COMP 322 Lecture 4 - Pointers & References

Junji Duan

2024/1/26

Today's Outline

- Pointers
- Pointer arithmetics
- References

Pointers - Introduction

- Regular variables:
  - Are locations in memory
  - When declared, a memory address is automatically assigned
  - We don't care about their physical address
  - Accessible by their names
- Use the address operator & to get the physical address of a variable

```
int var;
cout << &var;
```

Output would be something similar to 0x7ffc8e229ddc

- To store the address of a memory location in a variable, we need a special type of variables called pointer variable
  - Use the dereference operator * to declare a pointer variable
  - int *ptr = &var;
    - ptr is a pointer variable
    - ptr stores the address of the variable var
    - ptr is pointing to var
    - ptr and &var are exactly the same thing
    - *ptr and var are exactly the same thing
    - The type of a pointer variable should match the type of the variable whose address is being stored: var is of type int, so *ptr should be of type int as well.

- Spaces don't matter when declaring pointers. The following declarations are all equivalent:
  - int *ptr;
  - int* ptr;
  - int * ptr;
- Be careful when declaring multiple variables on the same line
  - int * ptr1, ptr2; wrong
    - Only one pointer is being declared
    - ptr2 is not a pointer, it is simply an integer variable
  - int *ptr1, *ptr2;
    - Both variables are pointers

Pointers - code example

What's the output of the following code?

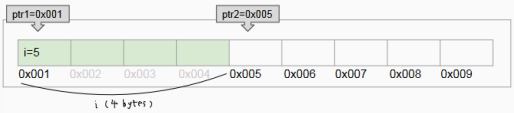
```
12 // main function
13 int main()
14 {
15     int a = 7, b = 9;
16     int *ptr1;
17     int *ptr2 = &b;
18     ptr1 = &a;
19
20     *ptr1 = 15;
21     *ptr2 = *ptr1;
22
23     cout << a << '\n'; 15
24     cout << b << '\n'; 15
25     return 0;
26 }
```

Pointer Arithmetics

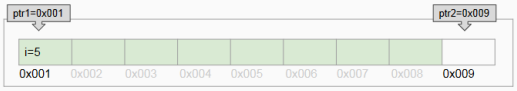
- Used to jump to different memory locations
- Only addition and subtraction (no multiplication and division)
- When you increment a pointer by one, it points to the next memory location
- Result depends on the size of the data type to which the pointer is pointing

- Imagine the memory as a table. Each cell is an element
- char *ptr; // ptr is a pointer of type char. Assuming that sizeof(char) = 1 byte
  - ++ptr; // will point to the next byte in the memory table (unless the compiler is applying memory padding or alignment for optimization)
- int *ptr; // ptr is a pointer of type int. Assuming that sizeof(int) = 4 bytes
  - ++ptr; // will point to a location that is 4 bytes away in the memory table

- int i = 5;
- int *ptr1 = &i; // ptr1 is a pointer of type int. Assuming that sizeof(int) = 4 bytes
- int *ptr2 = ++ptr1; // will point to a location that is 4 bytes away in the memory table



- double i = 5;
- double *ptr1 = &i; // ptr1 is a pointer of type double. Assuming that sizeof(double) = 8 bytes
- double *ptr2 = ++ptr1; // will point to a location that is 8 bytes away in the memory table



Pointers - Example 1

```
int main()
{
    int value = 100;
    int* pValue = &value;
    cout << "Value is equal to: " << *pValue << endl;
    cout << "Address of value = " << pValue << endl;
}
```

The output is:

```
Value is equal to: 100
Address of value = 0x7fff69d9b6b8
```

Pointers - Common mistakes

- Dereferencing invalid pointers
  - Uninitialized pointers point to random memory location

```
int main()
{
    int *ptr; // non-initialized pointer
              // points to random memory location
    *ptr = 12; // memory corruption

}
```

- Good practice to always:
  - assign pointers to NULL (or nullptr since Cx11) when they point to nothing
  - check if the pointer is not null before dereferencing it

```
int main()
{
    int *ptr = NULL;
    if (ptr != NULL)
    {
        cout << *ptr << endl;
    }
    else
    {
        cout << "pointer is NULL" << endl;
    }
}
```

- Mixing operator precedence rules to accidentally apply arithmetics on pointers instead of the value being pointed to
  - *++ptr; vs ++*ptr; // remember that ++ has higher precedence than *

Pointers - Example 2

```
int main()
{
    int value = 100;
    int* pValue = &value;
    cout << "Value is equal to: " << *pValue << endl;
    cout << "Address of value = " << pValue << endl;
    cout << ++*pValue << endl; // dereference the pointer
                               // then increment its value
    cout << *++pValue << endl; // increment the address of value
                               // then dereference the pointer
    cout << ++pValue << endl;  // increment the address of value
                               // to point to next memory location
}
```

The output is:

```
Value is equal to: 100
Address of value = 0x7ffcf1de7f34
101
-237076680
0x7ffcf1de7f3c
```

Pointers - Example 3

- What's wrong with the following function?

```
float *getPricePointer()
{
    float price = 9.99;
    return &price;
}
```

- What's wrong with the following function?
  - getPricePointer is returning the address of a local variable.
  - Local variables have limited scope and lifetime
  - price will be automatically destroyed as soon as the function returns
  - Its address will be pointing to an invalid memory location

```
float *getPricePointer()
{
    float price = 9.99;
    return &price;
}
```

Reference variables

- Reference variable is a C++ concept that doesn't exist in C
- Reference permits to assign multiple names to the same variable
- To declare a reference variable we use the address & operator
- int x;
- int &y = x; // be careful not to confuse with int *y = &x;
  - y is a reference to x
  - y is considered to be an alias for x
  - y and x are the same thing
  - y and x are two names for the same memory location

```
int main()
{
    int a = 100;
    int &b = a;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    b = 12;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

}
```

The output is:

```
a = 100
b = 100
a = 12
b = 12
```

Passing arguments by reference

By Value

```
#include <iostream>
using namespace std;

int getProduct(int x, int y)
{
    return x*y;
}

// main function
int main()
{
      int a = 4;
      int b = 5;
      int product = getProduct(a, b);
      cout << product;
}
```

By Reference

```
#include <iostream>
using namespace std;

int getProduct(int &x, int &y)
{
    return x*y;
}

// main function
int main()
{
      int a = 4;
      int b = 5;
      int product = getProduct(a, b);
      cout << product;
}
```

By Pointer

```
#include <iostream>
using namespace std;

int getProduct(int* x, int* y)
{
    return (*x)*(*y);
}

// main function
int main()
{
    int a = 4;
    int b = 5;
    int product = getProduct(&a, &b);
    cout << product;
}
```

By Reference

```
#include <iostream>
using namespace std;

int getProduct(int &x, int &y)
{
    return x*y;
}

// main function
int main()
{
    int a = 4;
    int b = 5;
    int product = getProduct(a, b);
    cout << product;
}
```

Reference variables vs pointers

- Reference variables must be initialized
  - int &x = var;
- Reference variable cannot be changed to reference another variable
  - Similar to constant pointers
- Unlike pointers, references cannot be NULL
- Pointer has its own memory address whereas a reference shares the same memory address with the variable it is referencing
- Reference variables are very commonly used as function parameters
  - Better performance by avoiding copying values
- References are safer than pointers so they are preferred to pointers whenever you have the choice (if there is no need for dynamic allocation)

Pointers - are they worth the headache?

- **Pointers are used for**
  - **Efficiency (no need to statically reserve a huge array in advance)**
  - **Implementation of complex data structures**
  - **Dynamic allocation of memory**
  - **Passing functions as parameters**
  - **Many advanced C++ techniques**
- **Misusing pointers is the mother of all software bugs**
  - **Memory leaks**
  - **Dangling pointers**
  - **Buffer overflow**
  - **Abduction by aliens ... :)**

Pointers - confusing the cat (C++ interview question)

- **What's the output of the following code: 4, 5**

```
int main()
{
    int *ptr = NULL;
    int i = 7;  i++;
    for(int j=0; j<=2; j++) {
        i = j;
    }
    ptr = &i;
    if (ptr != NULL) {
        (*ptr) *= (*ptr);
    }

    if (ptr != NULL) {
        cout << (*ptr)++ << endl;
        cout << i << endl;
    }
    else {
        cout << "pointer is NULL" << endl;
    }
}
```