Modular Design

- Why modularity
- Functions

Functions

- Declaration
- Definition
- Call

Details

- Parameters
- Whole arrays
- Order

Revisits

- gcd
- Newton Raphson
- Pi

Mechanics

- Call
- Return
- Calling variables unchangeable

Array parameters

- Elements
- Whole arrays
- 2-d arrays

Libraries

- stdio.h
- math.h
- stdlib.h
- Your own

Variable Storage classes

- Local
- static
- Global

Recursion
The programs encountered so far were fairly small. Complex tasks require large, complicated programs, often many thousands lines of code.
Modular Design

Large programs - how to:

- write?
- debug / modify?
- make understandable?
- reuse repeated code segments?

**Modular design**

Break overall task into smaller modules, each constructed from even smaller ones:

- Construct each part individually
- Easier to debug, modify and understand
- Reuse similar code parts in many places
**Problem:** Need to prepare a meal of spaghetti bolognese.
Solve this complex problem by breaking it into smaller, easier, ones:

```c
void spaghetti_bolognese() {
    prepare_spaghetti();
    prepare_sauce(SPICY);
    mix();
    serve();
}
```

```c
void prepare_spaghetti() {
    boil_water();
    add_spaghetti();
    while (!ready) {
        wait(2);
    }
}
```
Modular Design in C: Functions

- Modular design implemented using **functions**
- Convenient way to encapsulate long computations
  - Accept **parameters**
  - **Return** a value
- In other languages: “procedures”, “routines”, “methods”
- Every C program is just a collection of functions
- Functions:
  - Make **writing** and **modifying** code easier
  - Improve **readability** and facilitate **debugging**
  - Avoid code **duplication** and enable code **reuse**

"C has been designed to make functions efficient and easy to use. C programs generally consist of many small functions rather than a few big ones."

D. Ritchie  B. Kernighan
Outline

❖ Modular Design
  ▷ Why modularity
  ▷ Functions

❖ Functions
  ▷ Declaration
  ▷ Definition
  ▷ Call

❖ Details
  ▷ Parameters
  ▷ Declaration
  ▷ Order

❖ Revisits
  ▷ gcd
  ▷ Newton Raphson
  ▷ Pi

❖ Mechanics
  ▷ Call
  ▷ Return
  ▷ Calling variables unchangeable

❖ Array parameters
  ▷ Elements
  ▷ Whole arrays
  ▷ 2-d arrays

❖ Libraries
  ▷ stdio.h
  ▷ math.h
  ▷ stdlib.h
  ▷ Your own

❖ Variable Storage classes
  ▷ Local
  ▷ static
  ▷ Global

❖ Recursion
Three Ingredients of Functions

**Declaration**
Tells compiler what to expect: **prototype**
Consists of *name, parameters, and type of value returned*

```
return-type function-name(parameter-list);
```

**Definition**
Complete description of what function does

**Declaration** (as above)

**Body** - compound statement implementing the function

```
return-type function-name(parameter-list) {
    variable definitions
    statements // do what the function should do
    return statement
}
```

**Call**
Computes the function

```
function-name(parameter-list);
```

- **Explain by Examples**
Basic Function

**Example:** Function squaring an integer

<table>
<thead>
<tr>
<th>Declaration</th>
<th>int square(int x);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned-value type</td>
<td>int</td>
</tr>
<tr>
<td>Name</td>
<td>square</td>
</tr>
<tr>
<td>Parameter list &amp; type</td>
<td>(int x)</td>
</tr>
</tbody>
</table>

**Definition**

```c
int square(int x) {
    return x*x;
}
```

**Call**

```c
square(5);
```

**Whole program**

```c
#include <stdio.h>
int square(int x);
int main() {
    int value, sqr;
    printf("Integer: ");
    scanf("%d", &value);
    sqr = square(value);
    printf("Square: %d\n", sqr);
}
```

Variable names in declaration, deficit and call can differ.

**Name**
Follows variable name rules. Cannot be same as a variable in the program.
What Happens

❖ main() executed

❖ Calls function

❖ Function executed from beginning
❖ Continues till return
❖ Returns value following return
❖ Execution goes back to main

#include <stdio.h>
int square(int x);

int main() {
    int value, sqr;
    printf("Integer: ");
    scanf("%d", &value);
    sqr = square(value);
    printf("Square: %d\n", sqr);
}

int square(int x) {
    return x*x;
}
Variable Names Can Differ

- Variable names can differ at declaration, definition, and call

```c
#include <stdio.h>

int square(int y);

int main() {
    int value;
    printf("Integer: ");
    scanf("%d", &value);
    printf("Square: %d\n", square(value));
}

int square(int x) {
    return x*x;
}
```

- Declaration: just prototype, name irrelevant (later: can even be removed)
- Call: can apply to different variables, no need to know actual definition

```
square(value);
square(radius);
square(5);
```
Multiple Parameters

❖ Functions can have multiple parameters

❖ *Example:* \( x^n \) (\( n \) - nonnegative integer)

```c
#include <stdio.h>

double power(double x, int n);

int main() {
    double base;
    int exponent;
    printf("base and power: ");
    scanf("%lf %d", &base, &exponent);
    printf("%g^%d = %g\n", base, exponent, power(base, exponent));
}

double power(double a, int m) {
    double p = 1.0;
    int i;
    for (i = 0; i < m; i++)
        p *= a;
    return p;
}
```

*Declaration*

*Call*

*Definition*
Functions can have several return statements

First `return` encountered ends computation and returns value

**Example:** Absolute value of a double

```c
#include <stdio.h>

double abs_val(double y);

int main() {
    double v;
    printf("Type values, non numeric to exit: ");
    while (scanf("%lf", &v) == 1)
        printf("%g
", abs_val(v));
    return 0;
}

double abs_val(double x) {
    if (x<0)
        return -x;
    else
        return x;
}
```

Still returns only one value!
Some functions perform a task and return no value

**Example:** Function to facilitate and organize message printing
Called with message index as parameter, prints appropriate message
Does not return any value, `void` return value

```c
#include <stdio.h>

void messages(int i) {
    switch (i) {
    case 0:  printf("Finished successfully.\n");
             break;
    case 1:  printf("Input error.\n");
             break;
    case 2:  printf("Run-time error.\n");
             break;
    default: printf("Invalid parameter.\n");
             break;
    }
}
```

Void fun’s can have `return` - but without value

```c
#include <stdio.h>

void messages(int i);

int main() {
    ...
    if (..`bad input'..)
        messages(1);
    ...
}
```
Another Void Example

- Recall ordinal form of nonnegative integer (1st, 2nd, 24th,...)

```c
#include <stdio.h>

void print_ordinal(int x);

int main() {
    int x;
    printf("Nonnegative integer: ");
    scanf("%d", &x);
    print_ordinal(x);
    return 0;
}

void print_ordinal(int x) {
    int y=x%100, z;
    if (y>=11 && y<=13)
        z=4;
    else
        z=x%10;
    printf("Ordinal: %d%s\n", x,
            z==1 ? "st" : z==2 ? "nd" : z==3 ? "rd" : "th");
}
```

Can easily print in many different locations.

- Can add "return;" (without any parameter)
Void vs. Valued Functions

Difference in `return`

- Functions returning a value (int, double, etc.) typically have `return X;`
  ```c
  int square(int x) {
    return x*x;
  }
  ```
  **Some value**

- Functions returning nothing (void) have either NO return, or `return;`
  ```c
  void messages(int i) {
    return;
  }
  ```
  **No value**

Difference in `call`

- Functions returning value, have their return value utilized in the call:
  ```c
  sqr = square(x);
  printf("%d", square(x));
  ```

- Functions returning nothing, don’t have their value utilized in the call:
  ```c
  messages(i);
  ```
Outline

❖ Modular Design
  › Why modularity
  › Functions

❖ Functions
  › Declaration
  › Definition
  › Call

❖ Details
  › Parameters
  › Declaration
  › Order

❖ Revisits
  › gcd
  › Newton Raphson
  › Pi

❖ Mechanics
  › Call
  › Return
  › Calling variables unchangeable

❖ Array parameters
  › Elements
  › Whole arrays
  › 2-d arrays

❖ Libraries
  › stdio.h
  › math.h
  › stdlib.h
  › Your own

❖ Variable Storage classes
  › Local
  › static
  › Global

❖ Recursion
Parameters

❖ So far, functions had parameters
❖ A function can have **no parameters**
❖ Declared by *void* or (), namely **default** parameter list is *void*

```c
int getchar(void);
int getchar();
```

❖ We have seen this default use before

```c
int main(void);
int main();
```

❖ When parameters exist, **default** parameter type is **int**

```c
int square(int x);
int square(x);
```

▷ may produce warning - best avoid
Parameters Passed by Location

- Parameters passed according to location in list, not name

```c
#include <stdio.h>

int fun(int x, int y) {
    return x/y;
}

int main() {
    int x=6, y=3;
    printf("%d\n", fun(x, y));
    printf("%d\n", fun(y, x));
    return 0;
}
```

Quiz
Parameter Passed by Value

- Pass value of parameter
- Variables inside and outside function bear no relation to each other
- Can have same name
- Modifying variable in function doesn’t affect same-name variable in main
- No worry about names in function

```
#include <stdio.h>

int main() {
    int v=2, w=5, x=7;
    printf("main: v=%d x=%d\n", v, x);
    printf("main: w=%d x=%d\n", w, x);
    printf("main: x=%d\n", x);
    return 0;
}

int dbl(int x) {
    printf("dbl-1: x=%d\n",x);
    x *= 2;
    printf("dbl-2: x=%d\n",x);
    return x;
}
```

No relation
Returned Values

❖ Functions terminate upon encountering any `return`, or final `}

❖ Return a value of type specified in declaration
  ▸ Always return at most one value, specified after `return`
  ▸ If multiple `return` statements, first encountered returns value
  ▸ Returned value converted to declared type
  ▸ If there is no return by end of function, returns gibberish

❖ Functions can return no value
  ▸ Declared as `void`
  ▸ No return or nothing follows return

❖ Default return type is `int` (may produce warning)
 Declarations

❖ Tell compiler of function **prototype** (parameters- and returned types)
❖ Must **precede** function call
❖ Must agree with definition on all types

```c
int square(int x);
int main() {...}
double square(float x) {...}
```

❖ Since declaration just informs of types, can omit variable names

```c
int square(int);
int main() {...}
int square(int x) {...}
```

```
int power(int base, int exp);
int main() {...}
int power(int base, int exp){...}
```

Which is better?
Main

- No parameters (for now)
- Returns status code to operating system
  - 0 - okay
- **Must** appear
- **First** function executed

```c
int main() {
  ... 
  return 0;
}
```
Order of Functions

- Functions typically appear **sequentially, not inside** other functions.

```
int main() {...}
int square(int x) {...}
```

- **Order** of functions does **not matter**.

```
int square(int x) {...}
int main() {...}
```

- Except, they must be declared before called, **see next slide**.

```
#include <stdio.h>
int square(int x);
int main() {
    int value;
    printf("Integer: ");
    scanf("%d", &value);
    printf("Square: %d\n", square(value));
}
int square(int x) {
    return x*x;
}
```

```
#include <stdio.h>
int main() {
    int value;
    printf("Integer: ");
    scanf("%d", &value);
    printf("Square: %d\n", square(value));
}
int square(int x) {
    return x*x;
}
```

```
much less common
```

#lecture 7 ece15 introduction to computer programming using the c language

24
Recall: Definition includes declaration information

- If \textit{definition} appears \textit{before call}, separate \textit{declaration} not needed

\begin{verbatim}
int sqr(int x);

int main() {
    int n;
    printf("Integer: "); ...
    printf("Square: %d\n", sqr(n));
}

int sqr(int x) {
    return x*x;
}

int main() {
    int n;
    printf("Integer: "); ...
    printf("Square: %d\n", sqr(n));
}
\end{verbatim}

\begin{verbatim}
int fact(int n) {
    return n==1 ? 1 : n*fact(n-1);
}

int main() {
    int n;
    printf("Factorialize: "); ...
    printf("%d!=%d\n", n, fact(n));
}
\end{verbatim}

No declaration!!!

Definition b4 call

Definition after call
Declaration needed

Definition after call
Declaration needed
Which Should Come First?

Small Programs

❖ Define functions first
❖ Avoids declarations & duplication
❖ Prepare ingredients before use
❖ Clean

Larger Programs

❖ Don’t know which fun’s call others
❖ Some may call each other (risky!)
❖ Declare all functions first
❖ Then call in any order

#include <stdio.h>

```c
int max(int a, int b) { ... }
int min(int a, int b) { ... }
int main() { max(x,y); ... }
```

### Small Programs

```c
#include <stdio.h>

int max(int a, int b);
int min(int a, int b);
int main() { max(x,y); ... }
```
Outline

❖ Modular Design
  ❧ Why modularity
  ❧ Functions

❖ Functions
  ❧ Declaration
  ❧ Definition
  ❧ Call

❖ Details
  ❧ Parameters
  ❧ Declaration
  ❧ Order

❖ Revisits
  ❧ gcd
  ❧ Newton Raphson
  ❧ Pi

❖ Mechanics
  ❧ Call
  ❧ Return
  ❧ Calling variables unchangeable

❖ Array Parameters
  ❧ Elements
  ❧ Whole arrays
  ❧ 2-d arrays

❖ Libraries
  ❧ stdio.h
  ❧ math.h
  ❧ stdlib.h
  ❧ Your own

❖ Variable Storage classes
  ❧ Local
  ❧ static
  ❧ Global

❖ Recursion
GCD of Three Integers

Euclidean Algorithm computes greatest common divisor of two integers, \( \text{gcd}(a,b) \)

How to compute greatest common divisor of three integers, \( \text{gcd}(a,b,c) \)?

\[
\text{gcd}(a,b,c) = \text{gcd}(\text{gcd}(a,b), c)
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>gcd(a,b,c)</th>
<th>gcd(a,b)</th>
<th>gcd( gcd(a,b), c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>24</td>
<td>18</td>
<td>6</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>40</td>
<td>60</td>
<td>25</td>
<td>5</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Conclusion: Use euclidean algorithm twice
Program with Functions

❖ Without functions, write gcd code twice
❖ With functions - only once

```c
#include <stdio.h>

int gcd(int, int);

int main() {
    int a, b, c, g_ab, g_abc;
    scanf("%d%d%d", &a, &b, &c);
    g_ab = gcd(a, b);
    g_abc = gcd(g_ab, c);
    printf("gcd: %d\n", g_abc);
    return 0;
}
```

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

```
printf("gcd: %d\n", gcd(gcd(a, b), c));
```

Implement formula using the function `gcd(m, n)`

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

```
printf("gcd: %d\n", gcd(gcd(a, b), c));
```

Shorter?

❖ Without functions, write gcd code twice
❖ With functions - only once
```c
#include <stdio.h>
#define MAX_ITERATIONS 10

double f(double x);
double df(double x);

int main() {
    double x, fx, dfx;
    int i;

    printf("Initial point: ");
    scanf("%lf", &x);
    fa = f(x);

    for (i = 0; i < MAX_ITERATIONS && (fx != 0); ++i) {
        fx = f(x);
        dfx = df(x);
        x -= fx/dfx;
        printf("Iteration: %d, Solution: %.12f\n", i, x);
    }
    printf("Solution is: %.12f\n", x);
    return 0;
}

double f(double x) {
    return x*x-9;
}

double df(double x) {
    return 2*x;
}
```

Without functions, examine whole program
With functions, change just relevant functions

nr_fun.c
Perimeter of circumscribed n-gon: $a_n$

Perimeter of a circle: $2\pi r = \pi$

Perimeter of circumscribing n-gon: $b_n$

For all $n$: $a_n < \pi < b_n$

As $n$ increases, approximation improves

$\frac{1}{2}$

$b_4 = 4$

$a_4 = 2\sqrt{2}$

$b_8 = ?$

$a_8 = ?$

For all $n$:

$b_{2n} = \frac{2*a_nb_n}{(a_n + b_n)}$

$a_{2n} = \sqrt{a_nb_{2n}}$
```c
#include <stdio.h>
#define SR_ITER 35
#define PI_ITER 30

int main () {
    double a, b=4, x=1;
    int i, j;
    for (j = 0; j < SR_ITER; j++) {
        double fx = x*x - 2;
        double dfx = 2*x;
        if (dfx == 0) break;
        x -= fx/dfx;
    }
    a = 2*x;
    printf("a=%.15f  b=%.15f\n",a,b);
    for (i=0; i<PI_ITER; i++) {
        double b = 2*a*b/(a+b);
        x=1;
        for (j=0; j < SR_ITER; j++) {
            double fx = x*x - a*b;
            double dfx = 2*x;
            if (dfx == 0) break;
            x -= fx/dfx;
        }
        double a = x;
        printf("a=%.15f  b=%.15f\n", a,b);
    }
    return 0;
}
```

**Simpler with functions**

- `sqrt(2)`: Square root of 2
- `sqrt(a*b)`
Approximating $\pi$ - with Functions

```c
#include <stdio.h>
define PI_ITER 30
double sqrt(double);
int main () {
    int i;
    double a, b=4;
    a = 2*sqrt(2);
    printf("a=%.15f  b=%.15f\n",a,b);
    for (i=0; i<PI_ITER; i++) {
        b = 2*a*b/(a+b);
        a = sqrt(a*b);
        printf("a=%.15f  b=%.15f\n",a,b);
    }
    return 0;
}
```

```c
#define SR_ITER 35
double sqrt(double a) {
    int i;
    double x=1, fx, dfx;
    for (i=0; i<SR_ITER; i++) {
        fx = x*x - a;
        dfx = 2*x;
        if (dfx == 0) break;
        x -= fx/dfx;
    }
    return x;
}
```
Task:
Compute area of a circle of radius \( r \)

Function \( \text{pi}() \) calculates \( \pi \) iteratively

Called from main() to evaluate circle area

```c
#include <stdio.h>

double pi(void);
double sqrt(double);

int main() {
    double r;

    printf("Radius: ");
    scanf("%lf", &r);

    printf("Area: %.20f\n", pi() * r * r);
    return 0;
}

double pi(void) {
    double a, b;
    int i;
    b = 4;
    a = 2 * sqrt(2);
    for (i = 0; i < 20; i++) {
        b = 2 * a * b / (a + b);
        a = sqrt(a * b);
    }
    return a;
}
```

```c
#define SR_ITER 35

double sqrt(double a) {
    int i;
    double x = 1, fx, dfx;
    for (i = 0; i < SR_ITER; i++) {
        fx = x * x - a;
        dfx = 2 * x;
        if (dfx == 0) break;
        x -= fx / dfx;
    }
    return x;
}
```
Outline

❖ Modular Design
  ‣ Why modularity
  ‣ Functions

❖ Functions
  ‣ Declaration
  ‣ Definition
  ‣ Call

❖ Details
  ‣ Parameters
  ‣ Order

❖ Revisits
  ‣ gcd
  ‣ Newton Raphson
  ‣ Pi

❖ Mechanics
  ‣ Call
  ‣ Return
  ‣ Calling variables unchangeable

❖ Array parameters
  ‣ Elements
  ‣ Whole arrays
  ‣ 2-d arrays

❖ Libraries
  ‣ stdio.h
  ‣ math.h
  ‣ stdlib.h
  ‣ Your own

❖ Variable Storage classes
  ‣ Local
  ‣ static
  ‣ Global

❖ Recursion
What happens when the function is called?

- Memory allocated to function parameters `double x` and `double y`.
- Expressions `9.9*a` and `2/lambda` evaluated. Evaluation order unspecified.
- Resulting values assigned (copied into) variables `double x` and `double y`. Type conversions performed as needed. This is known as call by value*. All functions in C are called by value.
- Memory allocated to variables declared inside the function body, and the function-body block is executed.

*Alternative is call by reference where, as in PERL, a reference to the variable (e.g. its address) is passed to function. In C, the reference can be passed via pointers (next lecture).
What Happens at Function Return?

Again:

```c
double my_function(double x, double y) {...};
```

Called as:

```c
z = my_function(9.9*a,2/lambda);
```

**What happens when the function returns?**

- ✔️ If function contains `return expression;` function execution stops when the (first such) statement is reached.

- ✔️ `expression` evaluated, its type converted to `double` if needed, result is assigned to the variable `z`.

- ✔️ If a `return expression;` statement is never reached, execution stops with the closing brace of the function, `z` assigned arbitrary value.

- ✔️ In both cases, all memory allocated to the function parameters and local variables is released, and calling program continues with next statement.
Recall \( \text{gcd}(\text{int } m, \text{int } n) \), computing the greatest common divisor of two integers using the Euclidean algorithm.

```c
int gcd(int m, int n)
{
    return m;
}
```

```c
int foo(void)
{
    d = gcd(921, 2);
    return gcd(1, 2);
}
```

Function Call & Return: Example
Cannot Change Calling Parameters

When \( f(x) \) calls a function \( f(int \ y) \), value of \( x \) copied into \( y \)

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

\( f \) cannot change \( x \)

**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);
}
```

Swapping jinxed!

Next Lecture
Changing calling parameters

Unchanged!
A: What does the following program print?

```c
#include <stdio.h>
int f(int x) {
    return (++x);
}
int main(void) {
    int a = 0;
    printf("f(%d) = %d\n", a, f(a));
    printf("a = %d\n", a);
    return 0;
}
```

B: Why `int x; scanf("%d", x);` can’t work?
Outline

❖ Modular Design
   ‣ Why modularity
   ‣ Functions

❖ Functions
   ‣ Declaration
   ‣ Definition
   ‣ Call

❖ Details
   ‣ Parameters
   ‣ Declaration
   ‣ Order

❖ Revisits
   ‣ gcd
   ‣ Newton Raphson
   ‣ Pi

❖ Mechanics
   ‣ Call
   ‣ Return
   ‣ Calling variables unchangeable

❖ Array parameters
   ‣ Elements
   ‣ Whole arrays
   ‣ 2-d arrays

❖ Libraries
   ‣ stdio.h
   ‣ math.h
   ‣ stdlib.h
   ‣ Your own

❖ Variable storage classes
   ‣ Local
   ‣ static
   ‣ Global

❖ Recursion
Array Elements as Parameters

❖ Both **array elements**, and **whole arrays** can be function parameters
❖ To pass **array element** use: `array[1]`, `array[i]`, etc.

**Example:** Replace all array elements by their absolute value, print all

```c
#include <stdio.h>

int abs_val(int x) {
    return x>0 ? x:-x;
}

int main() {
    int i, myArray[5]={-1, 2, -4};
    for (i=0; i<5; i++)
        myArray[i] = abs_val(myArray[i]);
    for (i=0; i<5; i++)
        printf("%d ",myArray[i]);
    printf("\n");
    return 0;
}
```
Whole Arrays as Parameters

❖ To pass a **whole array** as function parameters, pass array name
❖ Need to pass length too!

```c
#include <stdio.h>

void absAll(int ar[], int length) {
    for (int i=0; i<length; i++)
        ar[i] = ar[i]>0 ? ar[i] : -ar[i];
    return;
}

void printAll(int ar[], int length) {
    for (int i=0; i<length; i++)
        printf("%d ", ar[i]);
    printf("\n");
}

int main() {
    int myArray[5] = {-1, 2, -4};
    absAll(myArray, 5);
    printAll(myArray, 5);
    return 0;
}
```

Unlike regular variables (int, etc.), array parameters can be changed by called functions.

More in next lecture
Both Elements and Whole Arrays

- Like previous examples, show both elements and whole array parameters.
- **Element** param’s (absAll to absVal): declare as int, pass as array[i].
- **Array** param’s (main to absAll, printAll): decl. int ar[], pass myArray.

```c
#include <stdio.h>

int absVal(int x){
    return x>0 ? x : -x;
}

int main(){
    int myArray[5] = {-1,2,-4};
    absAll(myArray, 5);
    printAll(myArray, 5);
    return 0;
}

void absAll(int ar[], int length){
    for (int i=0; i<length; i++)
        ar[i] = absVal(ar[i]);
    return;
}

void printAll(int ar[], int length){
    for (int i=0; i<length; i++)
        printf("%d ",ar[i]);
    printf("\n");
    return;
}
```

Again note that a function can modify array parameters!

More next lecture.
Example: Read array and calculate smallest element

Recall: Declaration does not require parameter names

This holds also for array parameters

```c
#include <stdio.h>
int smallest(int [], int);
void readAll(int [], int);

int main() {
    int num_elts;
    printf("# elements: ");
    scanf("%d", &num_elts);
    int array[num_elts];
    readAll(array, num_elts);
    printf("%d is smallest\n", smallest(array, num_elts));
    return 0;
}

int smallest(int array[], int num) {
    int i, min=array[0];
    for (i=1; i<num; i++)
        if (array[i]<min)
            min=array[i];
    return min;
}

void readAll(int ar[], int num) {
    for (int i=0; i<num; i++)
        scanf("%d", &ar[i]);
    return;
}
```

Again note that readAll can modify the array

Small functions make for more readable code!
#include <stdio.h>
#define SIZE 5

void absAll(int []);

int main() {
    int myArray[SIZE]={-1, 2, -4};
    absAll(myArray);
    return 0;
}

void absAll(int ar[]){
    for (int i=0; i<SIZE; i++)
        ar[i] = ar[i]>0 ? ar[i] : -ar[i];
    return;
}
Passing 2-dimensional Arrays

```c
#include <stdio.h>
#define SIZE 3

void transpose(int ar[][SIZE]);
void printMatrix(int mat[][SIZE]);

int main() {
    int mat[][SIZE] = {
        {1, 2, 3, 4, 5, 6, 7, 8, 9};
    }
    printf("Matrix:\n");
    printMatrix(mat);
    transpose(mat);
    printf("Transpose:\n");
    printMatrix(mat);
    return 0;
}

void transpose(int a[][SIZE]) {
    int i, j, temp;
    for (i=1; i<SIZE; i++)
        for (j=0; j<i; j++) {
            temp = a[i][j];
            a[i][j] = a[j][i];
            a[j][i] = temp;
        }
}

void printMatrix(int b[][SIZE]) {
    int i, j;
    for (i=0; i<SIZE; i++)
        for (j=0; j<SIZE; j++)
            printf("%d %s", b[i][j], j==SIZE-1? "\n":"");
}
```
Passing a Row of a 2-dim Array

```c
#include <stdio.h>
#define NR 3 // number of rows
#define NC 4 // number of columns

void initialize(int array[][NC]){
    for (int i=0; i<NR; i++)
        for (int j=0; j<NC; j++)
            array[i][j]=i*NC+j;
    return;
}

void print_one_dim(int array[], int length) {
    for (int i=0; i<length; i++)
        printf("%d ", array[i]);
    printf("\n");
    return;
}

int main() {
    int i, two_dim_array[NR][NC];
    initialize(two_dim_array);
    for(i=0; i<NR; i++) {
        printf("Row %d: ", i);
        print_one_dim(two_dim_array[i], NC);
    }
    return 0;
}
```

To pass row i of a 2-d array, pass `array[i]`

Initialize a 2-d array

Print it line by line

pass whole 2-d array

pass one row of a 2-d array
#include <stdio.h>
#define SZ 5

void doit(            );
int main(){
 int matrix[SIZE][SIZE];
 doit(     );
 return 0;
}

<table>
<thead>
<tr>
<th></th>
<th>call</th>
<th>definition</th>
<th>declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix (3,4)</td>
<td>matrix[3][4]</td>
<td>int a</td>
<td>int int b</td>
</tr>
<tr>
<td>whole matrix</td>
<td>matrix</td>
<td>int a[SZ][SZ]</td>
<td>int b[SZ][SZ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int a[][SZ]</td>
<td>int b[][SZ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int a[]</td>
<td>int b[]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Int [SZ][SZ]</td>
</tr>
</tbody>
</table>
A picture is worth a 1000 bytes

Cow getting out of focus?

What’s the connection?

Pictures represented as 2-d arrays
Each location corresponds to a picture element (pixel)
Its value indicates the color, 0 for black to 255 for white

Relevance

Image processing is an important application of programming

How to blur a picture?

Replace each pixel by the average of its surrounding pixels
Blurring Example

Pixel value will be replaced by average of 9 surrounding ones (here 112)

At edges average over fewer neighbors
## Blurring Example

### After blurring:

<table>
<thead>
<tr>
<th>80</th>
<th>79</th>
<th>84</th>
<th>75</th>
<th>79</th>
<th>71</th>
<th>84</th>
<th>89</th>
<th>97</th>
<th>95</th>
</tr>
</thead>
<tbody>
<tr>
<td>77</td>
<td>75</td>
<td>93</td>
<td>99</td>
<td>114</td>
<td>109</td>
<td>120</td>
<td>113</td>
<td>105</td>
<td>90</td>
</tr>
<tr>
<td>82</td>
<td>77</td>
<td>106</td>
<td>122</td>
<td>147</td>
<td>143</td>
<td>148</td>
<td>129</td>
<td>105</td>
<td>79</td>
</tr>
<tr>
<td>84</td>
<td>79</td>
<td>116</td>
<td>145</td>
<td>180</td>
<td>179</td>
<td>179</td>
<td>146</td>
<td>110</td>
<td>74</td>
</tr>
<tr>
<td>80</td>
<td>78</td>
<td>112</td>
<td>144</td>
<td>178</td>
<td>176</td>
<td>174</td>
<td>139</td>
<td>106</td>
<td>73</td>
</tr>
<tr>
<td>74</td>
<td>77</td>
<td>92</td>
<td>117</td>
<td>138</td>
<td>145</td>
<td>146</td>
<td>126</td>
<td>102</td>
<td>81</td>
</tr>
<tr>
<td>64</td>
<td>71</td>
<td>79</td>
<td>97</td>
<td>104</td>
<td>109</td>
<td>110</td>
<td>104</td>
<td>88</td>
<td>79</td>
</tr>
<tr>
<td>67</td>
<td>69</td>
<td>68</td>
<td>70</td>
<td>65</td>
<td>69</td>
<td>69</td>
<td>77</td>
<td>73</td>
<td>75</td>
</tr>
<tr>
<td>70</td>
<td>71</td>
<td>73</td>
<td>74</td>
<td>74</td>
<td>70</td>
<td>66</td>
<td>67</td>
<td>67</td>
<td>68</td>
</tr>
<tr>
<td>73</td>
<td>71</td>
<td>71</td>
<td>72</td>
<td>72</td>
<td>68</td>
<td>64</td>
<td>61</td>
<td>64</td>
<td>63</td>
</tr>
</tbody>
</table>
Declarations, main and inBound

```c
#include<stdio.h>
#include<stdbool.h>

#define M 10    // # rows
#define N 10    // # columns

int inBound(int i, int j);
int meanAroundPixel(int picture[][N], int i, int j);
void copy2dArray(int from[][N], int to[][N]);
void blurPicture(int picture[][N]);

int main() {
    ...
    int picture[M][N] = {...};
    blurPicture(picture);
    ...
    int inBound(int i, int j) { // is pixel inside picture?
        if((i < 0 || i >= M) || (j < 0 || j >= N))
            return false;
        return true;
    }
```
```c
int meanAroundPixel(int picture[][N], int i, int j) {
    int ii, jj, sum = 0, neighbors = 0;
    for(ii = i-1; ii <= i+1; ii++)
        for(jj = j-1; jj <= j+1; jj++)
            if(inBound(ii,jj)) {
                sum += picture[ii][jj];
                neighbors++;
            }
    return (int)((double)sum / neighbors + 0.5);
}

void blurPicture(int picture[][N]) {
    int i, j;
    int blurred[M][N];
    for(i = 0; i < M; i++)
        for(j = 0; j < N; j++)
            blurred[i][j] = meanAroundPixel(picture,i,j);
    copy2dArray(blurred, picture);
}

void copy2dArray(int from[][N], int to[][N]) {
    int i, j;
    for(i = 0; i < M; i++)
        for(j = 0; j < N; j++)
            to[i][j] = from[i][j];
}
```
Outline

❖ Modular Design
  ‣ Why modularity
  ‣ Functions

❖ Functions
  ‣ Declaration
  ‣ Definition
  ‣ Call

❖ Details
  ‣ Parameters
  ‣ Declaration
  ‣ Order

❖ Revisits
  ‣ gcd
  ‣ Newton Raphson
  ‣ Pi

❖ Mechanics
  ‣ Call
  ‣ Return
  ‣ Calling variables unchangeable

❖ Array parameters
  ‣ Elements
  ‣ Whole arrays
  ‣ 2-d arrays

❖ Libraries
  ‣ stdio.h
  ‣ math.h
  ‣ stdlib.h
  ‣ Your own

❖ Variable Storage classes
  ‣ Local
  ‣ static
  ‣ Global

❖ Recursion
In large programs, declarations provided in separate files

Called header files (.h)

Contain function declarations (rarely, also definitions)

C standard library defines several: stdio.h, stdlib.h, math.h

Included using #include directive

```
#include <stdio.h>  // declarations of input/output functions
#include <math.h>   // declarations of mathematical functions
```

Equivalent to typing the header file in that location
#include <stdio.h>

int scanf(...)  
  - Reads input  
  - Returns # values read

int printf(...)  
  - Prints output  
  - Returns # characters written

int getchar(void)  
  - Reads and returns one input character  
  - int, so it can hold EOF (end of file, next lecture)

int putchar(char)  
  - Writes one character  
  - If successful, returns printed char, otherwise returns EOF

int main() {  
  int a;  
  a=5;  
  return 0;  
}

#include <stdio.h>

int main() {  
  int a;  
  a=5;  
  printf("%d", a);  
  return 0;  
}

No stdio.h

stdio.h needed
> locate stdbool.h
/usr/include/stdbool.h
> more /usr/include/stdbool.h

```c
#ifndef __STDBOOL_H__
define __STDBOOL_H__

define __bool_true_false_are_defined 1

ifndef __cplusplus

define false 0
define true 1

define bool __Bool
#if __STDC_VERSION__ < 199901L && __GNUC__ < 3
typedef int __Bool;
#endif

endif /* !__cplusplus */

endif /* !__STDBOOL_H__ */
```

**stdio.h** similar but much more complex

ifndef.c
Redefining Macros

- Redefining a macro - compilation error
- May not know if header file included elsewhere
- Writing `#define` may cause error

- `#ifndef MACRO_NAME`
  - checks if `MACRO_NAME` defined
  - performs command only if macro not defined
  - till `#endif`

- `#undef MACRO_NAME`
  - Undefines `MACRO_NAME`
  - Can subsequently redefine

```c
#include <stdio.h>
#define A 1
// #define B 2
#ifndef A
#define A 2
#endif
#ifndef B
#define B 3
#endif

int main() {
    printf("A=%d\n", A);
    printf("B=%d\n", B);
    return 0;
}
```
#include <math.h>

Some systems: gcc -lm program.c

Rounding
- `double ceil(double)` - Ceiling
- `double floor(double)` - Floor
- `double round(double)` - Closest, half away from 0

Arithmetic
- `double sqrt(double)` - square root
- `double pow(double x, double y)` - $x^y$

Trigonometric
- `double cos(double)` - cosine
- `double sin(double)` - sine

Exponential
- `double exp(double x)` - $e^x$
- `double log(dbl.), log2(dbl.), log10(dbl.)` - logarithms
```c
#include <stdio.h>
#include <math.h>

int main() {
    double x, y;
    printf("x, y: ");
    scanf("%lf , %lf", &x, &y);
    printf("square root of %g: %g\n", x, sqrt(x));
    printf("round of %g: %g\n", y, round(y));
    printf("%g^%g: %g\n", x, y, pow(x,y));
    return 0;
}
```
#include <stdio.h>
#include <math.h>

int IsSquare(int);  
int IsPrime(int);

int main() {  
    int num;

    while (scanf("%d", &num) == 1) {  
        if (num < 0)  
            printf("%d is negative\n", num);

        else if (IsSquare(num))  
            printf("%d is a square\n", num);

        else if (IsPrime(num))  
            printf("%d is a prime\n", num);

        else  
            printf("%d is a regular Joe\n", num);

        return 0;
    }
    
    Wrote several versions

    int IsSquare(int n) {  
        int sq = (int) round(sqrt(n));  
        return (sq*sq == n);
    }

    int IsPrime(int n) {...}
Getting π via `math.h`

```c
#include <stdio.h>
#include <math.h>

int main() {
    printf("Pi from:\n ");
    printf("atan: %.15f\n", 4*atan(1.));
    printf("asin: %.15f\n", 2*asin(1.));

    // Following not part of standard, system dependent
    printf("definition: %.15f\n", M_PI);
    return 0;
}
```

How to find functions in `math.h`?

[Wiki](https://en.wikipedia.org/wiki/math.h)
# Lecture 7

## ECE15: Introduction to Computer Programming Using the C Language

### stdlib.h

- `#include <stdlib.h>`
- `int abs(int)` - absolute value
- `int exit(int)` - exists program (useful inside functions)
- `int rand()` - random integer
  - In range `[0,RAND_MAX]`, typically largest integer

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

int main() {
    printf("Maximum: %u\n", RAND_MAX);
    for (int i=0; i<5; i++)
        printf("%d\n", rand());
    return 0;
}
```

- Each number is a (complicated) function of previous - **pseudo random**
- Always same sequence
Changing the Random Sequence

- `#include <stdlib.h>`
- `void srand(unsigned int) - seeds rand`

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    srand(777);
    for (int i=0; i<5; i++)
    {
        printf("%d\n", rand());
    }
    return 0;
}
```

- Still always same sequence, but can change it by modifying seed
- How to get a different sequence each time?
- Seed with time (time in seconds since creation of Unix)

```c
srand(time(NULL));
```
How to roll 20 dice?

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    srand(time(NULL));
    for (int i=0; i<20; i++)
        printf("%d ", rand() % 6 + 1);
    printf("\n");
    return 0;
}
```
System Commands

- `#include <stdlib.h>`
- Can run system commands

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

int main() {
    system("pwd");
    system("ls");
    system("date");
    return 0;
}
```

- May be system dependent
#include <time.h>

time_t time() - time in seconds since ~1/1/1970 (beginning of Unix)
localtime(time_t) - converts time to structure with year, month, etc.
asctime() - converts time structure to ascii

```
#include <stdio.h>
#include <clock.h>

int main() {
    time_t t1, t2;
    printf("Type return to start"); getchar();
    t1=time(NULL);
    printf("Type return to stop"); getchar();
    t2=time(NULL);
    printf("Start: %d\nStop: %d\nElapsed: %d seconds\n", t1, t2, t2-t1);
    struct tm *local;
    local=localtime(&t2);
    printf("Local time and date: %s\n", asctime(local));
    return 0;
}
```
Nap Time

- `#include <unistd.h>` - unix standard library
- `sleep(int)` - sleep time in seconds

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    int seconds;
    printf("Seconds to nap: ");
    scanf("%d", &seconds);
    sleep(seconds);
    printf("\a\a\a");
    return 0;
}
```
Creating Your Own Library

- Store declarations and definitions in `library.h` (or relevant name)
- Begin program with `#include "library.h"`
- Searches for library in current directory
- `<file.h>` searches in C library directory
  - On mac: `/usr/include`

```
#include <stdio.h>
#include "functions.h"

int main() {
    printf("min(3,7)=%d\n", min(3,7));
    printf("max(3,7)=%d\n", max(3,7));
    return 0;
}
```

Typically definitions stored separately (next slide). This is just an example showing the “included” file is simply copied.

```
typically definitions stored separately (next slide). This
```

```
minmax.c
```

```
int min(int x, int y) {
    return x<y ? x : y;
}
int max(int x, int y) {
    return x>y ? x : y;
}
```

```
functions.h
```
Typical Large Programs

❖ *library.h* contains only *declarations* and *macros*

❖ *Definitions* stored separately in *library.c* or another *.c* file

```
#include <stdio.h>
#include "mm.h"

int min(int x, int y) {
    return x<y ? x : y;
}

int max(int x, int y) {
    return x>y ? x : y;
}

int main() {
    printf("min(3,7)=%d\n", min(3,7));
    printf("max(3,7)=%d\n", max(3,7));
    return 0;
}
```

❖ Compile

```
> gcc mm_main.c mm_fun.c
```

❖ Creates *a.out*
Separate Compilation and Linking

❖ To compile:  

> gcc part1.c part2.c creates a.out

❖ Large programs, many parts, don’t want to recompile all of them

❖ Create binary **object code** for each file, then **link** them

- part1.c ➝ part1.o
- part2.c ➝ part2.o
- part3.c ➝ part3.o

- source code ➝ object code ➝ executable ➝ a.out

❖ Create **object code** with `-c` option:

> gcc -c part1.c part2.c creates part1.o, part2.o

❖ To **link**, compile either .o or .c files

> gcc part1.o part2.o or > gcc part1.c part2.c creates a.out

❖ **Advantages:** Efficient, doesn’t re-compile all, can hide source code.

> gcc -c mm_main.c mm_fun.c
> gcc mm_main.o mm_fun.o
> gcc mm_fun.c mm_main.o

> gcc -c mm_main.c mm_fun.c
> gcc mm_main.o mm_fun.o
> gcc mm_fun.c mm_main.o

> gcc part1.o part2.o or > gcc part1.c part2.o creates a.out
Another Multiple-File Example

```c
#include <stdio.h>

part1.c
void p2();
int main() {
    printf("main calling p2\n");
    p2();
    printf("main exiting\n");
    return 0;
}

part2.c
#include <stdio.h>
void p3();
void p2() {
    printf("p2 calling p3\n");
    p3();
    printf("p2 returning to main\n");
}

part3.c
#include in all files where needed
void p3() {
    printf("p3 returning to p2\n");
}
```

#include in all files where needed

Declare fun. if not defined b4 call

```
> gcc part1.c part2.c part3.c

> gcc -c part1.c part3.c
> ls part*
part1.c part1.o part2.c part3.c part3.o
> gcc part1.o part2.c part3.o
```
Outline

❖ Modular Design
  ‣ Why modularity
  ‣ Functions

❖ Functions
  ‣ Declaration
  ‣ Definition
  ‣ Call

❖ Details
  ‣ Parameters
  ‣ Declaration
  ‣ Order

❖ Revisits
  ‣ gcd
  ‣ Newton Raphson
  ‣ Pi

❖ Mechanics
  ‣ Call
  ‣ Return
  ‣ Calling variables unchangeable

❖ Array Parameters
  ‣ Elements
  ‣ Whole arrays
  ‣ 2-d arrays

❖ Multiple files
  ‣ C standard library
  ‣ Your library
  ‣ Multiple files

❖ Variable Storage classes
  ‣ Local
  ‣ static
  ‣ Global

❖ Recursion
Storage Classes

Variable attributes

So far: **Type** (int, char, double)

Determine

- **Size** (# bytes) allocated
- **Interpretation** of binary string stored
- **Operations** permitted on variable

Next: **Storage class**

**Lifetime:** When memory is allocated (**variable is born**) and when released (**variable ceases to exist**) 

**Scope:** Subset of lifetime when variable is **visible** (accessible)

**Three storage classes**

- **local** - live & visible locally, those we’ve seen, most common
- **static** - live throughout program, visible locally
- **global** - live & visible throughout program
Local Variables

Most common storage class

**All variables we have seen were local**

Declared inside a block (several statements surrounded by `{ .. }`)

**Lifetime & scope:** From *declaration* till *exit from block*

```c
int main(void)
{
    int n=0;
    n++;
    int sqrt_n = sqrt(n);
    return 0;
}

int foo(double x) {
    int n;
    return x+n;
}

int main()
{
    int k=1;
    for (int i=0; i<3; i++) {
        double x=i+3;
        printf("%d %d %f", k, i, x);
    }
    double z=2;
    {
        double x = 5.0;
        printf("%d %f %f", k, z, x);
    }
    printf("%d %d %f", k, z);
}
```

**Remark:** Also called *automatic* variables, declaration can be preceded by keyword *auto* (seldom used)
What if a sub-block contains a variable of the same name as its parent block?

Inside the block, once the internal variable is declared, it gets used.

Original variable continues to exist, but is temporarily out of scope.

When internal block is exited, original variable returns to scope.

```
int main() {
  int a=1, b=2, c=3;
  {
    printf("a=%d b=%d c=%d",a,b,c);
    int a = 4;  double b = 5.0;
    a =  b;  c = 6;
    printf("a=%d b=%g c=%d",a,b,c);
    int  c=7;
    printf("a=%d b=%g c=%d",a,b,c);
  }
  printf("a=%d b=%d c=%d", a,b,c);
}
```
What does the following block print?

```
{ int a = 1, b = 2, c = 3;
  printf("%d %d %d\n",a,b,c);
  { int b = 4;
    double c = 5.0;
    printf("%d %d %f\n",a,b,c);
    a = b;
    { int c;
      c = b;
      printf("%d %d %d\n",a,b,c);
    }
    printf("%d %d %f\n",a,b,c);
  }
  printf("%d %d %d\n",a,b,c);
}
```
Local variables are actually stored in a **variable stack**. They are **pushed in** when they are born, and **popped out** when they die.

```c
{  int a = 1, b = 2, c = 3;
  printf("%d %d %d\n", a, b, c);
  {
    int b = 4;
    double c = 5.0;
    printf("%d %d %f\n", a, b, c);
    a = b;
    {
      int c;
      c = b;
      printf("%d %d %d\n", a, b, c);
    }
    printf("%d %d %f\n", a, b, c);
  }
  printf("%d %d %d\n", a, b, c);
}
```

![Variable Stack Diagram](image-url)
Static Variables

❖ Local variables that never die.

- **Scope:** same as local variables
- **Lifetime:** from allocation till end of program

❖ Declared **inside a block**

```c
static int n;
static double x = 0.5;
```

❖ Initialized **only once**, even if the block is entered many times

❖ Initialized to **zero by default**

Declaration, memory allocation, and initialization performed once, even if function called many times.

Example: **toggle a light**

```c
void toggle_light(void) {
    static int light = ON;
    if (light == OFF) {
        turn_light(ON);
        light = ON;
    } else {
        turn_light(OFF);
        light = OFF;
    }
}
```

#define OFF 0
#define ON 1
Two Applications

Count # times a function is called during program’s execution

```
#include <stdio.h>

void f(void) {
    static int count; // initialized to 0
    printf("%d ", ++count);
}

int main() {
    int i;
    for (i = 0; i < 5; i++)
        f();
    for (i = 0; i < 5; i++)
        f();
    return 0;
}
```

```
void g(void) {
    static bool first_time = true;
    if (first_time) {
        ...
        // initialize on first call
    }
    ...
    // after initialization
}
```
Global Variables

- **Declared** outside all blocks

  Lifetime: from allocation till end of program

  Scope: same - from allocation till end of program

- Here too scope can be **masked** by local variables of same name

- Can be used to communicate with functions, instead of parameters

- Initialized to 0

---

**Example**

```c
#include <stdio.h>

int x;

void f() {
    printf("%d\n", x+=2);
}

int main() {
    printf("%d\n", x+=5);
    f();
    printf("%d\n", x+=10);
}
```

globalExample.c
External Variables in Multiple Files

- Variables must be declared in files where used
- Local and static variables used in one function, hence in one file
- Global variable can be used in different functions, hence different files
  - If not defined in a file where used, must be declared there as `extern`
  - Can be defined only once, but can be declared (`extern`) many times

```c
#include <stdio.h>

int main() {
    f();
    printf("x=%d\n", x);
    return 0;
}

extern int x;

void f() {
    x++;
}
```

**What gets printed?**

1
What gets printed?

```c
#include <stdio.h>

int x = 20, y;

void f1(int x) {
    printf("x=%d y=%d\n",x,y);
    y = 11; x = 33;
}

void f2(int x) {
    printf("x=%d y=%d\n",x,y);
    x = 45; y = 15;
}

int main() {
    printf("x=%d y=%d\n",x,y);
    y = 5;
    f1(12);
    printf("x=%d y=%d\n",x,y);
    f2(y);
    x = 74;
    printf("x=%d y=%d\n",x,y);
    return 0;
}
```

Sometimes convenient
Confusing and risky!
Use sparingly

- `x=12 y=5`
- `x=11 y=11`
- `x=20 y=0`
- `x=20 y=11`
- `x=74 y=15`
Global variables convenient for debugging
Can print many variables from anywhere without passing all of them
Here `printAll` has a single parameter but prints several

```c
#include <stdio.h>
int i, val;
void printAll(char a[]) {
    printf("%s iteration=%d value=%d\n", a, i, val);
}
int main() {
    val=7;
    printAll("Before loop");
    for (i=0; i<5; i++) {
        val+=2;
        printAll("Inside loop");
    }
    printAll("After loop");
    return 0;
}
```

**Input:** Vertex coordinates of several triangles

**Output:** Area of each triangle, largest area, # multiplications

---

**Heron’s formula**

\[
\text{area} = \sqrt{p(p-a)(p-b)(p-c)}
\]

\[
p = \frac{a + b + c}{2}
\]
```c
#include <stdio.h>
#include <math.h>

int count; // count # multiplications, init 0

double distance(double x1, double y1, double x2, double y2) {
    count += 2;
    return sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
}

double heron(double a, double b, double c) {
    double p = (a + b + c)/2;
    count += 3;
    return sqrt(p * (p-a) * (p-b) * (p-c));
}

double tri_area(double x1, double y1, double x2, double y2, double x3, double y3) {
    double d12, d13, d23;
    d12 = distance(x1, y1, x2, y2);
    d13 = distance(x1, y1, x3, y3);
    d23 = distance(x2, y2, x3, y3);
    return heron(d12, d23, d13);
}
```

**Declarations and Functions**

Distance between \((x_1, y_1)\) and \((x_2, y_2)\)

Area of triangle whose sides have lengths \(a\), \(b\), \(c\)

Area of triangle whose vertices are \((x_1, y_1)\), \((x_2, y_2)\), and \((x_3, y_3)\)
```c
int main() {
    double x1, x2, x3, y1, y2, y3, max_area = 0;
    while (scanf("%lf%lf ...", &x1, &y1, &x2, &y2, &x3, &y3) == 6) {
        double area = tri_area(x1, y1, x2, y2, x3, y3);
        if (area > max_area)
            max_area = area;
        printf("Triangle (%f,%f) (%f,%f) (%f,%f) has area %f\n",
                x1, y1, x2, y2, x3, y3, area);
        printf("%d multiplications so far.\n", count);
    }
    printf("Largest area: %.2f\n\n", max_area);
    return 0;
}
```

Reading input from a file:

```
a.out < triangles.dat
```
## Summary of Storage Classes

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Static</th>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration</strong></td>
<td>int <code>sum = 0;</code> inside a block</td>
<td><code>static</code> int <code>sum;</code> inside a block</td>
<td>int <code>sum;</code> outside all blocks</td>
</tr>
<tr>
<td><strong>Lifetime</strong></td>
<td>To end of block</td>
<td>To end of program</td>
<td>To end of program</td>
</tr>
<tr>
<td><strong>Scope</strong></td>
<td>To end of block</td>
<td>To end of block</td>
<td>To end of program</td>
</tr>
<tr>
<td><strong>Initialization</strong></td>
<td>Not initialized</td>
<td>Once, to zero</td>
<td>Once, to zero</td>
</tr>
</tbody>
</table>

Can be masked
**Global & static variables initialized to 0**

**Local variables not initialized**

**Why?**

Efficiency!

Global and static created once.
Local may be created many times.

Want to initialize once?
Declare static!