ECE 449/590 – OOP and Machine Learning
Lecture 07 Memory Management,
The C++ List

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

September 14, 2022

Outline

Reading Assignment

- This lecture: Accelerated C++ 10, 5
- Next lecture: Accelerated C++ 6 – 8

Outline

References

```
int x = 5;
int &r = x;
r = 6; // what is x now?
```

- A reference associates a name (alias) to an object.
- The type of the reference to an object of type `T` is `T &`.
  - `T` should not be a reference type.
  - `&` here has nothing to do with taking address or bit-wise and.
- References must be initialized when defined.
  - Then it can be used in a way similar to a variable.
- C++ references work very differently than "references" in languages like Java or Python.
  - Their "references" behave almost the same as C/C++ pointers and not realizing such will lead to the "aliasing" problem.

Pointers

```
int x = 5;
int &r = x;
int *p = &x;
int *q = &r;
```

- A pointer is an object whose value is the memory address of the object it points to.
- The type of the pointer pointing to an object of type `T` is `T *`.
  - `T` should not be a reference type.
  - `*` here has nothing to do with multiplication or dereference.
- Use the address operator `&` to take the address of an object
  - Apply to both variables and references
  - `&` here has nothing to do with reference type or bit-wise and.

Dereference

```
int x = 5;
int &r = x;
int *p = &x;
int *q = &r;
*p = 6; std::cout << "x = " << x << std::endl; // x = 6
*q = 7; std::cout << "x = " << x << std::endl; // x = 7
```

- The object pointed by a pointer can be accessed using the dereference operator `*`.
- The members can be accessed using the arrow operator `->`.
- Pointers are primitive (built-in) types.

NULL Pointers

```
int *p = nullptr;

if (p == nullptr) {
    std::cout << "p is a NULL pointer" << std::endl;
}
```

- A `nullptr` is a pointer pointing to no object.
  - You cannot dereference `nullptr`.
  - The mistake of dereferencing `nullptr` pointer is so often that most modern operating systems configure their memory systems in a way such that it would lead to an exception.
- Invariant for pointers
  - It points to an object or is nullptr.

Call by Value

```
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp; // now the value of a and b are exchanged
}
int main() {
    int x = 1, y = 2;
    swap(x, y);
    std::cout << x << " " << y << std::endl;
    return 0;
}
```

- The answer is “1 2”.
  - Why not “2 1”?
  - `a` is a different variable than `x`. It just take the same value as `x` when the function is called.
  - Similarly, `b` and `y` are different, though with the same value.
  - So you swapped `a` and `b` but not `x` and `y`.
- There is no side effect for the arguments `x` and `y` from the caller `main`.
  - Help to prevent mistakes resulting from undesired side effects.

Call by Value (Cont.)

```
void swap(int *pa, int *pb) {
    int temp = *pa;
    *pa = *pb;
    *pb = temp; // now the value of *pa and *pb are exchanged
}
int main() {
    int x = 1, y = 2;
    swap(&x, &y);
    std::cout << x << " " << y << std::endl;
    return 0;
}
```

- The answer is "2 1" now.
- While `pa` and `pb` are still values, we may use them to change `x` and `y` indirectly.
  - There is no side effect for the arguments `&x` and `&y`.
  - However, we can use `&x` and `&y` to change `x` and `y`.

Call by Reference

```
void swap(int &a, int &b) {
    int temp = a; a = b; b = temp;
}
int main() {
    int x = 1, y = 2;
    swap(x, y);
    std::cout << x << " " << y << std::endl;
    return 0;
}
```

- The answer is also “2 1”.
- References provide alias (another name) to objects.
  - When `swap` is called, no object is constructed for `a` or `b`.
  - `a` is another name for `x`. We can simply say `a` binds to `x` and also `b` binds to `y`.
  - Swapping `a` and `b` will swap `x` and `y`.
- References allows to have side effects on arguments.

Scopes of Function Parameters

- All the parameters have the scope of function.
  - Scope: where you can refer to an object or variable.
- Call-by-value parameters are local variables.
  - Constructed from the arguments in the caller when the function is called.
  - Destroyed when the function returns.
- Call-by-reference parameters are alias.
  - Bind to the arguments in the caller when the function is called.
  - Arguments aren't destroyed when the function returns.
- C++ supports these two types of parameters.
  - C supports call-by-value only.
  - Java/Python supports a restricted version of call-by-value only – can you implement `swap` in Java or Python?

Outline

Operating System and C/C++

- Operating system (OS): a software system running on a hardware platform
  - Provide common services to application software
  - Enable multiple applications to share hardware resources
- OS' achieve these two goals by providing abstract models for processors, memory, and I/O devices.
  - The models are specified as functions and rules to call the functions, usually as part of the application programming interface (API).
- The C language can use these models efficiently as most of them are implemented in C or just part of C.
  - So does C++, which inherits most features of C and the C standard library.
- Let's focus on memory.

Memory and Virtual Memory

- Memory: an array of bytes
  - Store program and data (variables and objects)
- Memory address: index into the memory array
  - Pointer types hold memory address as values.
- Memory corruption: the contents of a memory location are unintentionally modified due to programming errors
  - Almost all hard-to-diagnosis undefined behaviors are caused by or lead to memory corruption.
- Virtual memory: an ideal memory model used by most modern OS'
  - Protection: each application runs in its own memory so errors in other applications won't cause memory corruption
  - Program model: a single memory array with a fixed size, no matter how physical memory is installed and organized
- Memory allocation: to use a piece of memory, it must be acquired first
  - If every piece of code only modifies the memory it is allowed to modify, then there will be no memory corruption.

Memory Management

- The memory is typically divided into the follow 4 areas to facilitate memory management.
  - The code area stores the binary code of the program.
  - The global area stores global variables and other variables that persist throughout program execution.
  - The stack stores function parameters and local variables.
  - The heap stores objects that are generated at runtime.
- Usually, these areas are managed as follows.
  - The OS will take care of the code area.
  - Variables in the global area are statically allocated by the compiler at compile-time.
  - The compiler will generate code to manage the stack automatically at runtime.
  - The programmer is responsible to manage objects in the heap through dynamic memory allocation.
- Let's focus on the stack and the heap.

The Stack

- The elements in the stack are usually called stack frames.
  - In the debugger, each frame is summarized as a function call on the call stack.
- A stack frame is generated and placed to the top of the stack when a function is called. It contains
  - The parameters of the function.
  - The return address.
  - The local variables in the function.
  - Information on where to store the result.
  - Information for the debugger to determine the call stack.
- The stack frame is destroyed when the function call returns.
  - Recall that non-reference parameters and local variables are destroyed when the function returns.
  - In other word, the lifetime of an object on the stack is within the function itself.

An Example of Lifetime

```
std::string &get_name() {
    std::string name = "jia";
    return name;
}
```

- The lifetime of `name` is within the function `get_name`.
- So it no longer exists when the function returns.
- The returned result, as a reference to an object that no longer exists, will lead to undefined behavior if being accessed.

Another Example of Lifetime

```
std::string get_name() {
    std::string name = "jia";
    return name;
}
```

- You have copied the value of the local variable `name` to the result of the function before `name` is destroyed.
- So the code is correct.

Pointers and Lifetime

```
std::string *get_name_ptr() {
    std::string name = "jia";
    return &name;
}
```

- The lifetime of `name` is within the function `get_name_ptr`.
- So it no longer exists when the function returns.
- The returned result, as a pointer to an object that no longer exists, breaks invariant for pointers and will lead to undefined behavior if dereferenced.

The Heap

- The area that a programmer can request a piece of memory to construct a new object at runtime.
  - We can say the object is <u>on the heap</u>.
- An object on the heap will remain there until being destroyed.
  - Usually the piece of memory will be returned to the heap for future use at the same time.
- The heap contains only a limited amount of memory.
  - A program may deplete the heap by not destroying objects on the heap that are no longer in use.
  - In C/C++, it is the responsibility of the programmer to destroy the object when it is no longer in use.
- Objects on the heap usually have no names at compile-time.
  - Need to use references or pointers.
  - C++ follows C to use pointers for <u>dynamic memory allocation</u>.

Dynamic Memory Allocation

```
// You are NOT supposed to write code like below!
int *p = new int(5);
std::cout << *p << std::endl; // 5
```

- The expression `new T(args)` will create an object with the arguments `args` on the heap, and return a pointer to the object.
- It will do two things:
  - Request a piece of memory from the heap that can hold an object of type `T`.
  - Construct the object from `args` on that piece of memory.
- If there is no argument, you shouldn't provide `args` and `()`.

Memory Deallocation

```cpp
// You are NOT supposed to write code like below!
int *p = new int(5);
std::cout << *p << std::endl; // 5
delete p;
```

- The objects on the heap can be used until they are `delete`d.
- The expression `delete p` will do two things:
  - Destroy the object pointed by `p`.
  - Return the piece of memory that was occupied by the object pointed by `p`. to the heap

Issues with Pointers and Dynamic Memory Allocation

- For dynamic memory allocation, programmers are expected to track the objects pointed by pointers.
  - They should avoid dereferencing a pointer when the object it points to has been destroyed.
- In reality, that's almost impossible for complex software systems.
  - In a program, usually there are many pointers pointing to the same object.
  - If one decide to delete an object on the heap through one pointer, all the other pointers pointing to the same object should no longer be dereferenced.
  - However, at that time, it is too late to tell which pointers actually point to the object.
- Helps from the compiler and the language implementations are needed to address these issues.

Solutions

- Solution 1: programmers should not delete objects
  - Almost all modern languages choose to do so by default.
  - Garbage collection (GC): the heap implementation is responsible to find objects that are no longer in use and to delete them automatically.
- Solution 2: programmers utilize OOP to let the compiler generate code that can handle pointers and dynamic memory allocation correctly
  - That's the solution of C++.
  - Though it takes more effort, it usually results in more predictable performance in comparison to GC.
  - It is also possible to implement GC based on Solution 2.
- You should not use `new[]` and `delete[]` in modern C++ programs. So we won't cover them in this course.

Outline

References and Pointers

Memory Management

The C++ List

The C++ List

```
#include <list>
#include <iostream>
int main() {
    std::list<int> integers;
    for (size_t i = 0; i < 10; ++i) {
        integers.push_back(i);
    }
    // How can we display the elements if [] is not supported?
    return 0;
}
```

- `std::list` is another kind of containers.
  - Defined in the standard header `list`
  - A template class like `vector`
- The elements are organized into a doubly-linked list.
  - Inserting/erasing an element anywhere within the container are fast.
  - However, random accesses (`[]`) are not supported.
- Let's focus on the interface of `std::list`.
  - Implementing a doubly-linked list correctly requires you to know many subtle features of the language.

Access Elements in Vector

```
std::vector<int> integers;
... // populate the vector
for (size_t i = 0; i < integers.size(); ++i) {
    std::cout << integers[i] << std::endl;
}
```

- Though we access elements using `[]`, the elements are accessed sequentially.
  - The only operations on `i` are to initialize it to `0`, increment it by `1`, and to compare it with the size.
  - We do not access the elements randomly as allowed by `[]`.
- However, the library has no way to know it.
  - A sequence is expressed as a range $[\text{begin}, \text{end})$.
  - If we make that knowledge available to the library, then it is possible to reuse the pattern of asymmetric ranges and loops to visit elements in other containers.

Iterators

- A concept to allow traversing all the elements in a container.
  - Each kind of containers will define C++ types for its OWN iterators.
  - Iterators are generalization of C/C++ pointers.
- An iterator is an object that
  - Identify a container and a place in the container.
  - Allow to access the element at that place if the element is valid.
  - Provide operations for moving between elements in the container.
  - Restrict the available operations in ways that correspond to what the container can handle efficiently.

List Iterators

```
std::list<int> integers;
... // populate the list
for (std::list<int>::iterator iter = integers.begin();
    iter != integers.end(); ++iter) {
    ...
}
```

- The type of iterators for `std::list<T>` is `std::list<T>::iterator`.
    - We usually use `T` to refer to a value type
        - A value type is a type that is not a reference.
    - The iterator type is within the scope of `std::list<T>`.
- `begin()` and `end()`, as suggested by their names, return the either ends of the asymmetric range.
- Operators `==`, `!=`, `++`, `--` are overloaded on iterator types.
    - Comparisons like `<` and `<=` are not always supported.
- So the for loop pattern for the asymmetric range still works.

    `for (index = begin; index != end; ++index)`

Access Elements using Iterators

```
for (std::list<int>::iterator iter = integers.begin();
    iter != integers.end(); ++iter) {
    std::cout << *iter << std::endl;
}
```

- For a container with n elements, an iterator should represent one of the $n+1$ places with the range $[\text{begin}, \text{end}]$.
- If it is within the asymmetric range $[\text{begin}, \text{end})$, there is a element at the corresponding place.
- The element can be accessed by the dereference operator `*`.
  - Unfortunately `*` is abused (it also stands for multiplication). Anyway, its meaning should be clear from the context.
  - The iterator must be within $[\text{begin}, \text{end})$ (cannot be end for this operation).

Vector Iterators

```
std::vector<int> integers;
... // populate the vector
for (std::vector<int>::iterator iter = integers.begin();
    iter != integers.end(); ++iter) {
    std::cout << *iter << std::endl;
}
```

- Why use iterators when it seems more easy to use indices?
  - (Compile-time) Polymorphism: using iterators allows to process elements in containers in a way independent of container types.
- Why not use iterators for vectors?
  - An previously stored iterator CANNOT be used if any element is inserted/erased from the container.

`auto` Type Deduction

```
for (auto iter = integers.begin(); iter != integers.end(); ++iter) {
    std::cout << *iter << std::endl;
}
```

- It is possible to ask the compiler to deduce the type of a variable for you when it is defined.
  - Use the `auto` keyword.
  - In the above case, the type of `iter` is deduced to be the same as the return type of `integer.begin()`.
- Quite convenient, though you still need to understand the C++ type system to reason with any compiling error.

Range-Based `for` Loops

```
for (int i: integers) {
    std::cout << i << std::endl;
}
```

- It is so common to iterate through a container using the range $[\text{begin}, \text{end})$ that C++ now allows range-based `for` loops.
- The `int i` says to make a copy of the elements in `integers`.
  - Similar to that in a function parameter list.
  - Use `int &i` if we need to modify the elements.
  - Use `const int &i` if we don't need to modify the elements but want to avoid the copy.

`auto` and Range-Based `for` Loops

```
for (auto i: integers) {
    std::cout << i << std::endl;
}
```

- `auto` type deduction works with range-based `for` loops.
- The `auto i` says to make a copy of the elements in `integers`.
  - `auto &i` or `const auto &i` are also valid here for their respective purposes.

Summary and Advice

- Memory is divided into 4 areas: code, global, stack, heap
  - Objects on the stack have a lifetime of the function and are managed by the compiler automatically.
  - Objects can be create on the heap in order to have a lifetime controlled by the programmers.
- A sequential container stores a linear sequence of elements.
  - `std::vector` is a kind of sequential containers that is optimized for fast random access.
  - `std::list` is a kind of sequential containers that is optimized for fast insertion and deletion anywhere.
  - Use `iterator`s to access elements in containers sequentially.