**Lecture Outline**

❖ **Top-down Modular Design**

❖ **Functions**
  - Function definition
  - Calling functions
  - Returning values
  - Function declaration

❖ **Scope and Lifetime of Variables**
  - Local, global, and static variables
  - Automatic variable/function stack

❖ **Example: Area of Triangles**
All the C programs we have encountered so far were quite small, up to a dozen or two dozen lines of code at most...

On the other hand, complex tasks require large, complicated C programs, that often contain many thousands lines of C code.
Top-Down Modular Design

Complex tasks require large, complicated programs that contain thousands of lines of C code:

- How would you write such a program?
- How would you debug and/or modify it?
- How could you make it understandable?
- Many similar pieces of C code will repeat at various places; would be nice to reuse them!

Solution: Top-down modular design

The overall task is broken into smaller modules, constructed from even smaller reusable modules:

- Each part can be developed individually
- Much easier to debug, modify, and understand
- Can reuse similar code parts in many places
Problem: Need to prepare a meal of spaghetti bolognese. We 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

Top-down modular design is implemented in C using **functions**. In fact, every C program consists of a collection of functions. **main()** itself is a function, it is simply the function that is **executed first**. All other functions need to be **explicitly called**. Construction from functions:

- Makes **writing** and **modifying** code much easier
- Improves **readability** and facilitates **debugging**
- Avoids code **duplication** and enables code **reuse** (even code from other programs!)

"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."

A function can be **called** (by any other function) and **returns** a value. Before it can be called it should be **declared**. It must also be **defined**.
Lecture Outline

❖ Top-down Modular Design

❖ Functions
  ▸ Function definition
  ▸ Calling functions
  ▸ Returning values
  ▸ Function declaration

❖ Scope and Lifetime of Variables
  ▸ Local, global, and static variables
  ▸ Automatic variable/function stack

❖ Example: Area of Triangles
Function definitions in C consist of a **declaration** (which describes the *name* of the function, its *parameters*, and the *type of value* it returns), and the **function body**, which is a compound statement that implements the function.

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

- It is possible for a function to **have no parameters**. In this case, it can be declared like this:
  ```c
  int getchar(void)  
  int getchar()
  ```

- It is also possible for a function to **not return any value**. If so, it should be declared like this:
  ```c
  void print_result(int)
  ```
Example: Message Printing

The function `messages` facilitates and organizes the printing of program messages.

*It can be called from anywhere within the program, with a message index as a parameter.*

```c
#include <stdio.h>

void messages(int i);

int main()
{
  ...
  if (a < 0)
     messages(0);
  ...
}
```

```c
void messages(int i)
{
    switch (i)
    {
         case 0: printf("Input parameter error.\n");
            break;
...
        case 2: printf("Finished successfully.\n");
            break;
    default: printf("Invalid message.\n");
    }
}
```
Theorem: An integer $d$ divides both $a$ and $b$ if and only if it divides $\gcd(a,b)$.

Corollary: An integer $d$ divides all three of $a$, $b$, and $c$ if and only if it divides both $\gcd(a,b)$ and $c$.

Corollary: The largest integer $d$ that divides all three of $a$, $b$, and $c$ is the largest integer that divides both $\gcd(a,b)$ and $c$. That is $\gcd(a,b,c) = \gcd(\gcd(a,b), c)$.

Example: GCD of Three Integers

We have seen how to compute the greatest common divisor of two integers $\gcd(a,b)$ using the Euclidean algorithm. Here, we are required to compute the greatest common divisor $\gcd(a,b,c)$ of three integers $a$, $b$, and $c$. 
The following program implements the formula in C using the function \texttt{gcd(m,n)}:

```c
#include <stdio.h>

int gcd(int, int);

int main(void)
{
  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",g_abc);
  return 0;
}

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

**gcd(a,b,c) = gcd(gcd(a,b), c)**
For all \( n \): \( b_n < \pi < a_n \)
**Task:** Compute the area of a circle of radius \( r \). The function \texttt{mypi()} calculates \( \pi \) iteratively. It is called from \texttt{main()} to evaluate the circle area.

```c
#include <stdio.h>
#include <math.h>
double mypi(void);
int main()
{
  double r;
  printf("Enter the radius of the circle: ");
  scanf("%lf",&r);
  printf("The circle area is %20.15f\n", mypi()*r*r);
  return 0;
}

double mypi(void)
{
  double a,b;
  int i;
  a = 4;
  b = 2*sqrt(2);
  for (i = 0; i < 10; i++)
  {
    a = 2*a*b/(a+b);
    b = sqrt(a*b);
  }
  return a;
}
```

**Definition:**
- \texttt{mypi()} function
- Initialization of variables
- Iterative calculation of \( \pi \)
- Call to \texttt{mypi()} in \texttt{main()}

**Call:**
- \texttt{mypi()} is called from \texttt{main()}

**Declaration:**
- \texttt{double mypi(void);}
- \texttt{int main()}
- \texttt{printf()}
- \texttt{scanf()}
- \texttt{return 0;}

**Example:**
```c
#include <stdio.h>
#include <math.h>
double mypi(void)
{
  double a,b;
  int i;
  a = 4;
  b = 2*sqrt(2);
  for (i = 0; i < 10; i++)
  {
    a = 2*a*b/(a+b);
    b = sqrt(a*b);
  }
  return a;
}
```
Lecture Outline

❖ Top-down Modular Design

❖ Functions
  ‣ Function definition
  ‣ Calling functions
  ‣ Returning values
  ‣ Function declaration

❖ Scope and Lifetime of Variables
  ‣ Local, global, and static variables
  ‣ Automatic variable/function stack

❖ Example: Area of Triangles
What actually happens when this function is called?

- Memory is allocated to the function parameters `double x` and `double y`.
- The expressions `9.95*a` and `2/lambda` that produce the function arguments are evaluated. Order of the evaluation is unspecified.
- The results of these evaluations are assigned (physically copied into) the 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 is allocated to the variables declared inside the function body, and the function-body block is executed.
**Quiz A:** Consider the following program. *What does it print?*

```c
#include <stdio.h>

int f(int x);

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

int f(int x)
{
  return (++x);
}
```

**Quiz B:** *Why* `int x; scanf("%d", x)` *cannot possibly work?*
Consider the function `my_function` declared as:

```
double my_function(double x, double y);
```

and called as:

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

**What actually happens when this function returns?**

- If this function contains the statement(s) `return expression;` in the function body, execution of the function body stops as soon as the (first such) statement is reached.

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

- If a `return expression;` statement is never reached, execution stops with the closing brace of the function body, and `z` is assigned garbage.

- In both cases, all the memory allocated to the function parameters and local variables is released, and the program continues with `z++` etc.
Recall the function \( \text{gcd}(\text{int } m, \text{int } n) \) that computes the greatest common divisor of two integers using the Euclidean algorithm.

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

int foo(void) {
    ...
    d = gcd(9 - 1, 2);
    return gcd(5, d);
}
```
**Definition & Declaration Revisited**

**Recall:** Before any C function can be called, it has to be either defined or declared in the same file!

**What actually happens during function definition?**

- Memory is allocated for the function itself and for the value it returns, just as memory is allocated for (global) variables.

**What actually happens during function declaration?**

- **Nothing!** Function declaration simply informs the compiler that the definition of the function will be found elsewhere.
- Declaration also describes the type of the returned value (needed for automatic conversions), and the number and type of all function parameters.
- Therefore, function declarations can be imported from header files (for example, declarations of `printf` and `scanf` from `<stdio.h>`).

**Example:**

```c
long power(int, int);
long power(int base, int exponent);
```

Which is better?
**Recommended:** Declare all the functions, then define them all.

**Even better:** Declare the functions in a separate header file.

```
#include <...>
#include "minmax.h"

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

**Okay, but not recommended:** Define every function before calling it.

```
#include <...>

int max(int a, int b);  
int min(int a, int b); 

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

If max calls min and min calls max, this is not even possible!
Our Task: Write a program that reads integers from the user, and determines for each such integer whether it is a prime, a square, a negative number, or none of the above.

Example: Is It Prime, Square, Or ...?

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

enum bool {FALSE, TRUE};
typedef enum bool boolean;

boolean IsSquare(unsigned int);
boolean IsPrime(unsigned int);
double Round2Whole(double);
```
Implementation of `main()`, assuming that all the functions are already available:

```c
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;
    }
}
```
Example: Is It Prime, Square, Or ...?

Implementation of the function `IsPrime()`:

```c
boolean IsPrime(unsigned int n) {
    int i, sqrt_n;

    if (n == 2) return TRUE;

    if ((n % 2 == 0) || (n < 2))
        return FALSE;

    sqrt_n = (int) sqrt(n);

    for (i = 3; i <= sqrt_n; i += 2)
        if (n % i == 0)
            return FALSE;

    return TRUE;
}
```
Example: Is It Prime, Square, Or ...?

Implementation of the functions IsSquare() and Round2Whole():

```c
boolean IsSquare(unsigned int n)
{
    int sq = (int) Round2Whole(sqrt(n));
    return (sq*sq == n);
}

double Round2Whole(double frac)
{
    return floor(frac + 0.5);
}
```
#include <stdio.h>
#define MAX_ITERATIONS 10

double f(double x);
double fd(double x);
int main()
{
    double a;
    int i;

    scanf("%lf", &a);
    for (i = 0; i < MAX_ITERATIONS && (fd(a)! = 0); i++)
    {
        a = a - f(a)/fd(a);
        printf("Iteration: %d, Solution: %.12f\n", i, a);
    }
    printf("Solution is: %.12f\n", a);
    return 0;
}
Lecture Outline

❖ Top-down Modular Design

❖ Functions
  ‣ Function definition
  ‣ Calling functions
  ‣ Returning values
  ‣ Function declaration

❖ Scope and Lifetime of Variables
  ‣ Local, global, and static variables
  ‣ Automatic variable/function stack

❖ Example: Area of Triangles
In addition to its type, every variable in C also has another attribute: its storage class.

- **Scope:** Where in the file (more precisely, in the entire C program) the variable is visible (accessible) and where it is not visible.

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

What does the storage class of a variable determine?

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

**Scope:** Where in the file (more precisely, in the entire C program) the variable is visible (accessible) and where it is not visible.

What are the possible storage classes?

The main storage classes in C are local variables (auto), global variables (extern), and static variables (static).
Local Variables in C

Local variables are the most common storage class in C. A variable is local if it is declared inside a block (compound statement).

All the variables we have seen so far were local!

Lifetime of a local variable: From the time when it is declared (allocated) until exit from the block where it is declared.

```c
int main(void)
{
    int n;
    scanf("%d", &n);
    if (n < 0) return 1;
    int sqrt_n = sqrt(n);
        ...
    return 0;
}
```

Remark: Local variables are also called automatic, and their declaration can be preceded by the keyword auto (seldom used).

```c
{   int a = 1, b = 2, c = 3;
    printf("%3d %3d %3d\n",a,b,c);
    {    int b = 4; double c = 5.0; 
    a = b;   }
    printf("%3d %3d %3d\n",a,b,c);
}
```
The scope of a local variable is from the point in the program where it is declared until the end of the block where it is declared.

Scope is similar to lifetime, except...

- Scope extends to the end of the block, rather than till an exit from the block. Thus scope is unaffected by return, break, or other jumps.
- Scope can be masked. A variable becomes invisible in an internal sub-block if another variable is declared therein using the same identifier.

```c
int main(void) {
  int n;
  scanf("%d", &n);
  if (n < 0) return 1;
  int sqrt_n = sqrt(n);
     ...
  return 0;
}
```

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

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

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

---

**Variable Stack**

- `a`: 1
- `b`: 2
- `c`: 3
- `a`: 4
- `b`: 4
- `c`: 5.0
- `c`: 3
- `b`: 2
- `a`: 4

Local variables are actually stored in a **variable stack**. They are pushed in when they are born, and popped out when they die.
What are static variables?

- **Static variables are local variables that never die!**
  
  **scope = scope of local variables**
  
  **lifetime = from allocation till end of program**

- Static variables are declared **inside a block** as follows:

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

- Static variables are initialized only once, even in the block is entered many times. Uninitialized static variables are zero by default.

Example:

```c
void toggle_light(void)
{
    static int light = ON;
    if (light == OFF)
    {turn_light(ON); light = ON;}
    else {turn_light(OFF); light = OFF;}
}
```
#include <stdio.h>

void f(void)
{
    static int count = 1;
    printf("%d ", count++);
}

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

Useful feature: Static variables make it possible to count how many times a function is called during the execution of a program.

Useful feature: Static variables make it possible to initialize functions
What are global variables?

- Global variables are variables declared outside of any block:
  
  ```
  scope = from declaration till end of program
  lifetime = from allocation till end of program
  ```

- Scope can be masked by local variables with identical identifiers.
- Can be used to communicate with functions, instead of parameters.

Example:

```c
#include <stdio.h>
int x = 20, y;
void f1(int x);
void f2(int x);
int main()
{   y = 5; f1(12); f2(y);
  x = 74;
  printf("x=%d, y=%d\n",x,y);
  return 0;
}

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;
}
```
### Storage Classes: Summary

<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 sum = 0; inside a block</td>
<td>static int sum; inside a block</td>
<td>int sum = 0; outside a block</td>
</tr>
<tr>
<td><strong>Scope</strong></td>
<td>From declaration within own block</td>
<td>From declaration within own block</td>
<td>From declaration to program end</td>
</tr>
<tr>
<td><strong>Lifetime</strong></td>
<td>While the block executes</td>
<td>From allocation to end of program</td>
<td>Throughout the program</td>
</tr>
<tr>
<td><strong>Initialization</strong></td>
<td>Not initialized</td>
<td>Zero</td>
<td>Not initialized</td>
</tr>
</tbody>
</table>

- **Local**: Declaration and scope are within a block, lifetime is while the block executes, initialization is not initialized.
- **Static**: Declaration is within a block, scope is within own block, lifetime is from allocation to end of program, initialization is zero.
- **Global**: Declaration is outside a block, scope is from declaration to program end, lifetime is throughout the program, initialization is not initialized.
Top-down Modular Design

Functions
- Function definition
- Calling functions
- Returning values
- Function declaration

Scope and Lifetime of Variables
- Local, global, and static variables
- Automatic variable/function stack

Example: Area of Triangles
Example: Area of Triangles

**Input:** Vertex coordinates of several triangles

**Output:** Area of each triangle + the largest area

### Heron’s formula

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

\[
p = \frac{a + b + c}{2}
\]
Include all the header files, define all constants, declare all global variables (if any), then forward declarations of all functions:

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

FILE *input; /* pointer to the input data file */
int count = 0; /* count the number of multiplications */

double distance(double x1, double y1, double x2, double y2);
double heron(double a, double b, double c);
double tri_area(double x1, double y1, double x2, double y2,
                double x3, double y3);
```

**global variables**

**forward function declarations**
Implementation of `main()`

```c
int main(void)
{
    double x1, x2, x3, y1, y2, y3, max_area = 0;

    if ( (input = fopen("triangles.dat","r") ) == NULL)
    {
        printf("Unable to read input file triangles.dat!\n");
        return 1;
    }

    while (6 == fscanf(input,"%lf%lf ...",&x1,&y1,&x2,&y2,&x3,&y3))
    {
        double area = tri_area(x1, y1, x2, y2, x3, y3);
        if (area > max_area) max_area = area;
        printf("The area of triangle (%f,%f) (%f,%f) (%f,%f) is %f\n", x1, y1, x2, y2, x3, y3, area);
    }

    printf("There were %d multiplications so far.\n", count);

    printf("The largest triangle area is %.2f\n\n", max_area);
    return 0;
}
```

- `local to while loop`
- `global variable`
Implement all the functions, from the top down:

define 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);
}
Implement all the functions, from the top down:

```c
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));
}
```