Outline

❖ Need for something new
  ‣ Three shortcomings of call by value

❖ Pointers
  ‣ The address & and indirection * operators
  ‣ Pointers and functions: call by reference
  ‣ Changing function parameters
  ‣ Returning multiple values
  ‣ Pointers and arrays
  ‣ Array arguments

❖ String arrays
  ‣ String functions
  ‣ Command-line arguments
  ‣ File input and output

❖ Dynamic Memory allocation

Perhaps hardest

Lecture 8
ECE15: Introduction to Computer Programming Using the C Language
Problem Changing Parameters

Recall: Function arguments are passed by value

When we write \( f(x) \) to call a function \( f(int \ y) \), value of \( x \) copied into \( y \)

\( f \) has no access to \( x \)

Example: A function that swaps the values of \( x \) and \( y \)

```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);
}
```

What to do?

Passing variables by value

Unchanged!
Example: A function computing the roots of a quadratic equation \( ax^2 + bx + c = 0 \)

```c
#include <stdio.h>

double quad_solve(int a, int b, int c){
    double root[2];
    ...
    root[0] = ... ; root[1] = ... ;
    return root[0], root[1];
}

int main() {
    double a = 1.0, b = 4.0, c = 1.0;
    double x[2];
    x = quad_solve(a,b,c);
    printf("Roots: %d and %d\n", x[0],x[1]);
    return 0;
}
```
Problem with Arrays

Example: Write a function calculating the average of an array of integers

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

int main() {
    int a[1000000] = {1, 2, ...};
    double avg = average(a, 1000000);
}
```

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

What to do?
Summary and Unified Solution

❖ **Shortcomings of call by value**
  - Functions cannot *change* the variables passed to them
  - Functions cannot return *multiple values*
  - Passing *arrays* to functions wastes memory and time

**One idea solves all problems**

Instead of passing variable values

Pass by value

Pass their addresses!

Pass by reference
Outline

❖ Need for something new
  ‣ Three shortcomings of call by value

❖ Pointers
  ‣ The address & and indirection * operators
  ‣ Pointers and functions: call by reference
  ‣ Changing function parameters
  ‣ Returning multiple values
  ‣ Pointers and arrays
  ‣ Array arguments

❖ String arrays
  ‣ String functions
  ‣ Command-line arguments
  ‣ File input and output

❖ Dynamic Memory allocation
Memory Allocation Reminder

Memory organized as sequence of **bytes**
Variables allocated prescribed number of bytes according to **type**
Local variables allocated **consecutive** locations in variable stack

**Example:**

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

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

**Address operator:** \&a is the *address* of variable a

- In example: \&x is 1000, \&d is 1008, and \&c is 1010
- Works only for variables: \&7 is a compilation error

**Dereference (indirection) operator:**
* a is the *variable at address* a, or *pointed to* by a

- In example: *(1000) is x and *(1008) is d

Variables holding addresses are called **pointers**
Pointers

Variables holding the **address** of (pointing to) some variable (or file, or function)

*If pointer \( p \) points to variable \( a \) then \( *p \) is \( a \) and \( p \) is \( \&a \) *

Address \( \& \) and indirection \( * \) are **dual operations**: 
\( \&a \) is address of \( a \) while \( *p \) is variable at address \( p \)
\( \&*p \) is \( p \) and \( *\&a \) is \( a \)
For every type: int, double, char, boolean, ...

There is a type pointing to it: pointer to int, pointer to double, ...

**Declaration:** type *p

int *p;  double *p dbl;  char *cp;  boolean *bp;

**Remember by:** *p (variable p points to) is an integer

**Assignment:** p = &a

Non-pointer types (e.g. int), can be assigned by us:

int x = 5;

For pointers, we don’t know address, hence ask program to initialize, via &

int *p;

p = &x;

Or

int *p = &x;

*p is an int, and the value of p (not of *p) is the address of x
Examples

Declaration + assignment

#include <stdio.h>

int main() {
    int a=1;
    int *pa; // pa points to int
    pa=&a;
    printf("*pa=%d\n", *pa);

    int b=2, *pb;
    pb=&b;
    printf("*pb=%d\n", *pb);

    int c=3, *pc=&c;
    printf("*pc=%d\n", *pc);
}

declarations.c

Pointers to other types

char c, *p1=&c;
double *p2, *p3, x;
p2 = &x;

Declare before reference

int *p4=&i, i;
### Example 1:

```c
int a = 3;
int *p, *q;
p = &a;
q = p;
```

### Example 2:

```c
int a = 30;
int y;
int *p;
p = &a;
y = *p;
*p = 42;
```
Local pointers not auto initialized. If pointer \( p \) not initialized then \( x = *p \) puts nonsense in \( x \) and \( *p = x \) can crash program.

Recall unintuitive syntax when initializing with declaration:

\[
\text{Interpret: } p \text{ is a pointer and its value (not } *p\text{'s value!) is } &x
\]

\textbf{Advantage:} declare and assign multiple variables and pointers in one line

A \textit{single} pointer can also be declared as

\[
\text{int* } p; \quad \text{int* } p, q;
\]

Different pointer types cannot be mixed

\[
\text{int* } p; \quad \text{double } a; \quad \text{p = &a;}
\]

But can be cast

\[
p = (\text{int*}) &a;
\]

Integer pointer now points to double. Still risky, more later.

Several levels of indirection are possible

\[
**p \leftrightarrow *(*p), ***p, ****p, \text{etc.}
\]
Address Printing

- In rare occasions where need to print actual address, can use `%p`
- Prints address in hexadecimal

```c
#include <stdio.h>
#define print_pointer.c
int main() {
    double a=3.14159, *pa=&a;

    printf("Address in hex: %p\n", pa);
    long unsigned int addy = (long) pa;
    printf("Address in decimal: %lu\n", addy);
    printf("Value is: %f\n", *(double*)addy);
    return 0;
}
```
**Precedence**

**Example**

```c
int i = 3, j = 5, *p = &i, *q = &j, *r;
double x;
```

**What happens:**

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

*`&p` is simply `p`*

**binary and unary operators**
Pointers Resolve Shortcomings

Three shortcomings of call by value

- Modifying calling parameters
- Returning multiple values
- Arrays

Show how all resolved by pointers
Modifying Function Parameters

To change function arguments: call with their addresses

Example: A function swapping the values of \( x \) and \( y \)

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

```c
int main() {
    int x = 5, y = 7;
    swap(&x, &y);
    ...
}
```

Functions can now change arguments!
**scanf Mystery Resolved**

**scanf()** updates contents of memory locations, hence called with their address &

**Q:** Must we *always* call with &?

```c
int main() {
    int a;
    scanf("%d", &a);
}
```

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

```c
int main() {
    int a, b;
    swap(&a, &b);
    int *p = &a, *q = &b;
    swap(p, q);
}
```
Returning Two Values

**Task:** Given $a$ and $b$ representing the numerator and denominator of a fraction $a/b$, reduce fraction to its lowest terms --- namely cancel all common factors of $a$ and $b$.

**Example:**

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

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

**Implementation**

Need to return two values!

**Declaration**

\[
void \ reduce(int \ *px, \ int \ *py);
\]

**Call**

\[
int \ a,b;
\ldots \nnreduce(&a, \&b);
\]
Implementation

```c
#include <stdio.h>
int gcd(int, int);

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

int gcd(int m, int n) {
    int tmp;
    while (n != 0) {
        tmp = n;
        n = m % n;
        m = tmp;
    }
    return m;
}

int main() {
    int num, den;
    printf("Numerator + denominator: ");
    scanf("%d %d", &num, &den);
    reduce(&num, &den);
    printf("Reduced: %d/%d\n", num, den);
    return 0;
}
```

Functions can now return two values!
**Example: Adding Two Fractions**

**Task:** Given two fractions $a_1/b_1$ and $a_2/b_2$, represented by their numerators and denominators, compute and reduce their sum $a/b$.

**Example:**

\[
\begin{align*}
\frac{a_1}{b_1} &= \frac{5}{12}, \\
\frac{a_2}{b_2} &= \frac{2}{15}
\end{align*}
\]

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

**Algorithm:**

- $b$ -- least common multiple of $b_1$ and $b_2$
- $a = a_1(b/b_1) + a_2(b/b_2)$

Reduce the resulting fraction

**Implementation:**

```c
void add(int *pa, int *pb, int a1, int b1, int a2, int b2);
int a, b, a1, b1;
... 
add(&a, &b, a1, b1, 2, 15);
```
int gcd(int, int);

void reduce(int *, int *);

int lcm(int m, int n) { // least common multiple
    return m*n/gcd(m,n);
}

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

int main() {
    int a1, a2, b1, b2, a, b;
    printf("a1 b1 a2 b2: ");
    scanf("%d %d %d %d", &a1,&b1,&a2,&b2);
    add(&a,&b,a1,b1,a2,b2);
    printf("%d/%d + %d/%d = %d/%d\n",a1,b1,a2,b2,a,b);
    return 0;
}
Every pointer holds a memory address, namely an integer. **Some, but not all, arithmetic operations allowed**

**Pointer variables cannot be added, multiplied, or divided**

3 arithmetic operations allowed on pointers:

- Add a constant integer: \( p + 3 \)
- Subtract a constant: \( p - 3 \)
- Subtract pointers: \( p - q \)

**Useful for handling arrays**

\[ \text{type } *p, *q; \]

\[ p, q \text{ must point to same type} \]
**Pointer Arithmetic**

** +/- constant:** If $p$ points to a type (int, double, char,..) of size `sizeof(type) = n`, then +/- 1 changes $p$ by $n$ bytes

Similarly: $p++$, $p--$, $++p$, $--p$, $p-42$, $p += 2$, $q = p-2*(++a)$

**Example:**

```c
char a[] = "pointer";
char *p = &a[3];
printf("%c%c%c!!\n", *(p+1), *(p-2), *(p-3));

int b[] = {3, 2, 4, 5, 1};
int *q = &b[3];
printf("%d %d %d!!\n", *(q+1), *(q-2), *(q-3));
```

**Subtracting** pointers of the same type yields # elements of that type between them in memory

```c
int c[] = {3, 2, 4, 5, 1};
printf("%ld\n", &c[3]-&c[1]);
```
**Example:** Assuming an `int` occupies 4 bytes:

Values of following 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>24</td>
</tr>
<tr>
<td><code>(char*)q - (char*)p</code></td>
<td>96</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>0x130</td>
</tr>
</tbody>
</table>

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

The basic use of an array is **referencing** its elements

```c
double salaries[800];
salaries[3] = -1000.;
printf("%d\n", salaries[7]);
```

**How is it implemented?**

- `salaries[0], ..., salaries[799]` are all variables of type `double`

**But what is `salaries` itself?**

- `salaries` is just a pointer to `salaries[0]`

**Array name is same as a pointer to its first element!**

**Array references implemented as:**

- `salaries[i]`  ↔  `*(salaries+i)`
- `salaries[2]`  ↔  `*(salaries+2)`
- `salaries[0]`  ↔  `*salaries`
- `&salaries[i]`  ↔  `salaries+i`
- `&salaries[2]`  ↔  `salaries+2`
- `&salaries[0]`  ↔  `salaries`
Array References via Pointers

Examples

**char array1[] = "pointer";**
```
int i;
for (i=0; i<7; i++) printf("%c", array1[i]);
```

**int array2[] = {0,1,2,3,4,5,6};**
```
int *q = &array2[3];
for (i=-3; i<4; i++) printf("%d", *(q+i));
```

**char a[] = "pointer";**
```
printf("%c%c%c %d", 4[a], 1[a], 0[a], 7[a]);
```
Arrays and Pointers: Differences

Arrays act as pointers to their first element

*Are arrays and pointers the same?*

Arrays do more than pointers

**double salaries[800];** Memory allocated for 800 variables of type double
**double *p;** Memory allocated for one variable of type double*

Therefore:

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

Pointers are more flexible than arrays

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

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

Arrays are "constant" pointers. Always point to same address!
Function call with array argument passes array name: a pointer to array’s first element

```c
void double_array(int ar_name[], int num_elts) {
    for (int i=0; i<num_elts; i++)
        ar_name[i]*=2;
}

void print_array(int *array, int num) {
    for (int i=0; i<num; i++)
        printf("%d ", *(array+i));
    printf("\n");
}
```

```c
int main() {
    int ar[] = {1, 2, 3, 4, 5};
    double_array(ar, 5);
    print_array(ar, 5);
    return 0;
}
```

Q: char word[10]; scanf("%s", word);

A: A string is an array of characters. `word` is a pointer (to `word[0]`). Already address, no need for &.
Arrays in Function Declaration

As we saw:

```c
void fun(int a[]);
equivalent
void fun(int *a);
```

When only declaring function, can omit name

```c
void print_array(int [], int);
void double_array(int *, int);
int main() {
    int ar[]={1, 2, 3, 4, 5};
    double_array(ar, 5);
    print_array(ar, 5);
    return 0;
}
```

```c
void double_array(int arr[], int num) {
    for (int i=0; i<num; i++)
        *(arr+i) *= 2;
}
```

```c
void print_array(int *array, int num) {
    for (int i=0; i<num; i++)
        printf("%d ", array[i]);
    printf("\n");
}
```
2-d Arrays as Pointers

Collection of 2 horizontal arrays, each of size 3

Initialization

\[ \text{int } \text{ar}[2][3] = \{(1,0,2), \{3,9,7\}\}; \]

\[ \text{int } \text{ar}[][3] = \{(1,0,2), \{3,9,7\}\}; \]

Addressing

\[ \text{ar}[i] \quad \text{'i'th array (pointer to its first element)} \]

\[ \text{ar}[i][j], \quad *\text{(ar}[i]+j) \]

\[ \text{ar}+i \quad \text{pointer to 'i'th array} \]

\[ *(\text{ar}+i)[j], \quad *(*(\text{ar}+i)+j) \]

1-dimensional array of size 6

Initialization

\[ \text{int } \text{ar}[][3] = \{1,0,2,3,9,7\}; \]

Addressing

\[ \text{ar} \quad \text{pointer to first element} \]

\[ *(\text{ar})[3*i+j], \quad *(*(\text{ar}+3*i)+j) \]

Crazy C stuff Don’t worry!

2dRef.c
Outline

❖ Need for something new
   ‣ Three shortcomings of call by value

❖ Pointers
   ‣ The address & and indirection * operators
   ‣ Pointers and functions: call by reference
   ‣ Changing function parameters
   ‣ Returning multiple values
   ‣ Pointers and arrays
   ‣ Array arguments

❖ String arrays
   ‣ String functions
   ‣ Command-line arguments
   ‣ File input and output

❖ Dynamic Memory allocation
Alternative declarations

// As arrays:
char class1[]="ece15"; // array
char class5[] = {'e','c','e','1','5','\0 '};

// As pointers:
char *class2 ="ece15"; // pointer
char* class3 ="ece " "15"; // concatenation
char* class4; class4="ece15"; // why?
...

char *pword;
scanf("%s", pword);
Useful String Functions

```
<int string.h>

int strlen(char *); // String length

actually, size_t, used for object sizes, can treat as integer

char *strcat(char *, char *); // Concatenate 1st, 2nd, returns location

char *strcpy(char *, char *); // Copies 2nd to 1st, returns destination

int strcmp(char *, char *); // 1st string <, =, or > than 2nd

<int stdio.h> and <stdlib.h>

int sscanf(char *string, char *format, arg1,..); // Read from string

int sprintf(char *string, char *format, arg1,..); // Write to string

int atoi(char *); // Convert to int

device atof(char *); // to double
```

Lecture 8
Arrays of int, double, char, .... *everything*

How about array of... *pointers*?

```
#include <stdio.h>

int main() {
    char *stringArray[] = {
        "First string",
        "Second string",
        "Third string"
    };
    int i;
    for (i=0; i<3; i++)
        printf("%s\n", stringArray[i]);
    return 0;
}
```

---

Many applications

One next
Command-Line Arguments

Convenient way to run frequent commands

Task: Convert Celsius to Fahrenheit

If run frequently, skip prompt

Provide the program command-line arguments

 Almost all Unix commands use command-line arguments

Lecture 8
C provides the program two values

- Integer containing number of strings in command line (incl. filename) - typically `argc`
- Array of pointers, each pointing to one of the strings - typically `argv`

```
> a.out Hello 1 world
```

```
argc: 4
```

```
#include <stdio.h>  

int main(int argc, char *argv[]) {
    int i;
    printf("%d arguments\n", argc-1);
    for (i=1; i<argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

```
What if all values integers, wanted to increment all?
```

```
printf("%d\n", atoi(argv[i])+1);
```

```
print_arguments.c
```

```
increment.c
```

#include <stdio.h>

int main() {
    double cel;
    printf("Celsius: ");
    scanf("%lf", &cel);
    printf("Fahrenheit: %.1lf\n", cel*1.8+32);
    return 0;
}

Command line arguments

Before

#include <stdio.h>

int main() {
    double cel;
    printf("Celsius: ");
    scanf("%lf", &cel);
    printf("Fahrenheit: %.1lf\n", cel*1.8+32);
    return 0;
}

After

#include <stdio.h>

int main(int argc, char *argv[]) {
    int cel = atoi(argv[1]);
    printf("Fahrenheit: %.1lf\n", cel*1.8+32);
    return 0;
}

Lecture 8
ECE15: Introduction to Computer Programming Using the C Language
Memory addresses are not of type `int`. Nevertheless, every pointer variable can take one special integer value: `constant 0`.

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

0 is never a valid address.

Setting pointer to `NULL` is a safe way to indicate it doesn’t point anywhere.

```c
int *p = (int *)100;
*p = 42; /* What will happen? */
```

```c
int i, *p = NULL;
....
p = &i;
if (p!=NULL) *p = 42;
if (!p) *p = 42;
```
FILE *pin = fopen("inputName", "r");
FILE *pout1 = fopen("outputName1", "w");
FILE *pout2 = fopen("outputName2", "a");

fscanf(pin, "%d", &x);

fclose(pin);
fclose(pout1);
fclose(pout2);

fprintf(pout1, "%d\n", x);
fprintf(pout2, "%d\n", 2*x);

String because more options: "r+", "w+", "wb", etc.

Erases everything previously there

Read
Write
Close
Example

Read values from input file, write squares in output file

```c
#include <stdio.h>

int main() {
    int x;
    FILE *input  = fopen("inputFileName", "r");
    FILE *output = fopen("outputFileName", "w");
    while (fscanf(input, "%d", &x) == 1)  
        fprintf(output, "%d\n", x*x);
    fclose(input);
    fclose(output);
    return 0;
}
```

`fopen` fails if "r" file doesn’t exist, no space for "w" file, etc. Returns NULL pointer, hence can check

```c
FILE *input
if ( (input = fopen("inputFileName", "r") ) == NULL ) {
    printf("Couldn’t open inputFileName\n");
    return 1;
}
... all okay... continue normal execution
```
Overwrite File

Double all values in a file, writing new values in same file

```c
#include <stdio.h>
#define MAX_NUM 100
int main() {
    int ar[MAX_NUM], i=0, j;
    FILE *fp = fopen("FileName", "r");
    while (fscanf(fp, "%d", &ar[i]) == 1)
        i++;
    fclose(fp);
    fp = fopen("FileName", "w");
    for (j=0; j<i; j++)
        fprintf(fp, "%d\n", 2*ar[i]);
    fclose(fp);
    return 0;
}
```

Double pay!!!

Need to store values
Use array

What if want to write at end of file?
open with “a”

No size bound?
Temporary file or dynamic memory allocation
Need for something new

- Three shortcomings of call by value

Pointers

- The address & and indirection * operators
- Pointers and functions: call by reference
- Changing function parameters
- Returning multiple values
- Pointers and arrays
- Array arguments

String arrays

- String functions
- Command-line arguments
- File input and output

Dynamic Memory allocation
Pointers to Void

Seen pointers to int, char, double, ....

What's a pointer to void?
What if you need a pointer whose type is known only at run-time?

Use pointer of type `void`, then cast it at run-time.

Pointers of `void`

Generic: do not to variable of specific type (`int`, `char`, etc.)

Can assign address

Cannot reference, do pointer arithmetic, etc. (*don’t know # bytes to use*)

To reference need to first cast, so point to a specific type

**Examples**

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

- ✔ just address
- no cast needed

```c
double a=5., b, *p1=&a;
void *p2
p2 = p1;
b = *p1 + *(double*)p2;
```

- ✔ just address
- cast needed
- cast needed
Dynamic Memory Allocation

Programs often require memory unpredictable at compilation

Variable whose type depends on input

Array whose size depends on input, or changes with time

What to do?

Allocate memory during run time
malloc and calloc

Defined in `<stdlib.h>`

Allocate requested number of bytes

Memory is contiguous

Return pointer to first byte (type `void*`)

Return `NULL` if unsuccessful

Memory automatically released only when program exits

Programmer can release earlier

Difference: call and initialization
Allocate memory for a double and store π in it

double *p;
p = (double*) malloc(sizeof(double));
*p=3.1415;

Allocate memory for character or integer and initialize it

char dataType, *pc;
int *pi;

printf("Type c for char, i for int: ");
scanf("%c", &dataType);

if (dataType=='c') {
    pc = (char*) malloc(sizeof(char));
    if (pc == NULL) return 1;
    *pc = '\0';
}
else {
    pi = (int*) malloc(sizeof(int));
    if (pi == NULL) return 1;
    *pi = 0;
}
double *p;
p = (double*) calloc(10, sizeof(double));

*p can be viewed as pointing to array of 10 doubles

Equivalently:

double *p;
p = (double*) malloc(10*sizeof(double));

for (int i=0; i<10; i++)
    p[i]=0;

Initialized!

Similar to malloc
Takes # items & size of each
Allocates memory
Initialized to 0
Dynamic vs. Compile-Time Memory

Very similar, but..

Compile-time memory has name

  Can be addressed using name and pointers

Run-time memory has no name

  Can be addressed only using pointers

```c
int v, *pv=&v;
v=3;
*pv=3;
int *pw = (int*) malloc(sizeof(int));
*pw = 3; // *pw has no name
```
Garbage Collection

Memory automatically released when program terminates

**User can release earlier**

free, defined in `<stdlib.h>`

```c
double *p;
p = (double*) malloc(sizeof(double));
// ... work with p
free(p);
```

Similarly for `calloc`

Memory size not necessary, known to operating system
Salaries Revisited

#include <stdio.h>
#include <stdlib.h>

int main()
{
    double *salaries;
    int n, i;

    printf("Num employees: ");
    scanf("%d", &n);  // read # employees
    salaries = (double*) malloc(n * sizeof(double));
    if (salaries == NULL)
        return 1;
    for (i=0; i<n; i++)
        scanf("%lf", salaries+i);
        // ...compute average, stdev, median, etc...
    free(salaries);
    ...
}