CS4414 Recitation 5
Pointers and Functions C++

09/24/2021
Alicia Yang, Sagar Jha
Pointers

- Memory, Pointers and References
- Types of pointers
- ownership
Memory

• Memory for C/C++/Java program: stack and heap

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

- Memory for C/C++/Java program: stack and heap

- 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)
  - Faster to allocate memory on stack (1 CPU operation) than heap
• Stack Allocation (Temporary memory allocation):

```c
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);
    return toal;
}
```
• **Stack Allocation (Temporary memory allocation):**

```c
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);
    return total;
}
```
Memory

• Stack Allocation (Temporary memory allocation):

  Stack free memory via stack pointer

```c
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);
    return total;
}
```

--- stack and heap allocation

Application’s memory

Stack

Heap

Main()  a, b

Stack

Static/global

Code(Text)
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
- Larger memory size compared to stack memory

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

• A pointer is a variable that stores the memory address of an object.

• Why use pointers?
  • to allocate new objects on the heap
  • to pass functions to other functions
  • to iterate over elements in arrays or other data structures
Pointers

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

• A pointer is a variable that stores the memory address of an object.

• Example:

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

Hey, what IS your memory address?
Hey, what IS stored IN your memory address?
• Reference, is an alias, is another name for an already existing variable. It only exist in source code

```cpp
int num = 10;
int* bar = &num;
int& ref = num;
ref = 2;
```
Types of Pointers

• C-style raw pointers

• Smart pointers
  • unique_ptr: prefer, low overhead
  • shared_ptr

• Iterators
**Types of Pointers**

--- raw pointers

Example* example = new Example(); // Use the * operator to declare a pointer type
// Use new to allocate and initialize memory

Example example2 = *example; // Copy the pointed-to object, by dereferencing the pointer
access the contents of the memory location.

Example* ecopy = &example2; // Declare a pointer that points to example using the
address of operator

ecopy->print(); // Accessing filed/function of an object’s pointer using ->

delete example; // release memory back to OS, delete ecopy is dangerous
// anything allocate with new, should delete the memory to prevent memory leak
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`

• 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 another object through a pointer and disposes of that object when the unique_ptr goes out of scope.

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

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(example1); ✓
```

Demo: https://github.com/aliciayuting/CS4414Demo.git
• Allow several shared_ptr objects 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)

```cpp
std::shared_ptr<Example> example = std::make_shared<Example>();
std::shared_ptr<Example> example = new Example();
std::shared_ptr<Example> example2 = example;
```

Less efficient, two allocations:
construct example, then construct control blo
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

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

```cpp
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**: a direct pointer to the memory array by the vector to store its elements.

- **Buggy code example**:

  ```cpp
  std::vector<int> intVector;
  intVector.push_back(1);
  int* pointerToInt = &intVector[0];  // We get the pointer to the first element from our vector.
  ```
• **Vector pointer**: a direct pointer to the memory array by the vector to store its elements.

• Buggy code example:

```cpp
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
intVector.push_back(3); // resize the internal array is deleted causing our pointer

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

• **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<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++

```cpp
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;
```
Functions
### Function Parameter

- **Pass by value**: passing the copy of the value
  ```cpp
  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
  ```cpp
  void fun(X *x);  // & means get the address_of
  X x;
  fun(&x);
  ```

- **Pass by reference**: passing a reference
  ```cpp
  void fun(X &x);  // & means the parameter type is reference
  X x;
  fun(x);
  ```
Function Parameter

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

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

int main()
{
    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

- Pass by reference
  (preferred to pass by reference than pass by pointer: References cannot be null.)

```cpp
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 keyword in parameter of `reference`: a promise that the variable being referenced to be changed through the reference.

```cpp
void foo(const std::string& x) // x is a const reference
{
    x = "hello"; // compile error: a const reference cannot have its value changed!
}
```
• Const keyword in parameter of **pointer**: 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.

```c
const type * identifier;
void fcn(const int* p){
    *p = expression;
}
```

// compiler complain: here it is illegal to have a const pointer’s content change
Function Returns

• Return by value: returning a copy of the value

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

• Return by reference

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

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

```cpp
int* test () {
    int c[5];
    for (int i = 0; i < 5; i++)
        c[i] = i;
    return c;
}

int main(){
    int * result = test();
    std::cout << "First Value is " << result[0] << std::endl;
    ...
}
```
• Why this code doesn’t work?

```cpp
int* test () {
    int c[5];
    for (int i = 0; i < 5; i++)
        c[i] = i;
    return c;
}

int main(){
    int * result = test();
    std::cout << "First Value is " << result[0] << std::endl;;
    ...
}
```
• Why this code doesn’t work?

```c
int* test () {
    int c[5];
    for (int i = 0; i < 5; i++)
        c[i] = i;
    return c;
}

int main(){
    int * result = test();
    std::cout << "First Value is " << result[0] << std::endl;
    ...
}
```
Function Returns

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

Fix1. std::array (better)
```cpp
std::array<int,5> test () {
    std::array<int,5> c;
    for (int i = 0; i < 5; i++)
        c[i] = i;
    return c;
}
```

Fix2. use heap
```cpp
int* test (void) {
    int* out = new int[5];
    return out;
}
```
(need to release the memory of the returned pointer)

Demo: [https://github.com/aliciayuting/CS4414Demo.git](https://github.com/aliciayuting/CS4414Demo.git)
Function Returns

• Example:

Does this program work as intended?

class Student{
private:
    std::string name;
public:
    Student(const std::string& name) :
        name(name){}
    std::string get_name() {
        return name;
    }
};

class CS4414{
private:
    std::vector<Student> students;
public:
    std::vector<Student> get_students(){
        return students;
    }
};

Demo: https://github.com/aliciayuting/CS4414Demo.git
Where to find the resources?

- Memory Heap and Stack: https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/
- 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/