CS 240 Programming in C

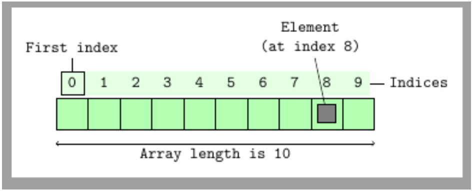
Array, Control Statements

September 16, 2024

Schedule

1. Array
2. Control Statement

Array



```
int a[10];
```

- To define an array without initialization, you have to have 3 parts, basic data type, name, and length within the square brackets.
- Once an array is defined, array indexing could be used to access the element in the array like a[9].
- Array index starts from 0 and can be any integer expression like a[1+2-1], or a[i+j*2-1].
- Out of range for indexing will not cause compiling error, we have to be careful.

Array Initialization

```
int a[] = {1,2,3};
// a has a length of 3

int b[10] = {1,2,3};
// b has a length of 10 and the rest elements
// will have a value of 0

int c[5] = {[1]=-1,[4]=-2};
// after c99; designated initializer
```

Array

- Array is a different data type compared to int, float, char, etc.
- Though an array is also a name to a fixed memory address, C defines its data type as a pointer data type, which means it represents an address value.
- And that address value are unchangeable.

Array

```c
int a,arr[3];

printf("%p\n", &a); // a's address
printf("%d\n", a);  // a's value
printf("%p\n", arr);  // arr's value
printf("%p\n", &arr); // arr's address
printf("%d\n", arr[0]);

// You will find out that the arr's value and its
// address values are the same.

// though &arr and arr have the same address value,
// they are pointing to di
```

Array

```
int a,arr[3];

printf("%p\n", arr);  // arr's value
printf("%p\n", &arr); // arr's address

printf("%p\n", arr + 1);
printf("%p\n", &arr + 1);


// though &arr and arr have the same address value,
// they are pointing to different data types:
// arr reference to int;
// &arr reference to int[3]
```

Sizeof() Array

We can also compute the sizeof of an array, but only within the scope where it is defined, not another function scope where it gets passed in.

```
    int a[3];
    sizeof(a);
```

not this. Since it is always the size of a pointer.

```
int array_pass(const int arr[]){
    printf("sizeof arr: %u\n", sizeof(arr));
}

int main(){
    int a,arr[3];
    array_pass(arr);
    // printf("sizeof arr: %u\n", sizeof(arr));
    return 0;
}
```

Array Copying

To copy array a to array b, like this

```
int a[] = {1,2,3}, b[3];
b = a;   // wrong, cannot even pass compiling
```

This is wrong, even though they have the same length and the same type:

```
Why?
```

Array Copying

- Arrays hold constant pointers. Once they are created, they point to a fixed address in the stack(which will be covered later.)
- So "b = a" tries to assign the address of a to b which can not change the address it points to is illegal.
- Even if it was legal, it will not be what the equation intends to achieve. Because b will point to the same block memory of a, instead of copying the data of a to the memory that b points to right now.

Array Copying

- Actually there are many ways to copy an array. Two are listed below.
- Write a loop and iterate every element.
- Use memcpy defined in <string.h>. Page 250.

memcpy

```
void *memcpy(s,ct,n)
//copy n characters from ct to s, and return s
```

- s and ct here are pointers since arrays are constant pointers, so we can copy the array an into array b using memcpy as follows:

  ```
  memcpy(b,a,sizeof(a));
  ```

- Lots of programmers prefer memcpy, because it is much fast than a loop, especially for large arrays, since there is no overhead for the computation of a loop, just copying.
- By the way, can we use memcpy for basic variables?
- Yes, but have to deal with their pointers.

memcpy

- By the way, can we use memcpy for basic variables ?
- Yes, but have to deal with their pointers

```
int a=0,b=1;

memcpy(b,a,sizeof(a)); // wrong

memcpy(&b,&a,sizeof(a)); right
```

- But nobody does this way. Just b=a.

Character Arrays (Strings)

Recall String constant (or literal)

- A sequence of 0 or more characters surrounded by double quotes
- Ended with a null character '\0'
- Quotes are not part of the string – they serve only to delimit

Character Arrays (mutable)

- Stored as an array of characters
- Aside from regular array initialization, we can initialize a character array like this:

```
char str[7] = "hello\n"; // at least length of 7
char str2[] = "12345\n"; // or implicit initialization
printf("%d\n", sizeof(str2));
```

- What will be print out and why?

Immutable string

Immutable string

- Character array is mutable but, constant string literal is not.
- Constant string returns a pointer.

```
char s[] = "hell0";
s[1] = '1';
char *s2 = "h2ll0";
s2[1] = 2; // this is wrong;
```

Statements: if, else if, else

```
if (condition 1)
 { branch 1 }
else if (condition 2)
 { branch 2}
[...]
else
 { branch n}
```

The curly braces are optional when the branch only contains 1 statement.

Statements and Blocks

- Recall that an expression is a combination of values, constants, variables, operators, and functions that evaluate another value
- An expression becomes a statement when it is followed by a semicolon For example, `x = 0;`
- Curly braces are used to group statements into a compound statement, or block

  ```
  {
    x = 0;
    y = 1;
  }
  ```

  which acts like one statement to the outside.
- Note that the closing brace is not followed by a semicolon

Statements and Blocks

- Syntactically, the grouped statements are equivalent to a single statement
- In control statements, there are often curly braces being used for grouping statements to one of those branches
- These are other use cases for blocks when we later talk about scopes.

If Statements

- Things to note:
- The `if` condition is just testing a numeric value in which 0 means false and non-zero means true.
- There is not a data type of boolean in C, instead it's 0 and non-zero
- We can use a shortcut in this test:
  - `if (expression)` is the same as `if (expression != 0)`
  - `if (!(expression))` is the same as `if (expression == 0)`

Else-If

- Things to note:
- Can have as many as you want
- They are evaluated in order
- If the condition evaluates to be true for one, its statement is executed, and we don't look at the rest
- An `else` at the end is equivalent to "none of the above"

Relational Operators

- Check the relationship between the values of their operands
- The expression always evaluates to 1 (true) or 0 (false)
- `x == y`: the values of x and y are equal
- `x != y`: the values of x and y are not equal
- `x > y`: x is greater than y
- `x < y`: x is less than y
- `x >= y`: x is greater than or equal to y
- `x <= y`: x is less than or equal to y

Assignment Versus Equality Operators

- = assignment operator (not a statement)
- == equality operator
- `(c == 9)` tests whether c is the newline character
- `(c = 9)` this has the value of 9
- It is better for you to write `(9 == c)` instead of `(c == 9)` since if you forgot the double equal sign, the first will throw out an error, however, the later will always assume the value of 9, which can be a big trouble.
- Let's see a demo.

Switch Statements

- Another way to do multi-way decisions

```
switch (expression) {
    case constant-expr1:
        statements
    case constant-expr2:
        statements
    default:
        statements
}
```

Switch Statements

- This will test whether the expression matches each of the constant expressions and execute the corresponding statements
- if there is no case being matched, the default statements will be executed.
- The constant expressions must be integer-valued
- Execution will fall through a switch (which means goes to the next switch statement) unless you add `break` after statements.
- That is to say an end of one case statement is not an end of one switch unless it is the last case statement.
- Compare to if-else, this is a big difference. Since in if-else one branch ends, the next if-expression will be evaluated, however, switched case will just fall through.
- Let's see a demo.

If-Else vs. Switch

- Suppose we need to test the value of a status variable, and there are 20 different values
- With `if-else`, we test `(status == 1)`, then `(status == 2)`, etc.
- By the time we reach 20, we have tested 19 times
- With `switch`, it is usually compiled into assembly as a *jump table*
- An array of `goto` instructions subscripted by the value of `status`
- If status is 20, we look up the `goto` at address 20 in the table
- This way we only execute that one `goto`
- Good practice is to always use `break`
- Falling through can be useful, but you should be careful with it as it may create unintended behavior if the program is modified later

Loops

- These are equivalent

```
for (expr1; expr2; expr3)
  statements;
```

```
expr1;
while (expr2) {
    statements;
    expr3;
}
```

- Note that any part of a for loop can be left out
  `for(init; loop-test; steps)`
- If `init` is left out, you must initialize it somehow
- If `steps` is left out, you must manage steps
- If `loop-test` is left out, you must break in some case

Comma Operator

- Most often used is in the for-loop statements
- Pairs of expressions separated by a comma are evaluated left-to-right
- Value of comma expression is the value of the rightmost comma-separated expression

Comma Operator

- Example of using the comma operator in a for-loop:

```
int a[] = {1,2,3}, i,j,temp;
for(i = 0, j = sizeof(a)/sizeof(int)-1;
                    i < j; i++, j--){

       temp = a[i];
       a[i] = a[j];
       a[j] = temp;
 }
```

Do While

```
do {
  statements;
} while (expression);
```

- Guaranteed to execute the statements at least once, regardless of whether the expression is true or false
- Used infrequently

Break and Continue

- `break`
  - Allows departure from a loop
  - Can be used in for, while, and do loops (similar to its use in switch)
  - Allows you to exit the current loop
    one level only; remember this is when you use to break in nested loops
- `continue`
  - Skips to the next iteration of the loop
  - It is used to selectively execute statements in a loop iteration