Introduction to C

Tutorial 9: Pointers
Last week…

• Functions
• Call stack
• Call by value
• Scope of variables
Agenda

• Pointers
• Call by reference
• Example
Pointers
& operator

We learned that every variable has an address in memory:

- Remember that memory allocation for a variable is taken care of by the operating system. Therefore, the memory address for variable \( a \) changes from execution to execution.

- \& operator gives us the address of the variable.
  - (The address of the first bit it takes up in memory)

Constants and expressions don’t have addresses.
Pointers

• A pointer is a variable that can store an address
• To define a pointer variable, add a * to the definition:
  ```c
  int *p;
  ```
• Pointers also indicate the type of the variable so that the right amount of memory can be allocated.
• Pointers allow you to access other variables in a roundabout way.
Initializing a pointer

• Like all other variables, you can (and it’s highly recommended to) initialize pointers.
  – You can initialize a variable with an address of another
    \[
    \begin{align*}
    \text{int } a &= 78; \\
    \text{int } *p &= &a; \\
    \end{align*}
    \]
    \[
    \begin{align*}
    \text{int } a &= 78; \\
    \text{int } *p; \\
    p &= &a; \\
    \end{align*}
    \]
  – Or you can initialize it to address 0 or NULL.
    \[
    \begin{align*}
    \text{int } *p &= 0; \\
    \end{align*}
    \]
    \[
    \begin{align*}
    \text{int } *p &= \text{NULL}; \\
    \end{align*}
    \]
  – Address 0 is an illegal address, so we will initialize our pointer this way to establish that the pointer has does not have a value yet.

It’s forbidden to initialize a pointer with a numbered address
The * operator returns the value being pointed to by the pointer.

- Don’t get confused with the use of * from pointer definitions: `int *p;`

It’s like jumping from the memory block of the pointer, to the memory location of the variable.

The expression `*p` is equivalent to `a` for all purposes!

```c
int a = 78;
int *p;
p = &a;
*p = 10;
```
What can you do with a pointer?

What will the program print?

```c
int a = 78;
int *p = &a;
*p = 19;
printf("a = %d\n", a);
printf("*p = %d\n", *p);
a = 100;
printf("a = %d\n", a);
printf("*p = %d\n", *p);
```

• Notice the difference between the operators:
• "address of" (&)
  "the value pointed to by" (*)
Variable type and its initialization

• Which initializations are correct? Which aren’t? Why?

```c
float *p = q, q;
float *q, *p = q;
float *q, *p = &q;
float q, *p = &q;
float *q, p = &q;
int x = 12, y = 2, *z = 14;
int x = 12, y = 2, *z = &x;
int x = 12, y = 2, *z = &(y+x);
int *z = &14;
int *z = 0;
```
Riddle 1

1. The value of a is
   A. &a  B. 5  C. 19  D. &b
2. The value of b is
   A. &a  B. 5  C. 19  D. &b
3. The value of pa is
   A. &a  B. 5  C. 19  D. &b
4. The value of *pb is
   A. &a  B. 5  C. 19  D. &b

```c
int a = 5;
int b = a;
int *pa = &a;
int *pb = &b;
*pa = 19;
pb = pa;
```
5. The value of \(a\) is
   A. 7  B. 5  C. 19  D. \&b
6. The value of \(b\) is
   A. 7  B. 5  C. 19  D. \&b
7. The value of \(*pb\) is
   A. 7  B. 5  C. 19  D. \&b
8. The value of \(*pa\) is
   A. 7  B. 5  C. 19  D. \&b

```c
int a = 5;
int b = a;
int *pa = &a;
int *pb = &b;

*pa = 19;
pb = pa;
b = 7;
*pb = 5;
```
Changing pointer value by changing the value being pointed to

• What is the difference between the two code snippets?

```c
int a = 78, b = 100;
int *p1 = &a, *p2 = &b;
p1 = p2;
*p1 = -7;
```

```c
int a = 78, b = 100;
int *p1 = &a, *p2 = &b;
*p1 = *p2;
*p1 = -7;
```

• Which variables change value when this code runs?
Passing parameters by address
(Call by reference)
In which example(s) do we need pointers? Why?

1. A function that computes the greatest common divisor between two numbers.
2. A function “swap” that switches the values between its two parameters.
3. A function that computes the sum of two complex numbers.
4. A function that receives a non negative number as input and then returns it.
Passing an address as a parameter

• By passing an address as a parameter, you can update – from within a function – a variable that is outside the function.

• Is the function changing its parameter?
  – Technically, no: The parameter that is passed in is an address of something, and that address does not change.
  – Conceptually, yes: The parameter that was passed is &a and the value of a is changed.

• Passing the address of a variable to a function is called calling by reference
Passing an address as a parameter: Example

Inside the function, change the actual parameter using the * operator.

Declare the formal parameter as pointer instead of a regular type.

Call the function with the address of the variable.

```c
void set_to_pi(float *p) {
    *p = 3.1415;
}

int main(void) {
    float a = 0;
    set_to_pi(&a);
    printf("%f", a);
    return 0;
}
```
Example of a run

```c
void swap(int *x, int *y)
{
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void)
{
    int a = 3, b = 7;
    swap(&a, &b);
    printf("a=%d , b=%d", a, b);
    return 0;
}
```

This is function swap, which will accompany us for the rest of the course.
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int a = 3, b = 7;
    swap(&a, &b);
    printf("a=%d, b=%d", a, b);
    return 0;
}
What happens at runtime?

```c
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int a = 3, b = 7;
    swap(&a, &b);
    printf("a=%d , b=%d", a, b);
    return 0;
}
```

Allocate memory for variables `a` and `b`.
What happens at runtime?

```c
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int a = 3, b = 7;
    swap(&a, &b);
    printf("a=%d , b=%d", a, b);
    return 0;
}
```

Pass by reference

Calling function `swap`
What happens at runtime?

```c
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int a = 3, b = 7;
    swap(&a, &b);
    printf("a=%d, b=%d", a, b);
    return 0;
}
```
What happens at runtime?

```c
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int a = 3, b = 7;
    swap(&a, &b);
    printf("a=%d, b=%d", a, b);
    return 0;
}
```
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int a = 3, b = 7;
    swap(&a, &b);
    printf("a=%d , b=%d", a, b);
    return 0;
}
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int a = 3, b = 7;
    swap(&a, &b);
    printf("a=%d , b=%d", a, b);
    return 0;
}

What happens at runtime?

The function finishes, the memory is freed.
What happens at runtime?

```c
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int a = 3, b = 7;
    swap(&a, &b);
    printf("a=%d , b=%d", a, b);
    return 0;
}
```

Back to the place from which we called swap
What happens at runtime?

```c
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int a = 3, b = 7;
    swap(&a, &b);
    printf("a=%d , b=%d", a, b);
    return 0;
}
```

The output: this is what we wanted!
What happens at runtime?

```c
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int a = 3, b = 7;
    swap(&a, &b);
    printf("a=%d , b=%d", a, b);
    return 0;
}
```

The program finishes. The memory that was allocated is freed.
Riddle 3

What will the values of the variables be after running mysteryFunction?

1. The value of $a$ is
   A. 7  B. 3  C. 10  D. Different each run
2. The value of $b$ is
   A. 7  B. 3  C. 10  D. Different each run
3. The value of $c$ is
   A. 7  B. 3  C. 10  D. Different each run
4. The value of $d$ is
   A. 7  B. 3  C. 10  D. Different each run
5. The value of $d^*$ is
   A. 7  B. 3  C. 10  D. Different each run

```c
void mysteryFunction(int *x,
                     int *y,
                     int z,
                     int *w)
{
    int *tmp = x;
    w = y;
    if (*tmp > *y) {
        z = *y;
    }
    *w = *x;
    *tmp = z;
}

int main(void)
{
    int a = 7, b = 3, c = 10, *d;
    mysteryFunction(&a, &b, c, d);
    return 0;
}
```
Common mistake: defining variables in main as pointers instead of regular variables
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int *a, *b;
    *a = 3; *b = 7;
    swap(a, b);
    printf("a=%d, b=%d", a, b);
    return 0;
}
What is the problem?

- The pointers **a** and **b** are defined in main, but the pointers are not initialized.

- That is to say, the pointers might be pointing to illegal addresses or memory locations that are already taken. This can lead to unexpected results.

- The best case scenario is that there is a runtime error so that we can go back and fix the problem.

- In the worst case scenario, the program will work and we won’t notice the error. This will leave the program open to errors in the future.
Example: Date arithmetic
Problem definition

• In a biology experiment, we need to how long after the beginning of the experiment to end it.
• The duration of the experiment will be measured in days.
• Try calculating it quickly: If the experiment started April 1st and should last for 140 days, when should it end?

• The calculation is not easy because each month has a different number of days.
  — Counting pages on a calendar?
Problem definition

• We’ll write a function with the following inputs:
  – Day
  – Month
  – Year
  – Number of days

• Required output
  – Day
  – Month
  – Year

• Do we need 7 parameters? (4 input, 3 output)?

• We can make due with 4 parameters

• 3 parameters (day, month, year) will be marked for input and for output.

• The program will use their values as a starting point for the calculation, and use them to return the result
Solution idea

• Remember the number of days we need to advance the date.
• We’ll compute the number of days left until the end of the experiment.
• If there are fewer days left than there are remaining in the month:
  – We’ll advance the date appropriately within the same month.
• Otherwise, we advance to the first day in the next month.
  – Reduce the number of days left in the experiment by the number of days that were in the month.
  – If we complete the month of December, then increment the year.
Helper function

- In order to implement the function, we’ll use a helper function.
- Input: Month and year
- Output: Number of days in the month

- Why do we need to provide the year for this function?
int daysInMonth(int month, int year)
{
    switch(month) {
        case 4:
        case 6:
        case 9:
        case 11:
            return 30;
        case 2:
            if (((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0))
                return 29; /* Leap year */
            return 28;
        default: /* (1, 3, 5, 7, 8, 10, 12) */
            return 31;
    }
}
And now, the work...

```c
void findFutureDate(int *day, int *month, int *year, int numDays)
{
    while (numDays > 0) {
        int leftThisMonth = daysInMonth(*month, *year) - *day + 1;
        if (numDays < leftThisMonth) { /* Result date is within the current month. */
            *day += numDays;
            numDays = 0;
        }
        else {
            *day = 1; /* Advance date to beginning of next month. */
            (*month)++;
            if (*month == 13) {
                *month = 1; /* Happy new year! */
                (*year)++;
            }
            numDays -= leftThisMonth;
        }
    }
}
```

Wait – if we are updating the 4th parameter – could it be argued that all four parameters are part of the input and output?

The parameter numDays gets updated continuously, and represents the number of remaining days to advance the date.
Example: Using the function

```c
int main(void) {
    int day = 1;
    int month = 4;
    int year = 2006;
    findFutureDate(&day, &month, &year, 140);
    printf("Experiment should end on %d/%d/%d\n", day, month, year);
    return 0;
}
```

- The first three parameters are passed using variables, the final one is passed with as a constant.
- Is it possible to pass the first 3 parameters as constants?
- Is it possible to pass the last constant as a variable?
- What is the type of parameter passing that is happening in each case?
Example: solution for a quadratic equation
Defining the problem

- We would like to write a function that solves a quadratic equation.
- The problem: A quadratic equation can have two solutions.
- But a function can only return one value!
Solution using pointers

• As previously stated, a function can only return one value.

• But we can also make a function update its parameters, using a call-by-reference
  – That is to say, we pass the function the address of the variables that it will update.

• In our case, we can pass the addresses of two variables to our function. After the function is performed, we will ensure that variables contain the results.
The function receives 5 parameters – 3 for input, 2 for output.

The function itself does not have a return value, so it is void.

```c
void solveSqEq(double *x1, double *x2, double a, double b, double c)
{
    double sqrt_delta = sqrt(b*b - 4*a*c);
    *x1 = (-b + sqrt_delta) / (2*a);
    *x2 = (-b - sqrt_delta) / (2*a);
}
```
And what if there isn’t a solution?

- Sometimes a quadratic equation does not have a solution.
- We can write our function so that it detects these cases and reports those situations accordingly.
- This can be an additional output – the number of solutions (0, 1, or 2).
- Does the function need a 6th parameter in order to do have this functionality?
int solveSqEeq(double *x1, double *x2,
               double a, double b, double c)
{
    double delta = b*b - 4*a*c, sqrt_delta;
    if (delta < 0.0) {
        return 0; /* No solutions */
    }
    if (delta == 0.0) {
        *x1 = -b / (2*a);
        *x2 = *x1;
        return 1; /* One solution */
    }
    sqrt_delta = sqrt(delta);
    *x1 = (-b + sqrt_delta) / (2*a);
    *x2 = (-b - sqrt_delta) / (2*a);
    return 2; /* Two solutions */
}
# Riddle solutions

<table>
<thead>
<tr>
<th>Riddle 3</th>
<th>Riddle 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>– 1 -B</td>
<td>– 1 -C</td>
</tr>
<tr>
<td>– 2 -A</td>
<td>– 2 -B</td>
</tr>
<tr>
<td>– 3 -C</td>
<td>– 3 -A</td>
</tr>
<tr>
<td>– 4 -D</td>
<td>– 4 -C</td>
</tr>
<tr>
<td>– 5 -D</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Riddle 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>– 5 -B</td>
</tr>
<tr>
<td>– 6 -A</td>
</tr>
<tr>
<td>– 7 -A</td>
</tr>
<tr>
<td>– 8 -B</td>
</tr>
</tbody>
</table>