Ch08-Pointers

August 25, 2020

1 Pointers

http://www.cplusplus.com/doc/tutorial/pointers/

1.1 Topics

- Computer Memory (RAM)
- Pointers - declaring and using pointers
- Pointer arithmetics
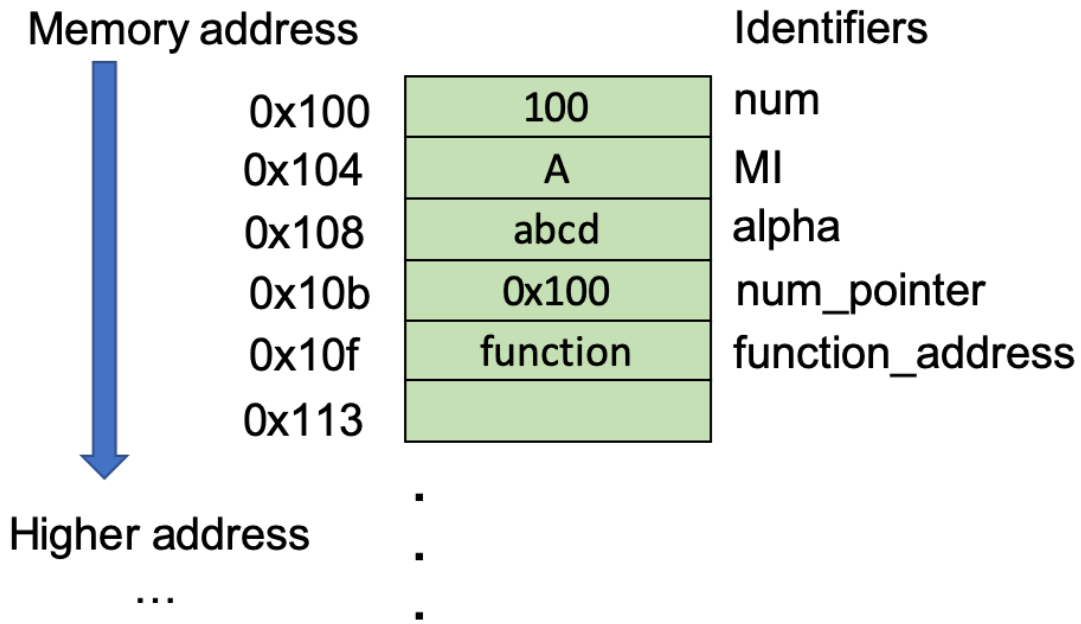- Dynamic memory (Heap)
- Function pointers

1.2 Headers

- run include headers and helper function cells if Kernel crashes or is restarted
- you do not need to include any special header to use pointers

```
[5]: // include headers
     #include <iostream>
     #include <string>

     using namespace std;
```
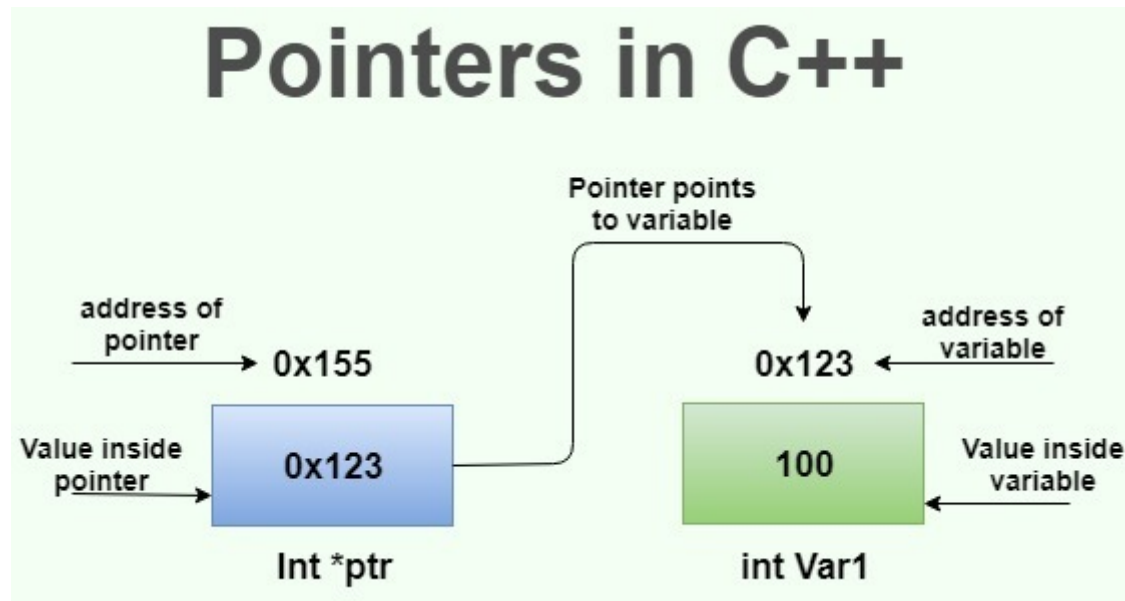
1.3 Computer Memory (RAM)

- the primary memory of computer is also called RAM (Random Access Memory)
- program must be loaded into RAM before it can be executed
- data must be loaded into RAM before program can use it
- literal values or variables are all store in memory
    - literaly values do not have identifiers to be used
- variables are programmer-controlled identifiers that maps to some memory location (address)
    - program uses memory address
    - programmers uses identifiers
- the following figure depicts a simple representation of RAM

Memory address		Identifiers
0x100	100	num
0x104	A	MI
0x108	abcd	alpha
0x10b	0x100	num_pointer
0x10f	function	function_address
0x113		

Higher address

…

Fig. A Simple View of Computer Memory

1.4 Pointers

- special variables that can store physical memory addresses of variables and identifiers
- variables represent values and are used interchangebly
- pointers represent memory address and and are used interchangebly
- like any variable you must declare a pointer before you can use it
- the following figure helps visualize pointer variable

Pointers in C++

Pointer points to variable

address of pointer → 0x155

address of variable ← 0x123

Value inside pointer → 0x123

Value inside variable ← 100

Int *ptr

int Var1

1.5 Pointer applications

- pointers are powerful features of C/C++ programs
- allows you to directly manipulate memory!
- there are many applications of pointers; some basic applications are described below

1.5.1 Address-of operator - &

- the address of a variable can be obtained by *address-of-operator (&)* infront of a variable name
- **&** is also used in function parameters for pass-by reference

```
[2]: int num = 100;
```

```
[3]: cout << "value of num = " << num << endl;
     cout << "address of num = " << &num << endl;
```

```
value of num = 100
address of num = 0x113196830
```

1.5.2 Dereference operator - *

- ***** (dereference operator)** can be used to read the **value pointed to by** some memory address

```
[4]: cout << "value pointed to by &num = " << *&num << endl;
```

```
value pointed to by &num = 100
```

1.6 Declaring pointers

- pointers can be declared using ***** de-reference/pointer operator
- syntax:

```
type * pointerVarName;
```

1.6.1 visualize pointers in pythontutor.com: https://goo.gl/zhCr3G

```
[10]: // declare pointers
      int num1; // variable NOT a pointer
      int * pNum1; // declare pNum1 of type int or pointer to int
      // declare and initialize pointers
      float * fltPtr = nullptr; // initialize with nullptr (pointing to no address)
      int * somePtr = &num1; // initialize somePtr with the address of num1
```

```
[18]: pNum1 = &num1; // assiging value to a pointer
      *pNum1 = 200; // dereferencing pNum1; assigning value to the location pointed
       ↪to by pNum1
      cout << "*pNum1 = " << *pNum1 << endl;
      cout << "pNum = " << pNum1 << endl;
```

```
cout << "num1 = " << num1 << endl;
cout << "&num1 = " << &num1 << endl;
```

```
*pNum1 = 200
pNum = 0x10e34df80
num1 = 200
&num1 = 0x10e34df80
```

1.7 Pointer arithmetic

- you can add or subtract values to or from pointers
  - pointers will simply point to a different memory location!
- one can move the pointer around pointing to various memory locations
  - that can be dangerous!

[19]:
```
pNum1 += 10; // add 10 to pNum1 value (address)
```

[19]: @0x7ffee5c388d0

[20]:
```
cout << "pNum1 = " << pNum1;
```

```
pNum1 = 0x10e34dfa8
```

[22]:
```
// now what value is pNum1 pointing to
cout << "*pNum1 = " << *pNum1;
```

```
*pNum1 = 0
```

[23]:
```
// let's subtract 10
pNum1 -= 10;
```

[24]:
```
cout << "pNum1 = " << pNum1 << endl;
cout << "*pNum1 = " << *pNum1;
```

```
pNum1 = 0x10e34df80
*pNum1 = 200
```

1.8 Invalid pointers and null pointers

- pointers are meant to point to valid addresses, in principle
- pointers can also point to any any address, including addresses that do not refer to any valid element
  - e.g., uninitialized pointers and pointers to non-existent elements of an array
- neither p nor q point to addresses known to contain a value in the following cell
- they do not cause error while declaring…
- but can cause error/problem if dereferenced such pointers
  - may crash program or point to a random data in memory

```
[2]: // invalid pointers
     int * p; // uninitialized pointer
     int some_num; // uninitialized variable
```

```
[7]: p = (&some_num); // add 10 to address of some_num
```

```
[7]: @0x7ffee5c388d0
```

```
[8]: cout << *p << endl;
```

```
0
```

```
[18]: cout << * << endl;
```

```
0
```

1.9 Dynamic memory

- memory needs from auto/local variables are determined during compile time before program executes
- at times memory needs of a program can only be determined during runtime
    - e.g., when memory needed depends on user input
- on these cases, program needs to dynamically allocate memory
- pointers are used along with other keywords **new** and **delete** to allocate and deallocate dynamic memory
- dynamic memory is allocated in **heap** segment
    - unlike regular auto variables that are declared on **stack**
- dynamic memory must be deallocated to prevent memory leak in the program
- syntax to allocate and deallocate dynamic memory:

```
// allocate memory
type * pointer = new type;
```

```
//deallocate memory
delete pointer;
```

- visualize in pythontutor.com: https://goo.gl/5qse7L

```
[27]: // allocate dynamic memory
      int * numb1 = new int;
      int * numb2 = new int;
```

```
[28]: // use dynamic memory
      *numb1 = 100;
      *numb2 = 50;
      cout << *numb1 << " + " << *numb2 << " = " << *numb1 + *numb2 << endl;
      cout << *numb1 << " - " << *numb2 << " = " << *numb1 - *numb2 << endl;
      cout << *numb1 << " * " << *numb2 << " = " << *numb1 * *numb2 << endl;
```

```
100 + 50 = 150
100 - 50 = 50
100 * 50 = 5000
```

[29]:
```cpp
// delete dynamic memory
// intialize them to nullptr just incase garbage collector has not deallocated␣
 ↪numb1 and numb2 yet!
numb1 = nullptr;
numb2 = nullptr;
delete numb1;
delete numb2;
```

1.10   Passing pointers to functions

- pointers can be passed to functions
- similar to passed-by-reference
    - if value pointed to by formal pointer parameter is changed, the value pointed to by actual pointer parameter will also be changed!
- pass pointers as constants (read-only) to prevent the side effect

[31]:
```cpp
// example 1: pass address of regular variables
int n1, n2 = 0;
```

[32]:
```cpp
n1 = 10; n2 = 15;
cout << n1 << " + " << n2 << " = " << addInts(&n1, &n2) << endl;
```

```
10 + 15 = 25
```

[33]:
```cpp
// example 2: pass ptr/dynamic variables
int * ptr1 = new int;
int * ptr2 = new int;
```

[34]:
```cpp
*ptr1 = 100;
*ptr2 = 200;
cout << *ptr1 << " + " << *ptr2 << " = " << addInts(ptr1, ptr2) << endl;
```

```
100 + 200 = 300
```

[35]:
```cpp
// side effect example!
int myAdd(int * p1, int * p2) {
    *p1 = 1000;
    *p2 = 2000;
    return *p1 + *p2;
}
```

[36]:
```cpp
cout << *ptr1 << " + " << *ptr2 << " = " << myAdd(ptr1, ptr2) << endl;
cout << *ptr1 << " + " << *ptr2 << endl; // values of *ptr1 and *ptr2 have been␣
 ↪changed by myAdd!
```

```
100 + 200 = 3000
1000 + 2000
```

[36]: @0x10e2fdec0

[37]:
```cpp
// prevent side effect by passing pointers as const (read-only)
int myAddBetter(const int * p1, const int * p2) {
    *p1 = 1000; // not allowed as compiler will throw error!
    *p2 = 2000; // not allowed!
    return *p1 + *p2;
}
```

input_line_51:3:9: error: read-only variable is not

assignable
```
    *p1 = 1000; // not allowed as compiler will throw error!
    ~~~ ^
```

input_line_51:4:9: error: read-only variable is not

assignable
```
    *p2 = 2000; // not allowed!
    ~~~ ^
```

Interpreter Error:

[38]:
```cpp
// prevent side effect by passing pointers as const (read-only)
int myAddBetter(const int * p1, const int * p2) {
    return *p1 + *p2;
}
```

[39]:
```cpp
*ptr1 = 100;
*ptr2 = 200;
cout << *ptr1 << " + " << *ptr2 << " = "
    << myAddBetter(ptr1, ptr2) << endl;
cout << *ptr1 << " + " << *ptr2 << endl;
// values of *ptr1 and *ptr2 guaranteed to stay the same!
```

```
100 + 200 = 300
100 + 200
```

[39]: @0x10e2fdec0

1.11 Pointers to functions

- pointers can store addresses of functions as well; called function pointers

7

- used for passing a function as an argument to another higher order function
- declaring function pointer is very similar to declaring variable pointers
- parenthesis around function pointer name is required!

```
type (* functionPtrName) ( parameter list... );
```

[25]:
```cpp
int addition (int a, int b) {
    return (a + b);
}
```

[30]:
```cpp
// function that takes two int pointers
int addInts(int * p1, int * p2) {
    return *p1 + *p2;
}
```

[26]:
```cpp
int subtraction (int a, int b) {
    return (a - b);
}
```

[47]:
```cpp
int m, n;
// function pointer; copy the address of subtraction into sub function pointer
int (*sub)(int, int) = subtraction;
```

[49]:
```cpp
// calling a function pointer is very similar to calling a function
cout << (*sub)(10, 20) << endl;
cout << subtraction(10, 20);
```

```
-10
-10
```

[17]:
```cpp
// passing function to a function!
// operation function takex 3 arguments
// two integers and one function pointer
int operation (int x, int y, int (*func)(int, int)) {
  int ans;
  ans = (*func)(x, y); // dereferece function; call func and store the result␣
  ↪in g
  return ans;
}
```

[19]:
```cpp
n = operation(100, m, sub);
cout << "m = " << m << endl;
cout << "n = " << n << endl;
```

```
m = 30
n = 70
```

1.12   Exercises

1. Write a program using dynamic memory that determines area and circumference of a circle.
   - must use functions to find the required answers
   - prompt user to enter radius of a circle

```
[41]:  // Solution to exercise 1
       #include <iostream>
       #include <cmath>

       using namespace std;
```

```
[42]:  float areaOfCircle(float * radius) {
           return M_PI * pow(*radius, 2);
       }
```

```
[43]:  float circumference(float * radius) {
           return 2 * M_PI * (*radius);
       }
```

```
[44]:  void solve() {
           float * radius = new float;
           cout << "Enter radius of a circle: ";
           cin >> *radius;
           cout << "radius of the circle: " << *radius << endl;
           cout << "area of the circle: " << areaOfCircle(radius) << endl;
           cout << "circumference of the circle: " << circumference(radius) << endl;
           // deallocate radius memory
           radius = nullptr;
           delete radius;
       }
```

```
[45]:  // you'd call this function in main() in a program file
       solve();
```

```
Enter radius of a circle: 5
radius of the circle: 5
area of the circle: 78.5398
circumference of the circle: 31.4159
```

2. Write a program using dynamic memory that determines area and perimeter of a rectangle.
   - must use functions to find area and perimeter
   - prompt user to enter length and widht of a rectangle

1.13   Summary

- learned about the basics of RAM and pointers
- declaring and using pointers
- function pointers and passing pointers to functions

- exercises and sample solutions

[ ]: