Pointers

October 20, 2021

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

[1]:
```cpp
// 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 stored in memory
  - literal values do not have identifiers associated with them
- variables are programmer-controlled identifiers that maps to some memory location (address)
  - CPU uses memory addresses
  - programmers use identifiers/variables
- the following figure depicts a simple representation of RAM

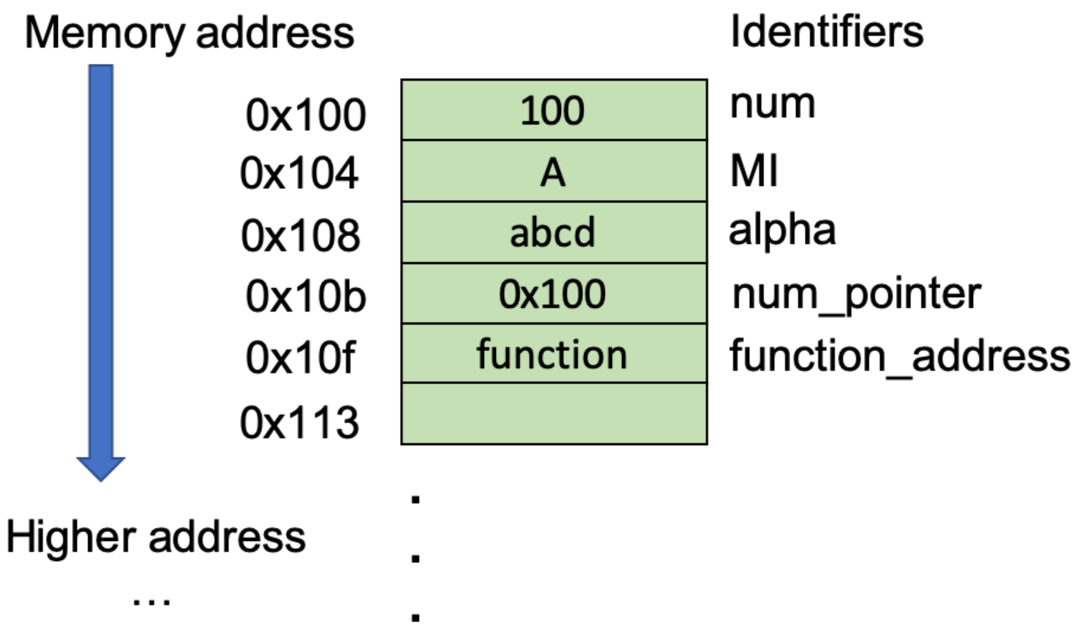
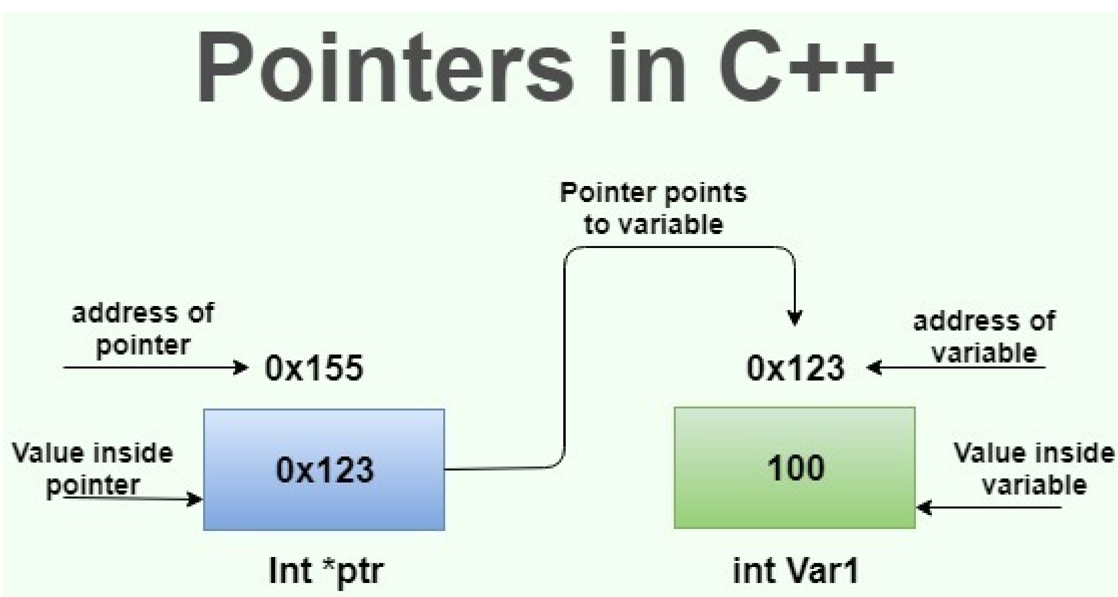


Fig. A Simple View of Computer Memory

1.4 Pointers

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



1.5 Pointer applications

- pointers are powerful features of C/C++ programming language
- pointers allow programmers to directly manipulate memory!
- there are many advanced applications of pointers; some basic examples are demonstrated below

1.5.1 Address of operator (&)

- the address of a variable can be obtained by *address-of-operator (& - ampersand symbol)* 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 = 0x10f2e0630
```

1.5.2 Dereference operator (*)

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

[4]:
```
// what is stored at the address of num?
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

[5]:
```
// 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
```

[6]:
```
pNum1 = &num1; // assigning value to a pointer
*pNum1 = 200; // dereferencing pNum1; assigning value to the location pointed␣
↪to by pNum1
```

[6]: 200

[7]:
```
// access values of variables and pointers
cout << "*pNum1 = " << *pNum1 << endl;
cout << "pNum = " << pNum1 << endl;
cout << "num1 = " << num1 << endl;
cout << "&num1 = " << &num1 << endl;
```

```
*pNum1 = 200
pNum = 0x10f2e0a60
num1 = 200
&num1 = 0x10f2e0a60
```

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 from security point of view!

[8]:
```
pNum1 += 10; // add 10 to pNum1 value (address)
```

[8]: @0x7ffee0f6a560

[9]:
```
cout << "pNum1 = " << pNum1;
```

```
pNum1 = 0x10f2e0a88
```

[10]:
```
// now what value is pNum1 pointing to
cout << "*pNum1 = " << *pNum1;
```

```
*pNum1 = 53058559
```

[11]:
```
// let's subtract 10
pNum1 -= 10;
```

[12]:
```
cout << "pNum1 = " << pNum1 << endl;
cout << "*pNum1 = " << *pNum1;
```

```
pNum1 = 0x10f2e0a60
*pNum1 = 200
```

1.8 Invalid pointers and null pointers

- pointers are meant to point to valid addresses, in principle
- however, pointers can 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 valid 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

[13]:
```
// invalid pointers
int *p, *q; // uninitialized pointer
int some_num; // uninitialized variable
```

[14]:
```
p = &some_num;
```

[15]:
```
cout << *p << endl;
```

```
0
```

[16]:
```
// add 10 to address of some_num
p += 10;
```

[17]:
```
cout << *p << endl;
```

```
0
```

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

```
input_line_30:2:11: warning: null passed to a callee
that requires a non-null argument [-Wnonnull]
 cout << *q << endl;
          ^



        Interpreter Exception:
```

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 the runtime
  - e.g., when amount and type of memory needed depends on user input
- in such 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** segment
- 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;
```

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

[19]:
```
// allocate dynamic memory
int * numb1 = new int;
int * numb2 = new int;
```

[20]:
```
// 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
```

[21]:
```
// 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

[22]:
```
// function that takes two int pointers
int addInts(int * p1, int * p2) {
    return *p1 + *p2;
}
```

[23]:
```
// example 1: pass address of regular variables
int n1, n2 = 0;
```

[24]:
```
n1 = 10; n2 = 15;
cout << n1 << " + " << n2 << " = " << addInts(&n1, &n2) << endl;
```

```
10 + 15 = 25
```

[25]:
```
// example 2: pass addresses of dynamic variables/pointers
int * ptr1 = new int;
int * ptr2 = new int;
```

[26]:
```
*ptr1 = 100;
*ptr2 = 200;
cout << *ptr1 << " + " << *ptr2 << " = " << addInts(ptr1, ptr2) << endl;
```

```
100 + 200 = 300
```

[27]:
```
// side effect example!
int myAdd(int * p1, int * p2) {
    *p1 = 1000;
    *p2 = 2000;
    return *p1 + *p2;
}
```

[28]:
```
cout << *ptr1 << " + " << *ptr2 << " = " << myAdd(ptr1, ptr2) << endl;
```

```
100 + 200 = 3000
```

[29]:
```
// however, values pointed to by ptr1 and ptr2 have been changed by myAdd!
cout << *ptr1 << " + " << *ptr2 << endl;
```

```
1000 + 2000
```

[30]:
```
// 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_42:3:9: error: read-only variable is not

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

input_line_42:4:9: error: read-only variable is not

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

```
~~~ ^
```

```
Interpreter Error:
```

[31]:
```
// prevent side effect by passing pointers as const (read-only)
int myAddBetter(const int * p1, const int * p2) {
    return *p1 + *p2;
}
```

[32]:
```
*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
```

[32]: `@0x10edd0ed0`

1.11 Pointers to functions

- pointers can store addresses of functions as well; called function pointers
- used for passing a function as an argument to another higher order function
- declaring function pointer is very similar to declaring functions
- parenthesis around function pointer name is required!
- syntax:

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

[33]:
```
// function that takes two integers and returns the sum
int addition (int a, int b) {
    return (a + b);
}
```

[34]:
```
int subtraction (int a, int b) {
    return (a - b);
}
```

[35]:
```
int m, n;
// function pointer; copy the address of subtraction into sub function pointer
int (*sub)(int, int) = subtraction;
```

[36]:
```
// calling a function pointer is very similar to calling a function
cout << (*sub)(10, 20) << endl;
cout << subtraction(10, 20);
```

```
-10
-10
```

[37]:
```
// passing function to a function!
// operation function takes 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;
}
```

[38]:
```
n = operation(100, m, sub);
cout << "m = " << m << endl;
cout << "n = " << n << endl;
```

```
m = 0
n = 100
```

1.12 Labs

1. The following lab demonstrates the usage of pointers, enum type and user-defined namespace.
   - use `pointers.cpp` file found in labs/pointers folder as a hint to complete the program
   - use Makefile to compile and build the program
   - fix al the FIXMEs and write #FIXED next to each fixme once fixed

1.13 Exercises

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

[39]:
```
// Solution to exercise 1
#include <iostream>
#include <cmath>

using namespace std;
```

[40]:
```
float areaRectangle(float * length, float * width) {
    return (*length) * (*width);
}
```

[42]:
```
float perimeterRectangle(float * length, float * width) {
    return 2*(*length + *width);
}
```

[43]:
```
void solve() {
    float * length = new float; //dynamic memory
    float * width = new float; //dynamic memory
    cout << "Enter length and width of a rectangle separated by space: ";
    cin >> *length >> *width;
    cout << "rectangle dimension: " << *length << " x " << *width << endl;
    cout << "area of the rectangle: " << areaRectangle(length, width) << endl;
    cout << "perimeter of the rectangle: " << perimeterRectangle(length, width)␣
 ↪<< endl;
    // deallocate dynamic memory pointed to by length and width
    length = nullptr;
    width = nullptr;
    delete length;
    delete width;
}
```

[44]:
```
// you'd call this function in main() in a complete C++ program file
solve();
```

```
Enter length and width of a rectangle separated by space: 10 5
rectangle dimension: 10 x 5
area of the rectangle: 50
perimeter of the rectangle: 30
```

1.13.1 a complete demo program

- complete C++ using all the concepts covered so far using pointers and dynamic memory is provided here demos/pointers/rectangle

2. 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

1.14 Kattis problems

- pointers and dynamic variables are not requirement to solve any Kattis problems
- as you solve harder problems requiring advanced data structrues and algorithms, you'll naturally use pointers

1.15 Summary

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

```
[ ]:
```