Introductory Robot Programming

ENPM809Y

Lecture 05 – Pointers and References

Zeid Kootbally

zeidk@umd.edu

Fall 2020

Overview I

Overview II

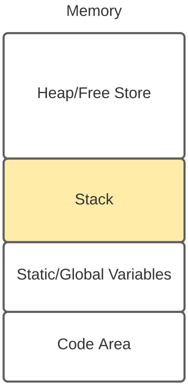
Overview III

Memory Allocation

- C++ supports three basic types of memory allocation.
  - Static memory allocation happens for static and global variables. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
  - Automatic memory allocation happens for function parameters and local variables. Memory for these types of variables is allocated on the stack when the relevant block is entered and freed when the block is exited.
  - Dynamic memory allocation is a way for running programs to request memory from the operating system when needed. This memory does not come from the program's limited stack memory. Instead, it is allocated from a much larger pool of memory managed by the operating system called the heap. On modern machines, the heap can be gigabytes in size.

Stack Memory

- The stack and the heap are both stored in the computer's RAM (Random Access Memory).
- An object created (without the `new` operator) in a function is stored on the stack. This object is destroyed (memory is freed) when it goes out of scope.
- The stack is set to a fixed size, and can not grow past this size. So, if there is not enough room on the stack to handle the memory being assigned to it, a stack overflow occurs. This often happens when a lot of nested functions are being called, or if there is an infinite recursive call.
- Memory allocation on the stack is much faster because of the mechanism used (moving the stack pointer).



Stack Memory

- When we run the program below, 8 bytes (2×4 bytes) of memory are allocated on the stack.
- Memory will be deallocated for `a` and `b` when they go out of scope.
  - First, memory will be deallocated for `b` at line 5.
  - Then memory will be deallocated for `a` when the function terminates at line 6.

```
1 int main(){
2     int a{};
3     {
4         int b{};
5     }
6 }
```

The Address-of Operator

- The address-of operator (&) allows us to see what memory address is assigned to a variable.

```cpp
int main(){
    int a{1};
    //--what is the value of a?
    std::cout << a << std::endl;//--1
    //--&a = what is the address of a?
    std::cout << &a << std::endl;//--0x7ffcb7128474
}
```

The Dereference Operator

- The dereference operator (*) allows us to access a value at a particular address.

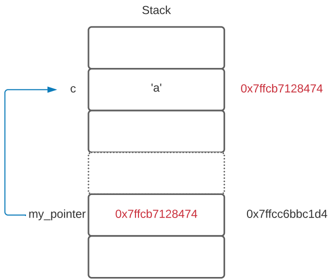
```
int main(){
    int a{1};
    //--what is the value of a?
    std::cout << a << std::endl;//--1
    //--what is the address of a?
    std::cout << &a << std::endl;//--0x7ffcc6bbc1d4
    //--what is the value at the address of a?
    std::cout << *(&a) << std::endl;//--1
}
```

Pointers

- With the address-of and the dereference operators added to our toolkit we can now talk about pointers.
- Pointer: A variable that holds a memory address as its value.
- Consider the following:

```
int main(){
    char c{'a'};
    char *my_pointer{&c};
}
```

- `my_pointer` is a variable whose value is the address of `c`.



Pointers | Use

- Why use pointers?
  - Pointers can be used inside a function to access data that are defined outside the function.
  - Pointers can be used to operate on C-style arrays efficiently.
  - You can use pointers to allocate memory on the heap (dynamic memory allocation).
  - Pointers are used for polymorphism in object-oriented programming.
  - Pointers can be used to access specific addresses in memory.
  - Pointers can return more than one value.
  - Pointers can improve the efficiency of your program.

Pointers | Declaring a Pointer

- Pointer variables are declared just like normal variables, only with an asterisk between the data type and the variable name.
- Warning: This asterisk is not a dereference operator. It is part of the pointer declaration syntax.
- Take a look at the style guide on pointers.

```
int *i_ptr;     //--asterisk next to the name
double* d_ptr; //--asterisk next to the type
char * c_ptr;  //--asterisk in the middle

//--two pointers to string objects (not recommended)
std::string *s1_ptr, *s2_ptr;
//--Be careful
int* my_ptr1, my_ptr2;//--one pointer to int and one plain int
```

- Note: You read pointer declarations from right to left.

Pointers | Initializing a Pointer

- If you want to declare a pointer and use it later in the program then initialize it to "point nowhere" with `nullptr` (since C++11). Your program is never allowed to store anything into memory address 0, so `nullptr` is a way of saying "point nowhere".

```
int *i_ptr{nullptr};
double* d_ptr;
d_ptr = nullptr;
```

- You can also initialize your pointer with the address of an existing variable.

```
int a{3};
int *ptr{&a};
```

- Warning: Always initialize your pointers. Uninitialized pointers are dangerous and can point anywhere.

Pointers | The Address-of Operator

- It is worth noting that the address-of operator does not return the address of its operand as a literal.
- Instead, it returns a pointer containing the address of the operand, whose type is derived from the argument (e.g., taking the address of an `int` will return the address in an `int` pointer).

```cpp
#include <iostream>
#include <typeinfo>

int main(){
    int x{1};
    int *p{&x};
    std::cout << &x << std::endl; //--0x7ffe760cef0c
    std::cout << p << std::endl; //--0x7ffe760cef0c
    std::cout << typeid(&x).name() << std::endl;//--Pi
    std::cout << typeid(p).name() << std::endl;//--Pi
}
```

Pointers | Typed Pointer

- Although pointers always hold the same type of data (addresses), we still need to provide a type to a pointer.

```
1 int main() {
2     int a{5};
3     double b{2.0};
4     int *ptr{nullptr};
5     ptr = &a;//--OK
6     ptr = &b;//--error: cannot convert double* to int*
7 }
```

- The compiler ensures that the type of the pointer and the variable address the pointer is being assigned to match.
- But this still does not answer why this is necessary. Pointers hold addresses and `int` at line 4 is not being used. The answer is in the next slide.

Pointers | Dereferencing a Pointer

- Once we have a pointer variable pointing at something, the other common thing to do with it is dereference the pointer to get the value of what it is pointing at.
- A dereferenced pointer evaluates to the contents of the address it is pointing to.

```cpp
int main() {
    int a{5};
    std::cout << &a; //--address of a
    std::cout << a; //--value of a
    int *ptr{&a}; //--points to a (holds the address of a)
    std::cout << ptr << std::endl; //--address held in ptr, which is &a
    std::cout << *ptr << std::endl; //--value that ptr is pointing to (value of a)
}
```

- Without a type, a pointer would not know how to interpret the contents it was pointing to when it was dereferenced. Without a type the compiler would misinterpret the bits as a different type.

Pointers | Dereferencing a Pointer

- One powerful feature of pointers is their ability to modify the content of variables they point to.
- Another feature is to be able to reassign a pointer to another variable.

```
int main() {
    int a{5}, b{6};
    int *ptr{&a};
    std::cout << a << std::endl; //--5
    std::cout << *ptr << std::endl; //--5 (value of a)
    *ptr = 10; //--changing the value of a through a pointer
    std::cout << a << std::endl; //--10
    std::cout << *ptr << std::endl; //--10 (value of a)
    //--reassigning a pointer
    ptr = &b;
    std::cout << *ptr << std::endl; //--6 (value of b)
}
```

Pointers | The Size of Pointers

- The size of a pointer is dependent upon the architecture the executable is compiled for.
  - A 32-bit executable uses 32-bit memory addresses, consequently, a pointer on a 32-bit machine is 32 bits (4 bytes).
  - With a 64-bit executable, a pointer would be 64 bits (8 bytes).
- All pointers in a program have the same size even though they can point to large and small types.

```
int main() {
    int a{5};
    double b{5.0};
    int *a_ptr{&a};
    double *b_ptr{&b};
    std::cout << "size of a_ptr: " << sizeof a_ptr << std::endl;
    std::cout << "size of b_ptr: " << sizeof b_ptr << std::endl;
}
```

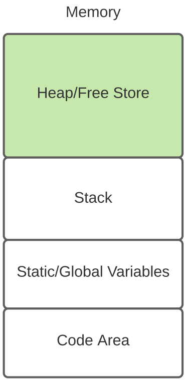
Pointers | Quiz

```
int main(){
    int x{10};//--stored at address 1000
    int *p;//--stored at address 2000
    vector<double> *q{nullptr};

    cout << x << endl;
    cout << p << endl;
    cout << &p << endl;
    cout << &x << endl;
    cout << sizeof p << endl;
    cout << sizeof q << endl;
    p = nullptr;
    cout << &p << endl;
    cout << p << endl;
    q = &x;
}
```

Heap Memory

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- Once you have allocated memory on the heap, you are responsible of deallocating that memory once you don't need it any more.
- If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes).
- Its better to use the heap when you know that you will need a lot of memory for your data, or you just are not sure how much memory you will need (like with a dynamic array).



Dynamic Memory Allocation

- Allocating memory on the heap needs to be explicitly performed by the programmer. To allocate data dynamically, we use the scalar (non-array) form of the `new` operator.

```
int main(){
    new int;
}
```

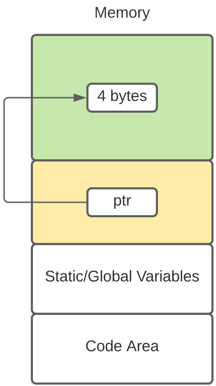
- We are requesting an integer's worth of memory from the OS. The `new` operator creates the object using that memory, and then returns a pointer containing the address of the memory that has been allocated.
- Most often, we'll assign the return value to our own pointer variable so we can access the allocated memory later. It should be clear now why we need pointers.

```
int main(){
    int *ptr{new int};
    *ptr = 5;
    int *my_ptr{new int{6}};
}
```

Dynamic Memory Allocation

```
int main(){
    int *ptr{new int{5}};
}
```

- The pointer variable is on the stack.
- The allocated storage for the `int` is on the heap.
- The allocated storage does not have an identifier.
- The only way to access this storage is via the pointer (because the storage does not have an identifier).
- If you lose this pointer because it goes out of scope or you reassign it then you lose your only way to access this storage → memory leak.



Dynamic Memory Allocation

- Memory leaks happen when your program loses the address of some bit of dynamically allocated memory before giving it back to the OS.
- When this happens, your program cant delete the dynamically allocated memory, because it no longer knows where it is.
- The OS also can't use this memory, because that memory is considered to be still in use by your program.
- Memory leaks eat up free memory while the program is running, making less memory available not only to this program, but to other programs as well.
- Programs with severe memory leak problems can eat all the available memory, causing the entire machine to run slowly or even crash. Only after your program terminates is the OS able to clean up and "reclaim" all leaked memory.

Dynamic Memory Allocation

- Not using `delete` on allocated storage on the heap leads to memory leak.

```
int main(){
    int i{10};
    int *ptr{new int{5}};
    std::cout << ptr << std::endl;//--0x55972558fe70
    std::cout << *ptr << std::endl;//--5
    ptr = &i;//-- reassigning pointer, lose dynamic storage
    std::cout << ptr << std::endl;//--0x7ffd936afe1c
    std::cout << *ptr << std::endl;//--10
}
```

```
int main(){
    int *ptr{new int{5}};
}
```

Dynamic Memory Allocation

- To prevent memory leak, you MUST deallocate the storage you allocated with `new` and return it back to the OS.
- To deallocate or free storage we use the `delete` operator.

```
int main(){
    int *ptr{new int{5}};
    /*Do some work with the pointer*/
    delete ptr;
}
```

- Warning: "Only `delete` what you `new`". Deleting a pointer that is not pointing to dynamically allocated memory may crash your program.

Dynamic Memory Allocation

- The `delete` operator does not actually delete anything. It simply returns the memory being pointed to back to the OS.
- The OS is then free to reassign that memory to another application (or to this application again later).
- `delete` does not delete the pointer variable. This means that you can reassign a new value just like any other variable.
- Warning: "Only `delete` what you `new`". Deleting a pointer that is not pointing to dynamically allocated memory will crash your program.

Dynamic Memory Allocation | Dangling Pointers

- C++ does not make any guarantees about what will happen to the contents of deallocated memory, or to the value of the pointer being deleted.
- In most cases, the memory returned to the operating system will contain the same values it had before it was returned, and the pointer will be left pointing to the now deallocated memory.
- A pointer that is pointing to deallocated memory is called a dangling pointer.
- Dereferencing or deleting a dangling pointer will lead to undefined behavior.
- *Best practice*: Make the pointer point to either `nullptr` or to another variable after memory deallocation.

```
int main(){
    int *ptr{new int{5}};
    /*Do some work with the pointer*/
    delete ptr;
    ptr = nullptr;
}
```

```
int main(){
    int x{};
    int *ptr{new int{5}};
    /*Do some work with the pointer*/
    delete ptr;
    ptr = &x;
    *ptr = 3;
    std::cout << x;
}
```

Dynamic Memory Allocation | Valgrind

- Valgrind is an instrumentation framework for building dynamic analysis tools. It comes with a set of tools that can automatically detect many memory management and threading bugs.
- You need to install it first with `sudo apt-get install valgrind` and read about its integration in Clion.
- Run Valgrind on the following code.

```
int main(){
    std::cout << new int << std::endl;
    int *ptr{new int};
    int *my_ptr{new int{6}};
}
```

Dynamic Memory Allocation | Quiz

- Question: Does the following program contain a dangling pointer?

```
int main(){
    int *ptr{nullptr};
    ptr = new int;
    *ptr = 5;
    ptr = nullptr;
    delete ptr;
}
```

Dynamic Memory Allocation | Arrays

- Dynamic memory allocation for C-style arrays.

```
int main(){
    int *array_ptr{nullptr};
    size_t size{};
    cout << "How many entries in the array? ";
    cin >> size;
    //allocate that many integers contiguously on the heap
    array_ptr = new int[size];
    delete [] array_ptr;//--free allocated storage
}
```

Pointer Comparisons

- Determine if two pointers point to the same location (not the data where they point).

```cpp
int main() {
    std::string s1{"Hello"};
    std::string s2{"Hello"};

    std::string *p1{&s1};
    std::string *p2{&s2};
    std::string *p3{&s1};

    std::cout << (p1 == p2) << std::endl; //--false
    std::cout << (p1 == p3) << std::endl; //--true
}
```

Pointer Comparisons

- Determine if two pointers point to the same data.

```cpp
int main() {
    std::string s1{"Hello"};
    std::string s2{"Hello"};

    std::string *p1{&s1};
    std::string *p2{&s2};
    std::string *p3{&s1};

    std::cout << (*p1 == *p2) << std::endl; //--true
    std::cout << (*p1 == *p3) << std::endl; //--true
}
```

Constness

- There are several ways to qualify pointers using `const`.
  - Pointers to constants.
    - The data pointed to by the pointers is constant and cannot be changed.
    - The pointer itself can change and point somewhere else.
  - Constant pointers.
    - The data pointed to by the pointers can be changed.
    - The pointer itself cannot change and cannot point somewhere else.
  - Constant pointers to constants.
    - The data pointed to by the pointers is constant and cannot be changed.
    - The pointer itself cannot change and cannot point somewhere else.

Constness | Pointers to Constants

- The data pointed to by the pointers is constant and cannot be changed.
- The pointer itself can change and point somewhere else.

```
int main() {
    int highest_score {100};
    int lowest_score {23};
    const int *ptr {&highest_score};

    *ptr = 90; //--error
    ptr = &lowest_score; //--ok
}
```

Constness | Constant Pointers

- The data pointed to by the pointers can be changed.
- The pointer itself cannot change and cannot point somewhere else.

```
int main() {
    int highest_score {100};
    int lowest_score {23};
    int *const ptr {&highest_score};

    *ptr = 90; //--ok
    ptr = &lowest_score; //--error
}
```

Constness | Constant Pointers to Constants

- The data pointed to by the pointers is constant and cannot be changed.
- The pointer itself cannot change and cannot point somewhere else.

```
int main() {
    int highest_score {100};
    int lowest_score {23};
    const int *const ptr {&highest_score};

    *ptr = 90; //--error
    ptr = &lowest_score; //--error
}
```

Pointers and Functions

- We can pass pointers to functions: This is called pass by pointer and a copy of the pointer is made in the function.
- The function parameter is a pointer.
- The argument can be a pointer or the address of a variable.

```
//--Argument is the address of a variable
void DoubleValue(int *int_ptr){
    *int_ptr *= 2;
}

int main(){
    int value{10};
    std::cout << value << std::endl;//--10
    DoubleValue(&value);
    std::cout << value << std::endl;//--20
}
```

Pointers and Functions

- We can pass pointers to functions: This is called pass by pointer and a copy of the pointer is made in the function.
- The function parameter is a pointer.
- The argument can be a pointer or the address of a variable.

```
//--Argument is a pointer
void DoubleValue(int *int_ptr){
    *int_ptr *= 2;
}
int main(){
    int value{10};
    int *ptr{&value};
    std::cout << value << std::endl;//--10
    DoubleValue(ptr);
    std::cout << value << std::endl;//--20
}
```

Pointers and Functions

- We can return pointers from functions.
- Safe: Return pointers to data that was passed in or memory dynamically allocated in the function.

```
//--Returning data that was passed in
int* LargestNumber (int *num1, int *num2){
    if (*num1 > *num2)
        return num1;
    else
        return num2;
}

int main(){
    int a{100}, b{200};
    int *largest_ptr{nullptr};
    largest_ptr = LargestNumber(&a,&b);
    std::cout << *largest_ptr << std::endl;//--dereference pointer: 200
}
```

Pointers and Functions

- We can return pointers from functions.
- Safe: Return pointers to data that was passed in or memory dynamically allocated in the function.

```
int *MyFunction (){
    int a{5};
    int *ptr = new int;
    *ptr = a;
    return ptr;
}

int main(){
    int *ptr{MyFunction()};
    delete ptr;
}
```

Pointers and Functions

- We can return pointers from functions.
- Unsafe: Return a pointer to a local variable.

```
int* Function1(){
    int var{5};
    return &var;
}

int* Function2(){
    int var{5};
    int *ptr{&var};
    return ptr;
}
```

References

- A reference variable is an alias, that is, another name for an already existing variable.
- Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.
- References are often confused with pointers but three major differences between references and pointers are:
  - You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.
  - Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
  - A reference must be initialized when it is created. Pointers can be initialized at any time.

Next Class | 10/08

- Lecture06: Smart Pointers.
- No Quiz.
- No Assignment.
- Stay safe!