Pointers

August 8, 2020

1 Pointers

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

1.1 Table of Contents

1.2 Headers and helper functions

- run include headers and helper function cells if Kernel crashes or is restarted

[1]:
```
// include headers
#include <iostream>
#include <string>
#include <ctime>

using namespace std;
```

[2]:
```
// printArray helper function
template<class T>
void printArray(T v[], int len) {
    char comma[3] = {'\0', ' ', '\0'};
    cout << '[';
    for (int i=0; i<len; i++) {
        cout << comma << v[i];
        comma[0] = ',';
```

```
    }
    cout << ']';
}
```

1.3 Pointers

- special variables that store physical memory addresses of data or other variables
- like any variable you must declare a pointer before you can work with it

1.4 Address-of operator (&)

- the address of a variable can be obtained by *address-of-operator (&)* infront of a variable name

[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 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;`

- visualize in pythontutor.com: https://goo.gl/zhCr3G

[3]:
```
// 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
```

[4]:
```
pNum1 = &num1; // assiging value to a pointer
*pNum1 = 200;
cout << "*pNum1 = " << *pNum1 << endl;
```

```
cout << "pNum = " << pNum1 << endl;
cout << "num1 = " << num1 << endl;
cout << "&num1 = " << &num1 << endl;
```

```
*pNum1 = 200
pNum = 0x107ee9830
num1 = 200
&num1 = 0x107ee9830
```

1.7 Pointers and arrays

- concept of arrays is related to that of pointers
- arrays work very much like pointers where index is used to deference the address of each cell

[5]:
```
int intarray[5];
int * ptr;
int i;
```

[6]:
```
printArray(intarray, 5);
```

```
[0, 0, 0, 0, 0]
```

[7]:
```
ptr = intarray; // copy base address of intarray to ptr
i = 0;
*ptr = 10; // same as intarray[i] = 10;
cout << ptr << " == " << intarray << endl;
cout << *ptr << " == " << intarray[i] << endl;
```

```
0x107ee9e10 == 0x107ee9e10
10 == 10
```

[8]:
```
// pointer arithmetic + and - (adds/subtracts size of pointer type)
ptr++;
i++;
*ptr = 20; // same as intarray[i] = 20;
cout << ptr << " == " << intarray+i << endl;
cout << *ptr << " == " << intarray[i] << " == " << *(intarray+i) << endl;
```

```
0x107ee9e14 == 0x107ee9e14
20 == 20 == 20
```

[9]:
```
ptr++;
i++;
*ptr = 30; // same as intarray[i] = 30;
cout << ptr << " == " << intarray+i << endl;
cout << *ptr << " == " << intarray[i] << " == " << *(intarray+i) << endl;
```

```
0x107ee9e18 == 0x107ee9e18
30 == 30 == 30
```

[10]:
```
ptr++;
i++;
*ptr = 40; // same as intarray[i] = 40;
cout << ptr << " == " << intarray+i << endl;
cout << *ptr << " == " << intarray[i] << " == " << *(intarray+i) << endl;
```

```
0x107ee9e1c == 0x107ee9e1c
40 == 40 == 40
```

[11]:
```
ptr++;
i++;
*ptr = 50; // same as intarray[i] = 50;
cout << ptr << " == " << intarray+i << endl;
cout << *ptr << " == " << intarray[i] << " == " << *(intarray+i) << endl;
```

```
0x107ee9e20 == 0x107ee9e20
50 == 50 == 50
```

[12]:
```
// look at all the elements of intarray
printArray(intarray, 5)
```

```
[10, 20, 30, 40, 50]
```

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

[16]:
```
// invalid pointers
int * p; // uninitialized pointer
int myarray[10];
int * q = myarray+20; //element out of bounds
```

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

```
input_line_25:2:11: warning: null passed to a callee
that requires a non-null argument [-Wnonnull]
 cout << *p << endl;
          ^
```

Interpreter Exception:

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

```
0
```

1.9 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 variable pointers
- parenthesis around function pointer name is required!

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

[15]:
```
int addition (int a, int b) {
    return (a + b);
}
```

```
input_line_23:1:5: error: redefinition of
'addition'
int addition (int a, int b) {
    ^

input_line_21:1:5: note: previous definition is
here
int addition (int a, int b) {
    ^
```

Interpreter Error:

[16]:
```
int subtraction (int a, int b) {
    return (a - b);
}
```

```
input_line_24:1:5: error: redefinition of
'subtraction'
int subtraction (int a, int b) {
    ^

input_line_22:1:5: note: previous definition is
here
int subtraction (int a, int b) {
```

```
^

Interpreter Error:
```

[17]:
```
int operation (int x, int y, int (*func)(int, int)) {
  int g;
  g = (*func)(x, y); // dereferece func
  return g;
}
```

[18]:
```
int m, n;
// function pointer
int (*sub)(int, int) = subtraction;
```

[19]:
```
m = operation(10, 20, addition);
n = operation(100, m, sub);
cout << "m = " << m << endl;
cout << "n = " << n << endl;
```

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

1.10 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
- 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;
type * arr = new type[num_of_elements]; // num_of_elements can be a variable

//deallocate memory
delete pointer;
delete[] arr;
```

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

[26]:
```
// allocate dynamic memory
int * numb1 = new int;
int * numb2 = new int;
```

[27]:
```
// 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]:
```
// 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;
```

[30]:
```
// array example
size_t size = 5; // this value can be determined during program execution from␣
 ↪user input e.g.
float * tests = new float[size];
```

[31]:
```
// dynamic array is no different from static array
tests[0] = 100;
tests[1] = 95;
tests[2] = 0;
tests[3] = 89;
tests[4] = 79;
```

[32]:
```
printArray(tests, size);
```

```
[100, 95, 0, 89, 79]
```

1.11 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 side effect
- arrays can be passed as pointers

[33]:
```
// function that takes two int pointers
int addInts(int * p1, int * p2) {
    return *p1 + *p2;
}
```

[34]:
```
// example 1: pass address of regular variables
int n1, n2 = 0;
```

[35]:
```
n1 = 10; n2 = 15;
cout << n1 << " + " << n2 << " = " << addInts(&n1, &n2) << endl;
```

```
10 + 15 = 25
```

[36]:
```
// example 2: pass ptr/dynamic variables
int * ptr1 = new int;
int * ptr2 = new int;
```

[37]:
```
*ptr1 = 100;
*ptr2 = 200;
cout << *ptr1 << " + " << *ptr2 << " = " << addInts(ptr1, ptr2) << endl;
```

```
100 + 200 = 300
```

[38]:
```
// side effect example!
int myAdd(int * p1, int * p2) {
    *p1 = 1000;
    *p2 = 2000;
    return *p1 + *p2;
}
```

[39]:
```
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
```

[39]:
```
@0x112ec2010
```

[40]:
```
// 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_53:3:9: error: read-only variable is not
assignable
    *p1 = 1000; // not allowed as compiler will throw error!
    ~~~ ^
input_line_53:4:9: error: read-only variable is not
assignable
    *p2 = 2000; // not allowed!
    ~~~ ^
```

Interpreter Error:

[41]:
```
// prevent side effect by passing pointers as const (read-only)
int myAddBetter(const int * p1, const int * p2) {
    return *p1 + *p2;
}
```

[42]:
```
*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
```

[42]:
```
@0x112ec2010
```

[62]:
```
// passing array to function as pointer
int arr[4] = {100, 200, 300, 400};
```

[63]:
```
// similar to int sumArray(int a[], int len)
int sumArray(const int * a, int len) {
    int s = 0;
    for(int i=0; i<len; i++) {
        s += a[i]; // s += *a; a++;
    }
    return s;
}
```

[65]:
```
cout << "sum of arr = " << sumArray(arr, 4) << endl;
```

```
sum of arr = 1000
```

1.12 Returning array from function

- since we can return a pointer from a function, we can return base address of array!
- caveat is that the local variable that is being returned has to be static!

[67]:
```
int * getRandomNumbers() {
    static int rands[5];
    // set the seed
    srand(time(nullptr));
    for (int i=0; i< 5; i++) {
        rands[i] = rand() % 100; // number between 0 and 99
    }
    return rands;
}
```

[71]:
```
int *rng;;
```

```
input_line_87:2:7: error: redefinition of 'rng'
 int *rng = getRandomNumbers();
      ^

input_line_85:2:7: note: previous definition is
here
 int *rng = getRandomNumbers();
      ^
```

Interpreter Error:

[76]:
```
rng = getRandomNumbers()
```

[76]:
```
@0x7fff4fcd7708
```

[77]:
```
for(int i=0; i< 5; i++) {
    cout << *(rng++) << endl; //post-increment
}
```

```
60
34
39
79
58
```

1.13 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

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

using namespace std;
```

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

[3]:
```
float circumference(float * radius) {
    return 2 * M_PI * (*radius);
}
```

[4]:
```
void solve() {
    float * radius = new float;
    cout << "Enter radius of circle: ";
    cin >> *radius;
    cout << "radius of circle: " << *radius << endl;
    cout << "area of circle: " << areaOfCircle(radius) << endl;
    cout << "circumference of circle: " << circumference(radius) << endl;
    // deallocate radius memory
    radius = nullptr;
    delete radius;
}
```

[5]:
```
// you'd call this function in main() in a program file
solve();
```

```
Enter radius of circle: 5
radius of circle: 5
area of circle: 78.5398
circumference of 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

1.14 Pointers to struct types

- pointers can also store memory addresses of user-defined struct types
- pointers user -> arrow operator to access members of struct types

[48]:
```
// Student struct type
struct Student {
    int id;
    string name;
```

```
    float tests[3];
};
```

[49]:
```
// declare dynamic variable / allocate memory dynamically
Student * st = new Student{100, "John S. Smith", {100, 0, 95}};
```

[50]:
```
cout << "name = " << st->name << endl;
```

```
name = John S. Smith
```

1.15 Passing struct pointers to functions

[51]:
```
// passing pointer-type to a function
void printStudent(Student * s) {
    cout << "id: " << s->id << " name: " << st->name << endl;
    cout << "test scores: ";
    printArray(s->tests, 3);
}
```

[52]:
```
printStudent(st);
```

```
id: 100 name: John S. Smith
test scores: [100, 0, 95]
```

[53]:
```
Student s1 = {200, "Jane Doe", {100, 100, 100}};
Student *sptr = &s1; // sptr is an alias to s1
printStudent(sptr);
```

```
id: 200 name: John S. Smith
test scores: [100, 100, 100]
```

[55]:
```
// passing pointer-type to a function
// a value pointed to by formal parameter is modified!
// may produce unintended side effect!!
void printStudent1(Student * s) {
    s->id = 999;
    cout << "id: " << s->id << " name: " << st->name << endl;
    cout << "test scores: ";
    printArray(s->tests, 3);
}
```

```
input_line_71:4:6: error: redefinition of
'printStudent1'
void printStudent1(Student * s) {
     ^

input_line_70:4:6: note: previous definition is
here
```

```
void printStudent1(Student * s) {
     ^
```

```
        Interpreter Error:
```

[56]:
```
printStudent1(st);
```

```
id: 999 name: John S. Smith
test scores: [100, 0, 95]
```

[57]:
```
// check the value of st; it should be changed to 999!
printStudent(st);
```

```
id: 999 name: John S. Smith
test scores: [100, 0, 95]
```

[58]:
```
cout << st->id << endl;
```

```
999
```

[59]:
```
// pass as a constant to prevent unintended side effect when passing pointer␣
 ↪type
void printStudent2(const Student * s) {
    s->id = 999; // not allowed!
    cout << "id: " << s->id << " name: " << st->name << endl;
    cout << "test scores: ";
    printArray(s->tests, 3);
}
```

```
input_line_75:3:11: error: cannot assign to variable
's' with const-qualified type 'const Student *'
    s->id = 999; // not allowed!
    ~~~~~ ^
input_line_75:2:36: note: variable 's' declared const
here
void printStudent2(const Student * s) {
                   ~~~~~~~~~~~~~~~~^
```

```
        Interpreter Error:
```

[60]:
```
// pass as a constant to prevent unintended side effect when passing pointer␣
 ↪type
// const makes the parameter read-only!!
void printStudent3(const Student * s) {
    cout << "id: " << s->id << " name: " << st->name << endl;
    cout << "test scores: ";
    printArray(s->tests, 3);
}
```

[61]:
```
printStudent3(st);
```

```
id: 999 name: John S. Smith
test scores: [100, 0, 95]
```

[ ]: