Getting Ready for Canadian Computing Competition

Class 5
December 28, 2019

Outline

- Pointers
- Passing by reference
  - Integers
  - Vectors
  - Structures
  - Arrays
- Const keyword
- Dangling Pointers

Pointers

- variables are locations in computer memory which can be accessed by their identifier (variable name)
- Each variable can be located in the memory by its address

Pointer Operator Summary

- & is the address-of operator, and can be read simply as "address of“
- * is the dereference operator, and can be read as "value pointed to by“
- They are complementary: an address obtained with & can be dereferenced with *

Declaring Pointers

- syntax: type * name;
- where type is the data type that the pointer points to

- Example

Int * num_p; // int pointer

Char * char_p; // char pointer

- Although they point to different data types that have difference sizes, they are just pointers
- All pointers take up 4 bytes of memory

Exercise

What will the following print? (Note: sizeof (int) == 4, sizeof (char) == 1)

```
int* num_p;
char *char_p;
vector <int> vec = {1,2,3};
vector <int> *vec_p = &vec;
cout <<sizeof (num_p) << endl;
cout <<sizeof (char_p) << endl;
cout <<sizeof (vec) << endl;
cout <<sizeof (vec_p) << endl;
```

Summary of *

- * is used for
  - Pointer declaration
  - Deference operator

They are not the same thing!

Example

We can change the value of an integer implicitly using pointers.

```
int num = 10;
cout << "before: " << num << endl;
int * p = &num;
*p = 20;
cout << "after: " << num << endl;
```

This prints:

```
10
20
```

Exercise

What does this print?

```
int num1 = 30, num2 = 20;
cout << "before: " << num1 << " " << num2 << endl;
int * p1, *p2;
p1 = &num1;
p2 = &num2;
*p1 = 10;
*p2 = *p1;
cout << "after: " << num1 << " " << num2 << endl;
```

Exercise

What does this print?

```
int num1 = 30, num2 = 20;
cout << "before: " << num1 << " " << num2 << endl;
int * p1, *p2;
p1 = &num1;
p2 = &num2;
*p1 = 10;
*p2 = *p1;
cout << "after: " << num1 << " " << num2 << endl;
```

before: 30 20

after: 10 10

Exercise

Note the difference between

```
int num1 = 30, num2 = 20;
cout << "before: " << num1 << " " << num2 << endl;
int * p1, *p2; // and int *p1, p2;
p1 = &num1;
p2 = &num2;
*p1 = 10;
*p2 = *p1;
cout << "after: " << num1 << " " << num2 << endl;
```

Null Pointers

- pointers are meant to point to valid addresses
- uninitialized pointers can point to unknown places
- Example

```
int *p;
cout << *p << endl; // you don't know what this will print
```

- Accessing this pointer causes undefined behavior
- If you want a pointer to point to nowhere, we use the null pointer (0 or nullptr)

```
int *p = nullptr; // recommend nullptr over 0
int *q = 0;
```

Null Pointers

- You can compare pointers with nullptr
- Example

```
int *p;
int *q = nullptr;
if (p == nullptr) {
    cout << "p is null" << endl;
}
if (q == nullptr) {
    cout << "q is null" << endl;
}
```

This prints:
q is null

Passing by Reference

- Caller passes a pointer instead of a value as an argument to a function
- The callee can modify the value of the object/value in the caller
- This another way to “return” multiple values

Example 1 - passing an int by reference

Previously, we had this which printed 1. We can now pass a pointer

```
void foo (int x) {
    x++;
}

int main()
{
    int a = 1;
    foo(a);
    cout << a << endl;
    return 0;
}
```

Exercise - swap

In bubble sort, we used the C++ built-in function swap to swap the location of two integers. We can now implement our own swap.

Exercise - swap

In bubble sort, we used the C++ built-in function swap to swap the location of two integers. We can now implement our own swap.

```
void my_swap (int *a, int *b) {

    int temp = *b;

    *b = *a;

    *a = temp;

}
```

Exercise

Complete the main function that calls my_swap to print the results below

```
int main() {
    int a = 1;
    int b = 2;
    cout << a << " " << b << endl; // 1 2
    // your code goes here
    cout << a << " " << b << endl; // 2 1
    return 0;
}
```

Example 2 – passing a vector by reference

Given the main function below, write two edit_vec functions that change the first element of v to 0 (one by value, one by reference)

```
int main() {

    vector <int> v = {1,2,3};

    edit_vec (v);

    cout << v[0] << endl; // 1

    vector <int> *p = &v;

    edit_vec (p);

    cout << v[0] << endl; // 0

}
```

Example 2 – passing a vector by reference

Given the main function below, write two edit_vec functions that change the first element of v to 0 (one by value, one by reference)

```
int main() {

    vector <int> v = {1,2,3};

    edit_vec (v);

    cout << v[0] << endl; // 1

    vector <int> *p = &v;

    edit_vec (p);

    cout << v[0] << endl; // 0

}
```

What other way can we print v[0]?

A. p[0]
B. *p[0]
C. (*p)[0]
D. v.back();

Exercise

What does the following print?

```
int main () {
    int a = 1;
    int b = 2;
    vector <int *> v = {&a, &b};
    vector <int *> *p = &v;
    cout << (*p)[0] << endl;
}
```

A. 1
B. Some memory address
C. {1, 2}
D. Error

Exercise

What does the following print?

```
int main () {
    int a = 1;
    int b = 2;
    vector <int *> v = {&a, &b};
    vector <int *> *p = &v;
    cout << *((*p)[0]) << endl;
}
```

A. 1
B. Some memory address
C. {1, 2}
D. Error

Example 3 – passing struct by reference

If we want to print its field using its pointer:

```
struct Posn {
    int x, y;
};
int main() {
    Posn p1 {1,1};
    Posn *p = &p1;
    cout << *p.x << endl; // ERROR, operator precedence problem
    cout << (*p).x << endl;

}
```

Example 3 – passing struct by reference

If we want to print its field using its pointer:

```
struct Posn {
    int x, y;
};
int main() {
    Posn p1 {1,1};
    Posn *p = &p1;
    cout << *p.x << endl; // ERROR, operator precedence problem
    cout << (*p).x << endl; // too many symbols…

}
```

Example 3 – passing struct by reference

If we want to print its field using its pointer:

```
struct Posn {
    int x, y;
};
int main() {
    Posn p1 {1,1};
    Posn *p = &p1;
    cout << *p.x << endl; // ERROR, operator precedence problem
    cout << (*p).x << endl;
    cout << p->x << endl;
}
```

Example 3 – passing struct by reference

Let's write a function move_to_origin, that changes the x and y of a Posn to (0, 0).

```
struct Posn {
    int x, y;
};
int main() {
    Posn p1 {1,1};
    Posn *p = &p1;
    cout << *p.x << endl; // ERROR, operator precedence problem
    cout << (*p).x << endl;
    cout << p->x << endl;
    move_to_origin(p); // or move_to_origin(&p1);
    cout << p1.x << " " << p1.y << endl; // 0 0
}
```

Exercise

Given the following structure and a vector of students, write the function update_grade (vector <Students> *s, int studentNum, int newGrade) that updates the student with studentNum to newGrade. You can assume that studentNum exists in the vector.

```
struct Student {
    int studentNum, grade;
};
```

Exercise

Sample main and output:

```
int main() {
    vector <Student> s = {{0, 99}, {1, 100}, {2, 98}};
    cout << s[0].grade << endl; // 99
    update_grade (&s, 0, 100); // update student 0’s grade
    cout << s[0].grade << endl; // 100
}
```

Example 4 – passing arrays

- Arrays are always passed by reference
- In fact the variable name of an array is a pointer that points to the first element of the array
- Example: what does this print?

```
void edit_arr (int a[]) {
    a[0] = 2;
}
int main() {
    int arr[10] = {1,2,3};
    cout << arr[0] << " ";
    edit_arr(arr);
    cout << arr[0] << endl;
    return 0;
}
```

A. 1 1
B. 1 2

Example 4 – passing arrays

- In the last example, edit_arr was:

```
void edit_arr (int a[]) {
  a[0] = 2;
}
```

we can also write:

```
void edit_arr (int *a) {
    a[0] = 2;
}
```

Or

```
void edit_arr (int *a) {
  *a = 2;
}
```

Passing Arrays to Functions

```
void edit_arr (int *a) {
   *a = 2; // because a points to the first element
}
```

- What use is int *a if it always points to the first element?
- We should be able to move a to the next element in the array

Pointer Arithmetic

- We can also do + and – operations on pointers
  - + moves it to the next element
  - - moves the pointer to the previous element

- Out of bounds? Adding numbers to an int pointer?

Example – Pointer Arithmetic

We used to print an array like this

```
int main() {
    int arr[10] = {1,2,3};
    int len = 3;
    for (int i = 0; i < len; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

   return 0;
}
```

Example – Pointer Arithmetic

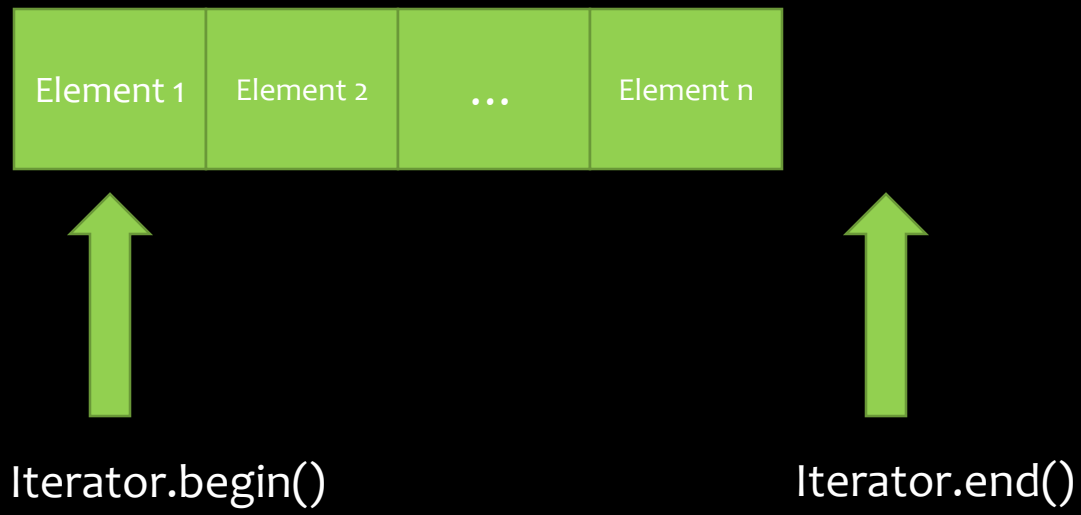
Using pointer arithmetic:

```
int main() {
    int arr[10] = {1,2,3};
    int len = 3;
    int *arr_p = arr;
    for (int i = 0; i < len; ++i) {
        cout << *arr_p << " ";
        ++arr_p; // move to next element
    }
    cout << endl;
}
```

The same applies for string, bool and other arrays.

Recall: Vector Iterators

- vec.begin() – iterator for beginning of vector (like a pointer)
- vec.end() – iterator for end of vector (like a pointer)
  - only use iterator for vec.erase(start, end) for now



vec.begin(), vec.end()

- vec.begin() and vec.end() just pointers
- We can dereference them to obtain the first element
- We can also do vec.begin() + 1 to get to the next element

Example: we can use these pointers to access elements of the vector

```
vector <int> v {1,2,3};
cout << *v.begin() << endl;  // 1
cout << *(v.end() - 1) << endl; // 3
```

vec.begin(), vec.end()

- Can we ruin these pointers?

```
vector <int> v {1,2,3};
v.begin() = v.end(); // reassigning the pointer
cout << *v.begin();
```

vec.begin(), vec.end()

- Can we ruin these pointers?

```
vector <int> v {1,2,3};
v.begin() = v.end(); // reassigning the pointer
cout << *v.begin(); // prints 1
```

- They are like const vectors
- **const** is a keyword that signifies that a variable cannot be modified
- "const" applies to the keyword before it,
  - If there are no keywords before it, it will apply to the keyword after it

const Keyword Examples

- const int a = 0; // a is a constant integer, const -> a
- int const a = 0; // a is a constant integer, const -> a

- const int *p = &a; // p points to a constant integer
  - *p is constant (you cannot modify a through p)
  - p itself is not constant
- int * const p = &a; // p is a constant pointer
  - *p can be modified
  - p cannot point to something else

Exercise

Are these valid assignments in the same program?

```
    const int a = 1;
1.   a = 2;
   int b = 1;
   const int *p = &b;
2.   *p = 2;
3. b = 2;
```

A. Yes
B. No

Exercise

Are the following independent code snippets valid?

4.
```
const int b = 1;
int *p = &b;
*p = 2;
```

5.
```
int b = 1;
int * const p = &b;
int d;
int *c = &d;
p = c;
```

A. Yes
B. No

Exercise

Are the following independent code snippets valid?

```
6. int b = 1;

   int * const p = &b;

   int *c = &b; // assigning to myself

   p = c;
```

A. Yes
B. No

Passing by Reference Applications

- Passing by value makes a copy of the value and passes the copy over
- Passing by reference passes a pointer to the current object/variable
- You should try to pass vectors and structures by reference because they are generally large and take a long time to run
- If you do not want other functions to modify them, you can add the keyword **const**

Dangling Pointers

- When a pointer points to something that does not exist
  - A variable that goes out of scope
  - Unpredictable behaviour
- Example:

```
int * foo () {
    int a = 1;
    return &a;
}

int main() {
    int * dangling_pointer = foo();
    cout << *dangling_pointer << endl; // unpredictable behaviour
}
```

Pointer Applications

CCC 2019 S1 Swap

CCC 2016 S3 Graph Theory