ECE15: Introduction to Computer Programming Using the C Language

Lecture Unit 8: Pointers
Why Do We Need Pointers?

Pointers
- The address `&` and indirection `*` operators
- Pointers and functions: call by reference
- The `NULL` constant and the `void *` type

Pointer Arithmetic in C

Arrays and Pointers
Recall that in C, arguments are **passed to functions by value**. When a function $f(int \ y)$ is called as $f(x)$, the current value of the variable $x$ is sampled and *copied into* $y$. The function has no access to $x$.

**Example:** We often need to swap the values of two variables $x \leftrightarrow y$. Write a function for this.

```c
void swap(int x, int y)
{
   int temp = x;
   x = y;
   y = temp;
}
int main(void)
{
   int x = 3, y = 4;
   swap(x,y);
   ...
}
```
### Problems with Arrays

**Average of a large array:**

```c
double average(int b[], int N)
{
    int i, sum = 0;
    for (i = 0; i < N; i++)
        sum += b[i];
    return sum / (double) N;
}
```

**Will the program copy the 1,000,000 integers from a[] to b[]?**

**Array of unknown size?**

```c
#include <stdio.h>
int main(void)
{   int i, N;
   scanf("%d",&N);
   int a[N] = {0};  
   for (i = 0; i < N; i++)
   scanf("%d",&a[i]);  
   ...
}
```

- Will the program copy the 1,000,000 integers from a[] to b[]?

- What to do?

### What to do?

```c
#include <stdio.h>
int main(void)
{   int i, N;
   scanf("%d",&N);
   int a[N] = {0};  
   for (i = 0; i < N; i++)
   scanf("%d",&a[i]);  
   ...
}
```
Functions Returning Two Values?

**Example:** Write a function that computes the roots of a general quadratic equation \(ax^2 + bx + c = 0\). Print these roots.

```c
#include <stdio.h>

double quad_solve(int a, int b, int c);

int main(void)
{
    double a = 1.0, b = 4.0, c = 1.0; double x[2];
    x = quad_solve(a, b, c);
    printf("The roots are %d and %d\n", x[0], x[1]);
    return 0;
}

double quad_solve(int a, int b, int c)
{
    double root[2];
    ...
    return root[1], root[2];
}
```
We have observed several fundamental problems in C:

- A function cannot change the variables passed to it by value.
- Passing arrays to a function by value wastes a lot of memory and time.
- Cannot handle arrays whose size is unknown at compilation time.
- Functions cannot return to the calling environment more than one value.

The same idea solves all these problems:

Instead of using the variables themselves, use their addresses!
Why Do We Need Pointers?

Pointers

- The address & and indirection * operators
- Pointers and functions: call by reference
- The NULL constant and the void * type

Pointer Arithmetic in C

Arrays and Pointers
Refresher on Memory Addressing

Every variable has an address in memory, which can be used to access it!

Memory is organized as a long sequence of **bytes**. Every variable is allocated a prescribed number of bytes in memory, according to its **type**. Local variables are allocated **consecutively**, in the variable stack.

**Example:**

```c
{ double x;
  short d;
  char c;
  ...
}
```

Every variable has an address in memory, which can be used to access it!
What Is a Pointer?

**Address Operator** 
\&: If \( a \) is a variable of any type, the expression \&\( a \) evaluates to the address of \( a \) in memory.

- In the foregoing example: \&\( x \) is 1000, \&\( d \) is 1008, and \&\( c \) is 1010.
- The address operator works only for variables: \&7 is a compilation error.

For every valid type in C (such as int, double, char, boolean) there is another type: pointer to type, denoted type* (such as int*, double*, char*, boolean* and so on).

**Pointer variables:** A pointer \( p \) is a variable of type pointer to another type, declared by type* \( p \). Pointer variables hold addresses of other variables (or even non-variables, such as files or functions).

- \( \text{int* } p; \)
- \( \text{int* } p; \)
- \( \text{char* } ptr; \)
- \( \text{FILE* } in; \)
- \( \text{int** } p; \)
- \( \text{int* } *p; \)

Examples:
Pointer Declarations: Examples

Example:
```
char c;
char *p1, *p2 = &c;
double *p;
double x;
p = &x;
```

Example:
```
int a = 3;
int *p,*q;
p = &a;
q = p;
```

Example:
```
int *p;
double a;
p = &a;
```

Different pointer types cannot be mixed!

But they can be cast.

initialization

assignment
If \( p \) is a pointer that contains the address of a variable \( a \), then \( *p \) is that variable \( a \). This operation is called \textit{indirection} or \textit{dereferencing}.

Addressing \& and indirection * are \textbf{dual operations}: &a fetches the address of a variable \( a \) and stores it in a pointer \( p \), while \( *p \) takes the address stored in \( p \) and fetches the variable \( a \) at that address.
The declarations `int *p;` and `double *p;` can be now understood as follows: the variable pointed to by `p` is of type `int` or of type `double`.

Several levels of indirection are possible: `**p ↔ *(p)`), `***p`, and `****p`, and so on, are all potentially valid expressions.

Pointers that have not yet been assigned an address point to an arbitrary place in memory. Thus `x = *p` will put garbage in `x`, while `*p = x` could crash the computer.

Example:

```
int a = 30;
int y;
int *p;
p = &a;
y = *p;
*p = 42;
```
### Expressions Involving & And *

**Example.** Consider the following declarations:

\[
\text{int } i = 3, \ j = 5, \ \*p = \&i, \ \*q = \&j, \ \*r; \\
\text{double } x;
\]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( **p )</td>
<td>((&amp;p))</td>
<td>3</td>
</tr>
<tr>
<td>( &amp;&amp;p )</td>
<td>&amp;(*p)</td>
<td>1</td>
</tr>
<tr>
<td>( * (r = &amp;j) *= * p )</td>
<td>((r = &amp;j) *= (*p))</td>
<td>15</td>
</tr>
<tr>
<td>( *(p) + 7 )</td>
<td>(7*(*p)/(*q) + 7)</td>
<td>11</td>
</tr>
<tr>
<td>( *p == &amp;*p )</td>
<td>(p == &amp;(*p))</td>
<td>1</td>
</tr>
</tbody>
</table>

*\&p is simply \(p\) unary and binary operators
Why Do We Need Pointers?

Pointers

- The address & and indirection * operators
- Pointers and functions: call by reference
- The NULL constant and the void * type

Pointer Arithmetic in C

Arrays and Pointers
Example: A function that swaps the values of two variables $x \leftrightarrow y$.

```c
void swap(int* px, int* py)
{
    int temp = *px;
    *px = *py;
    *py = temp;
}

int main(void)
{
    int x = 5, y = 7;
    swap(&x, &y);
    ...
```
Pointers as Function Parameters

**Note:** The `scanf()` standard library function updates the content of a memory location, hence it is always passed its **address**!

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

int main(void)
{
    int a, b;
    int c, d;
    int *p = &c;
    scanf("%d", &a);
    scanf("%d", p);
    ...
    swap(&a, &b);
    swap(p, &d);
    ...
}
```
Example: Simplifying Fractions

**Task:** Write a function that, given variables \(a\) and \(b\) that represent the numerator and the denominator of a fraction \(a/b\), reduces the fraction to the lowest terms --- that is, cancels all the common factors of \(a\) and \(b\).

**Example:**
\[
\frac{a}{b} = \frac{75}{100} \quad \Rightarrow \quad \frac{a}{b} = \frac{3}{4}
\]

**Function prototype:**

\[
\text{void reduce(int *x, int *y);}\]

**Algorithm:** Call \(\text{gcd}(a,b)\) to compute the greatest common factor of \(a\) and \(b\), then divide both \(a\) and \(b\) by this common factor.

**Declaration**

\[
\text{int a,b;}\]
\[
\text{...}\]
\[
\text{reduce(&a,&b);}\]

**Call**
int gcd(int, int);

void reduce(int *numerator, int *denominator)
{
    int d = gcd(*numerator,*denominator);
    if (d > 1)
    {
        *numerator /= d;
        *denominator /= d;
    }
}

int gcd(int m, int n)
{
    while(n != 0)
    {
        int tmp;
        tmp = n;
        n = m % n;
        m = tmp;
    }
    return m;
}
Task: Write a function that, given two fractions \( \frac{a_1}{b_1} \) and \( \frac{a_2}{b_2} \), represented by their numerators and denominators, computes their sum \( \frac{a}{b} \).

Example:

\[
\frac{a_1}{b_1} = \frac{5}{12} \quad \frac{a_2}{b_2} = \frac{2}{15}
\]

\[
\frac{a}{b} = \frac{5 \cdot 5 + 2 \cdot 4}{60} = \frac{33}{60} = \frac{11}{20}
\]

Function prototype:

```c
void add(int *a, int *b, int a1, int b1, int a2, int b2);
```

Declaration

```c
int a, b, a1, b1;
...
add(&a, &b, a1, b1, 2, 15);
```

Algorithm: Compute \( b \) as the least common multiple of \( b_1 \) and \( b_2 \). Then \( a = a_1 \left( \frac{b}{b_1} \right) + a_2 \left( \frac{b}{b_2} \right) \). Then reduce the resulting fraction.
int gcd(int, int);
void reduce(int *x, int *y);
int lcm(int, int); // least common multiple

void add(int *a, int *b, int a1, int b1, int a2, int b2)
{
    *b = lcm(b1,b2);
    *a = a1*(b1/b) + a2*(b2/b);
    reduce(a,b);
}

int lcm(int m, int n)
{
    return m/gcd(m,n)*n;
}
Why Do We Need Pointers?

Pointers

- The address & and indirection * operators
- Pointers and functions: call by reference
- The NULL constant and the void * type

Pointer Arithmetic in C

Arrays and Pointers
Pointers to void

What's a pointer to void?

Example:

```c
int a, b, *p1;
void *p2 = &b;
p1 = &a;
b = *p1 + *(int*)p2;
```

Lecture Unit 8
ECE15: Introduction to Computer Programming Using the C Language
Pointers to Type `void`

What if you need a pointer whose type becomes known only at run-time?

Use a pointer of type `*void`, then cast its type at run-time!

- A pointer of type `*void` is generic. It points to some address in memory, but not to a variable of specific type (such as `int`, `char`, `double`, etc.)
- A pointer to `void` can be cast at run-time to become a pointer to any type!
- For an uncast pointer of type `*void`, you cannot do indirection (we do not know how many bytes to fetch from `*p`) and pointer arithmetic. However, you can do everything else, including assigning an address to `p`.

Example:

```c
int a=5, b, *p1;
char c = 'x';
void *p2, *p3 = &c;

p1 = &a;
p2 = p1;
b = *p1 + *(int*)p2;
```

- works without a cast
- cast is needed!
Memory addresses are not of type int. Nevertheless, every pointer variable is allowed to take one special integer value: constant 0.

For clarity, when dealing with pointers, it is common practice to use NULL instead of 0. The constant NULL is defined in <stdio.h> as 0.

The C language and the operating system guarantee that 0 is never a valid address. Thus if p is 0, then *p cannot overwrite anything.

Giving a pointer the value NULL is a safe way to indicate that this pointer is invalid or has not yet been assigned to point anywhere.

```c
int *p = NULL;
*p = 42; /* What will happen? */
```
Why Do We Need Pointers?

Pointers
- The address & and indirection * operators
- Pointers and functions: call by reference
- The NULL constant and the void * type

Pointer Arithmetic in C

Arrays and Pointers
Every pointer variable holds an address of some byte in memory, which is just an integer. However, pointers are not integers!

Pointer variables cannot be multiplied, divided, or added!

Nevertheless, the C language does allow a limited number of arithmetic operations on pointer variables of the same type:

**Basic principle:** Suppose that \( p \) is a pointer to a given type (such as `int`, `double`, `char`, `boolean`, etcetera). If `sizeof(type)` is \( n \), then \( p++ \) advances \( p \) by \( n \) bytes, whereas \( p-- \) moves \( p \) back by \( n \) bytes.
Pointer Arithmetic

Example:

```c
typedef char my_type;
my_type a[] = {'p','o','i','n','t','e','r'};
my_type *p;
p = &a[3];
printf("%c%c%c!!\n", *(p+1), *(p-2), *(p-3));
```

Which arithmetic operations on pointers are OK?

- Adding or subtracting an integer, or any expression that evaluates to type `int`, for example: `p-42`, `p += 2`, `q = p-2*(++a)`.
- Subtracting two pointers of the same type. This evaluates to the distance between the corresponding addresses in memory.

These operations are very useful when dealing with arrays!
**Example.** Suppose that a variable of type `int` occupies 4 bytes, and consider the following declarations:

```c
int *p, *q;
p = (int*)300;
q = (int*)400;
```

What are the values of the following pointer-arithmetic expressions?

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p + 1</code></td>
<td>304</td>
</tr>
<tr>
<td><code>(char*)p + 2</code></td>
<td>302</td>
</tr>
<tr>
<td><code>p++</code></td>
<td>300</td>
</tr>
<tr>
<td><code>q - p</code></td>
<td>25</td>
</tr>
<tr>
<td><code>(char*)q - (char*)p</code></td>
<td>100</td>
</tr>
<tr>
<td><code>q - (char*)p</code></td>
<td>Error</td>
</tr>
<tr>
<td><code>printf(&quot;%p&quot;, p)</code></td>
<td>0x12c</td>
</tr>
</tbody>
</table>

But `p` is now 304 not an expression
Why Do We Need Pointers?

Pointers

- The address & and indirection * operators
- Pointers and functions: call by reference
- The NULL constant and the void * type

Pointer Arithmetic in C

Arrays and Pointers
The most useful feature of an array is that all its elements can be subscripted, as follows:

```c
double salaries[800];
for (sum = 0, i = 0; i < 800; i++)
    sum += salaries[i];
```

The array elements salaries[0], salaries[1], ..., salaries[799] are all variables of type double. But...

**What is salaries itself?**

**Answer:** salaries is the address of (first element of) the array!

Subscripting in C is implemented as follows:

```
&salaries[0]  salaries
&salaries[1]  salaries+1
&salaries[2]  salaries+2
...          ...
```

```
salaries[i]  *(salaries+i)
```

**Pointers!**
Any expression of the form `xx[yy]` is **first** converted to `*(xx + yy)`, **then** evaluated. This works iff `xx + yy` is a valid pointer.

**Example:**

```c
char my_array[] = "pointer";
char *p; int i;
p = my_array;
for (i = 0; i < 7; i++) printf("%c", p[i]);
```

**Example:**

```c
int my_array[] = {0,1,2,3,4,5,6};
int *p; int i;
p = &my_array[3];
for (i = -3; i < 4; i++) printf("%d", p[i]);
```

**Example:**

```c
char a[] = "pointer";
printf("%c%c%c = %d", 4[a],1[a],0[a],7[a]);
```
The subscription rules in C imply that arrays are equivalent to pointers in many situations. Are arrays and pointers the same thing?

Arrays are much more than pointers!

double salaries[800];
double *p;

Memory allocated for 800 variables of type double
Memory allocated for one variable of type double*

Therefore:

*salaries = 3.5;
x = salaries[42];

*p = 3.5;
x = p[42];

Arrays are much less flexible than pointers!

int a[800], b[800];
int *p = a, *q = b;

p++;
p = b;
p = q;
a++;
a = b;

Arrays are "constant" pointers. They point forever to the same address!
**Task:** There are 800 salaried employees at a company. Write a C program that computes the average salary, and the number of salaries that are above average. Use **arrays** and **functions**!

```c
void read_array(double a[], int n);
...
double salaries[800];
...
read_array(salaries, 800);
```

What **really** happens?

- What is being passed to the function is **only** the pointer `salaries` to the first element of the array. The first parameter `a[]` of the function is **really just a pointer**, not an array:

  ```c
equivalent
void read_array(double a[], int n);
void read_array(double *a, int n);
```

- The function call effectively executes a pointer assignment to its local parameter: `a = salaries`. Using this pointer, the function can access all the elements of `salaries[800]` as `*a, *(a+1), ..., *(a+799)`.
#include <stdio.h>
define NUM_EMPLOYEES 800

void read_array(double *a, int n);
double sum_array(double *a, int n);
int above_bound(double *a, int n, double bound);

int main(void)
{
    double salaries[NUM_EMPLOYEES], avg;
    int above_avg;

    read_array(salaries, NUM_EMPLOYEES);
    avg = sum_array(salaries, NUM_EMPLOYEES) / NUM_EMPLOYEES;
    above_avg = above_bound(salaries, NUM_EMPLOYEES, avg);
    printf("Average: %lf, Above average: %d\n", avg, above_avg);
    return 0;
}
void read_array(double *a, int n)
{
    int i;

    for (i = 0; i < n; i++)
        scanf("%lf", a+i);  // equivalent to &a[i]
}

double sum_array(double *a, int n)
{
    double sum = 0.0;

    while (n--) sum += *(a+n); // equivalent to a[n]
    return sum;
}

int above_bound(double a[], int n, double bound)
{
    int i, counter = 0;

    for (i = 0; i < n; i++)
        counter += (*(a+i) > bound); // equivalent to a[i]
    return counter;
}