Arrays and Functions

Abhiram Ranade

Arrays and Functions

Array elements can be passed to functions

```
int z[200];  cin >> z[0] >> z[1];
z[2] = gcd(z[0], z[1]);
```

Entire arrays can also be passed to functions

- Need some new ideas.   Next.

Functions that operate on arrays: what we might want

Function to find the average of the numbers in an array?

```
double marks[100];  … read in marks …
double averageMarks = average(marks);
// or something like this.
```

Presumably averaging is a common operation, we should write an averaging function once for all.

Other desirable functions

- maximum(arrayName) : returns maximum?
- occurs(arrayName, value) : returns true if value occurs in array arrayName

All this is possible.

- Actual call is slightly different: length of array will also need to be an argument.
- More preparation is needed to understand how to write such functions.

Outline

- Memory and Addresses: Review
- Pointers
- Pointers and Functions
- An alternate, official view of arrays
- How to define functions that operate on arrays.

Memory and Addresses: Review

- Memory of a computer is made up of capacitors (typically)
- 8 bits make a byte.
- Each byte has an address.
- Addresses start at 0 and go up to memory size in bytes – 1.

Memory Allocation

- When variables are defined, memory is allocated for them.
- Amount of allocated memory = what is required for variables of that type.
- int requires 4 bytes. double requires 8 bytes. …
- Elements of an array allocated consecutively.
- Address of a variable: address of first byte allocated for it.

Example

int p = 3, q[]={11,12,13,14}, r=9;

This may cause:

Addresses	Allocated for	Content
1000-03	p	3
1004-07	q[0]	11
1008-11	q[1]	12
1012-15	q[2]	13
1016-19	q[3]	14

…

“Address of” operator: &

- Operator & applied to a variable gives its address

```
cout << &p <<‘ ‘<< &q[2] << endl;
```

- prints the starting addresses of p and q[2] resp 1000 and 1012. Convention: in Hexadecimal (radix 16)

$1000 = 3x16^2 + 14x16 + 8 = 3e8$

- Digits: 10, 11, 12, 13, 14, 15 = a, b, c, d, e, f

3e8 3f4 will be printed

- Often value printed will be prefixed by 0x to indicate that what follows is a hexadecimal number.

Properties of addresses

Addresses are numerical, but C++ treats them as a different type.

- It is an error to add one address to another (what could it possibly mean?)
- Addresses of int variables are of a type different from addresses of float variables.
- Addresses can be stored in variables.

Pointers: Variables for storing addresses

- Variables meant for storing addresses of variables of type T themselves have type "T*"

```
int* iptr;
```

- Memory allocated to create a variable iptr. iptr can hold address of int variables

```
int x; iptr = &x;
cout << iptr <<' '<< &x; // prints same value
```

- Pointer names: identifiers, usual rules.

“content of” operator: * (also “dereferencing” operator)

- *a : the variable whose address is a
- * is inverse of &

```
int x, y;   int* iptr;
iptr = &x;
*iptr = 4;  // 4 stored in x.
iptr = &y;
*iptr = 6;  // 6 stored in y.
```

Actual interpretation of definition int* ptr;

- Spaces don't matter with operators, and unary * associates to right. Hence should be read as

```
int (*iptr);
```

- Says *iptr is of type int. Thus iptr is of type address of int, or int*. indirect definition.

```
int* iptr, jptr;
```

- declares iptr to be int*, but jptr is int.
- What is expected in this blank?

```
int* iptr = …  // integer or address?
```

A rule regarding pointers

- Addresses of variables of type T can only be stored in variables of type T*.

```
int x;
double y;
int*p;
p = &y;  //not allowed, will not compile.
```

Addresses as function arguments

```
void add10(int* p){
   *p = *p + 10;
}
int main(){
  int x = 3;
  add10(&x);  // &x has type int*
  cout << x << endl;
}  // add10(x) would be an error.  Why?
```

Remarks

- When we apply * to a pointer p, we get the variable stored at address p.
- This happens even if the variable is defined in the some other activation frame.
- The type of *p is T if p is “pointer to T”
- Contract metaphor: tailor is not given cloth, but address of place from where to collect it.

Remarks

- We said earlier “function operates only on variables defined in its activation frame”. Not correct with pointer variables.
- We should have said: “function can refer only to variable names defined in its own activation frame.”
- Variables in other frames can be accessed indirectly, by dereferencing pointers, without knowing their name.

Exercise 1: What does this do?

```
void g(double *x, double *y){
  if(*x > *y){
    double z = *x;
    *x = *y;
    *y = z;
}
int main(){
  double x, y;   cin >> x >> y;    g(&x, &y);
  cout << x <<‘ ‘<< y << endl;
}
```

Exercise 2: What will this do?

```
int main(){
  double x, y, z;
  cin >> x >> y >> z;
  g(&x, &y);
  g(&y, &z);
  g(&x, &y);
  cout << x <<' '<< y <<' '<< z << endl;
}
```

Arrays: Our view so far

- Defining an array:

elemtype aname[asize];

Variables aname[0], … , aname[asize – 1] are created, each of type elemtype.

- aname : name of array created, name of the entire collection of variables.
- aname[i] : element with index i from aname.

Arrays: The official view

- aname : Officially, this is just the address of the zeroth created variable, i.e. &aname[0]
- address of zeroth element = address of first byte of memory allocated for the entire array.
- aname[i] : An expression with [] being the operator, and aname, i the operands.
- It will turn out to mean the right thing: NEXT

Official interpretation of a[b]

- a[b] : well defined only if a is of type T* for some type T, and b is of type int.
- Interpretation of a[b]: “The variable of type T stored at address a + S*b where S = size of one element of type T in bytes.”

int q[] = {11, 12, 13, 14}; // q has type int*

q[3] : The variable of type int stored in memory at address q + 4*3.

Example

```
int q[]={11,12,13,14};
```

Addresses	Allocated for	Content
1004-07	q[0]	11
1008-11	q[1]	12
1012-15	q[2]	13
1016-19	q[3]	14

q = 1004

q + 12 = 1016 : indeed where q[3] is stored.

Why bother if the official view gives the same result as our view?

- The computer uses the official view to determine where a[b] is in memory.
- Every time you write a[b], a multiplication and addition needs to be performed.
- Help you understand what happens if array index is out of range. NEXT
- Useful for writing functions on arrays. NEXT

What if index is out of range?

- q[10] : Variable of type int stored in memory at address q + 4x10 = 1044 i.e. at the position in memory where q[10] would have been if q had length at least 11.

- Although address 1044 was not allocated for array q, the computer will assume there is a variable there of the required type.

(contd.)

- q[10] = 100;

100 will be written into some other variable!

- x = q[10];

will cause junk data to be stored into x.

- a + S * b in general may correspond to non-existent address, or forbidden address. In such cases, program may halt.
- Summary: ensure index is in range!

What do you think this does?

```
int main(){
  int q[]={11,12,13,14};
  int *iptr = q;
  cout << iptr[0];
  iptr[1] = q[2];
  cout << q[1];
}
```

Value of iptr[0]

- Well defined? Yes. iptr is pointer to int, and 0 is int.
- What it means: Element of type int stored at iptr + 4*0, i.e. at iptr.
- But iptr = q. Hence iptr[0] means element of type int stored at q, i.e. q[0].
- Alternatively: iptr[0] must be same as q[0] since iptr is same as q.

Function calls on arrays

A design question

- Arguments to a function call are copied from the activation frame of calling function to corresponding parameters in activation frame of called function.
- Should arrays be copied? Arrays can be very long, and this may take too much time.

Key idea

- Make the starting address of the array be the argument, so only that will be copied.
- How to supply the starting address?

  Array name = starting address as per official view!
- Knowing the starting address and the index of the element we can calculate which location in memory is to be accessed.

Function to find average of elements

```
double average(int *A, int n){
  double sum = 0;
  for(int i=0; i<n; i++) sum = sum + A[i];
  return sum/n;
}
int main(){
  int q[] = {11, 12, 13, 14}; cout << average(q,4);
}
```

Will this compile correctly?

Types of arguments and corresponding parameter should match.

q: int*. A: int*

4: int. n: int.

will compile.

How this executes

main executes. call average encountered.

Activation frame created for average.

value of q copied to A.   value 4 copied to n.

In execution: A has same value as q. So A[i] will mean the same thing as q[i].

So sum will get the sum of elements of q.

So correct average will be returned.

Function to read into an array

```
void readarray(double *m, int n){
  for(int i=0; i<n; i++) cin >> m[i];
}
int main(){
  double marks[100];  readarray(marks,100);
}
```

Points to note

- If an ordinary variable is an argument, its value is copied. If the corresponding parameter changes, it does not affect the copied variable.
- If an array name is an argument, array name is copied. When corresponding parameter is used to access elements, elements of original array get accessed, and they can get modified.

Lookup function

```
int occurs(int k, int *key, int n){
  int res = - 1; // impossible index value;
  for(int i=0; i<n && res == -1; i++)
    if(key[i] == k) res = i;
  return res;
}
```

Use in marks program

```
int main(){
  double marks[100]; int rollno[100];
  for(int i=0; i<100; i++) cin >> rollno[i] >> marks[i];
  while(true){
    int r; cin >> r;
    int index = occurs(r, rollno,n);
    if(index != -1) cout << marks[index] << endl;
  }
}
```