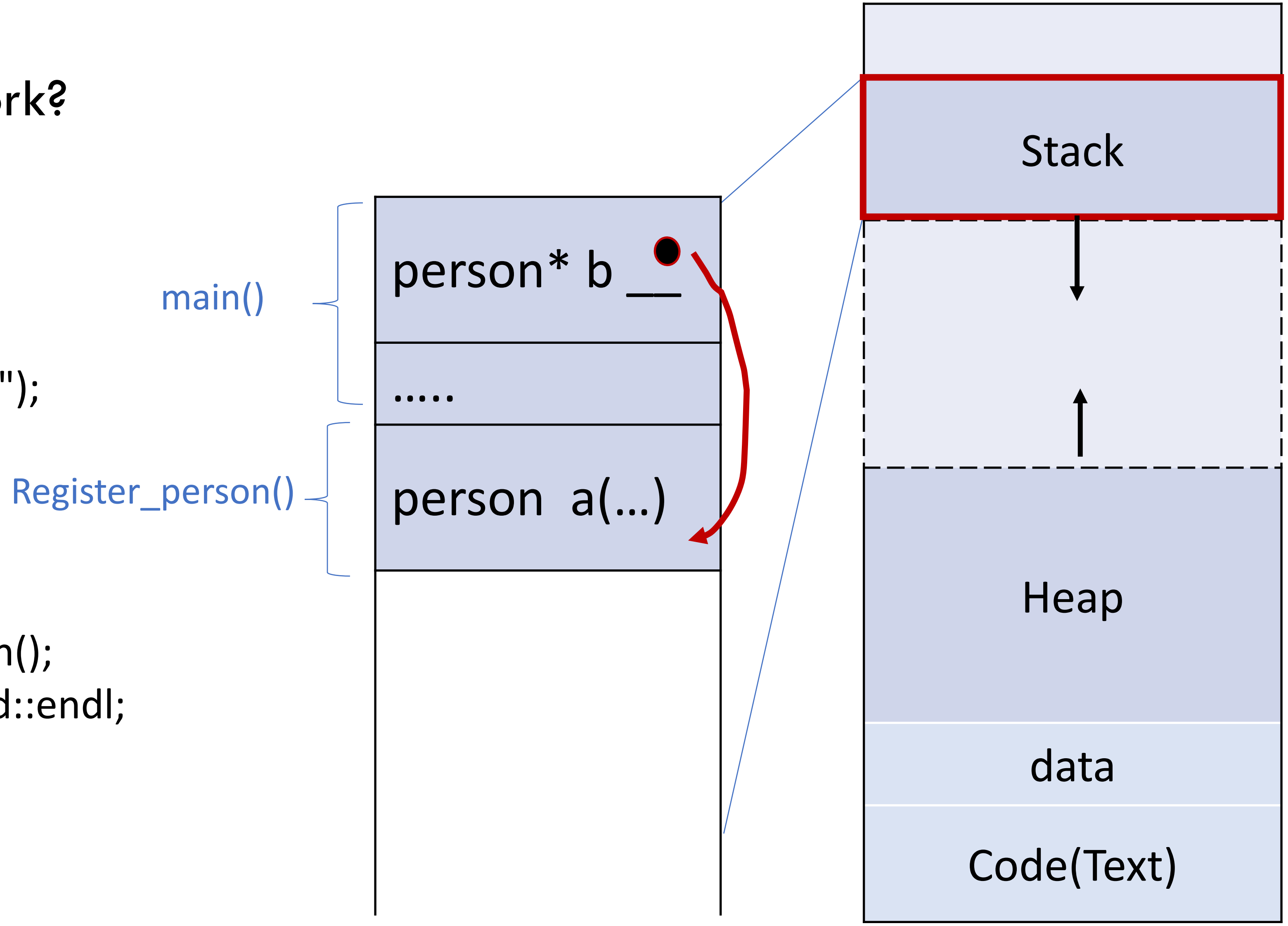
```
person* register_person(){
     person a("alicia", 1, "chess");
     return &a;
}

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

✗

Memory

- Why this code doesn’t work?

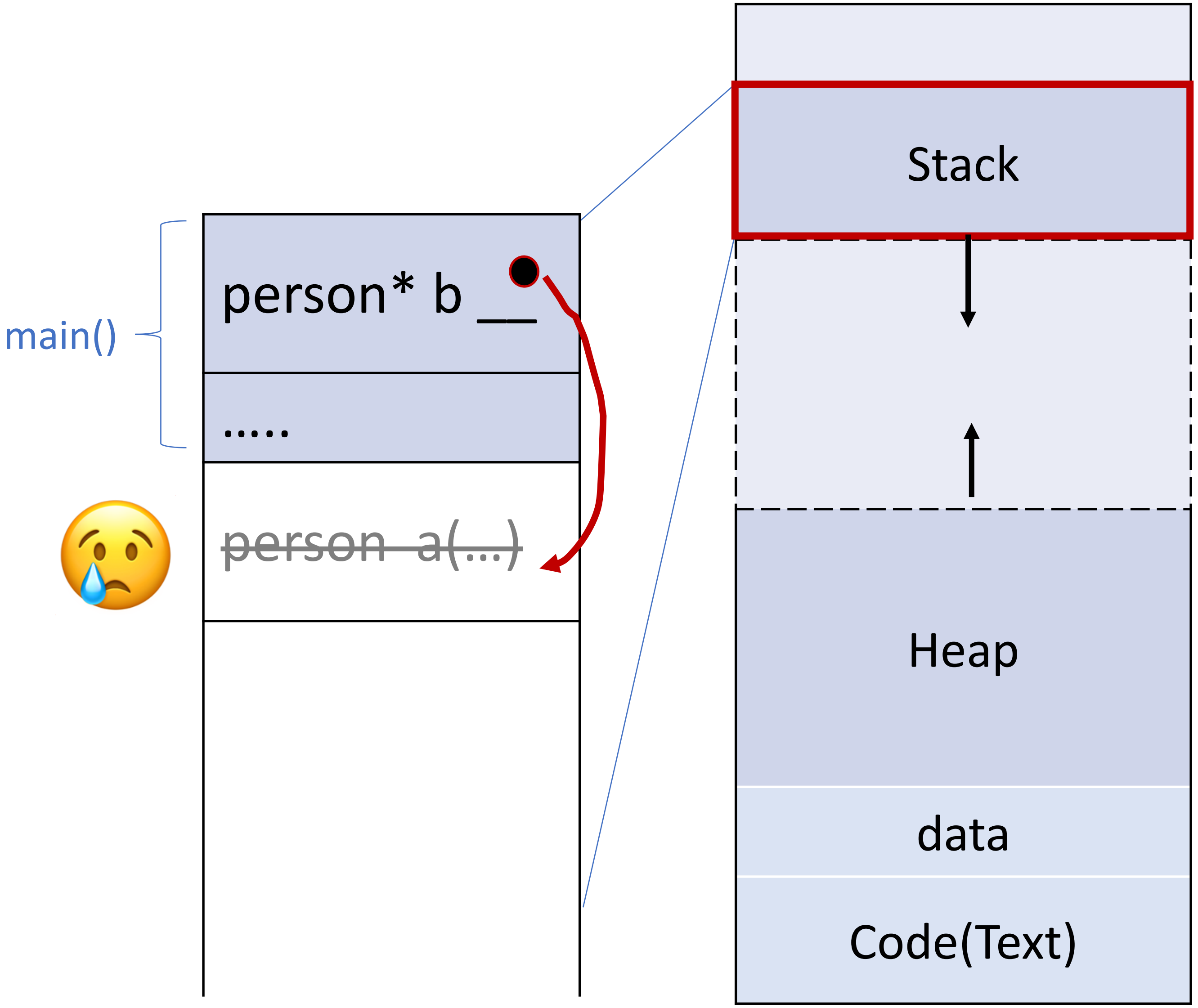
```
person* register_person(){
      person a("alicia", 1, "chess");
      return &a;
}

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

demo

- Return by value
- Pass 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)
```

Function Returns

--- array

demo

- Return by value
- Return by reference
- Return a pointer
- Return by a smart pointer

Fix3. return by smart pointer

```
std::unique_ptr<person> register_person(){
    std::unique_ptr<person> a = std::make_unique<person>("alicia", 1, "chess");
     return std::move(a);
}
```

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