Functions for Modular Programming

Objectives

At the end of the meeting, students should be able to:

- explain modular programming
- enumerate other built-in functions in C
- create their own functions

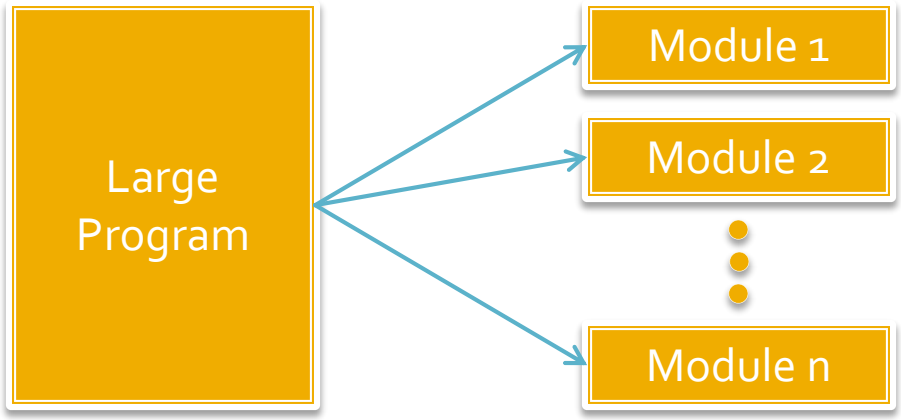
Modular Programming

- Programs in the real world can be extremely large (e.g., compilers and operating systems can consist of thousands of lines of code, often produced by teams of programmers)

Year	Operating System	SLOC(Million)
1993	Windows NT 3.1	4 – 5
1994	Windows NT 3.5	7 – 8
1996	Windows NT 4.0	11 - 12
2000	Windows 2000	more than 29
2001	Windows XP	40
2003	Windows Server 2003	50

Functions

- One way to manage large programming projects is to properly divide the task into small manageable modules known as functions.



A program is a collection of functions

- `main()` -- all programs basically have this function, and execution starts with this main() function
- a large program typically has many other functions which are invoked/called by the main() function or other functions

Predefined Functions

- Functions are not really new, we have been using many predefined functions such as
  - `printf()` -- for screen output
  - `scanf()` -- for keyboard input
  - `sqrt()` -- a math function for square root
  - `strcpy()` -- to assign a string to a string var

Predefined Functions

- Lots more can be used when we include stdio.h, math.h, string.h, assert.h, etc.

  -- have a look at the directory /usr/include/

Some useful predefined functions

- Math functions
  - `sqrt(x), sin(x), cos(x), …`
  - `rand()` // returns a pseudo-random int
  - `srand(seed)` // re-seeds the random generator
- String functions
  - `strcpy(x, y)` // copies string y to string x
  - `strlen(x)` // returns number of chars in x

Generating random numbers
(for games of chance and simulations)

```
main()
{
  int j;
  srand(1); /* change the seed */
            /* for a different sequence */
  for (j=0; j<200; j++) {
     printf("%d ", rand()%10 );
  } /* prints 200 random digits */
}
```

Creating and Calling your own Functions

- Function Heading (or Function Prototype)
  - How to use your function
- Function Definition
  - What your function is going to do
- Function Call
  - Executing your function

Defining a Function

SYNTAX:

```
[return_type] function_name(parameters)
{
    function body…
    ……
    [return return_value;]
}
```

Calling a Function

SYNTAX:

```
function_name(parameters);
```

Defining your Function
(without parameters)

```
#include<stdio.h>



main(){
}
```

void means that the function will not return anything

```
void sayMessage(){
    char name[20];
    printf("Hello What is your name?");
    scanf("%s",name);
    printf("%s is a nice name",name);
}
```

Calling your Function (without parameters)

```
#include<stdio.h>
// Function Heading
void sayMessage();

main(){
    sayMessage();
    sayMessage();
    sayMessage();
}
```

Function Calls

multiple calls may be made

```
//Function Definition
void sayMessage(){
    char name[20];
    printf("Hello What is your name?");
    scanf("%s",name);
    printf("%s is a nice name",name);
}
```

Defining your Function (with parameters)

```
#include<stdio.h>


main{
}
```

```
//Function Definition
int add ( int x, int y )
{
   int sum;
   sum = x + y;
   return sum;
}
```

Calling your Function (with parameters)

```
#include<stdio.h>
// Function Heading
int add(int, int);

main(){
    int a, b, sum, sum2;
    a=2;
    b=3;
    sum = add(1, 2);
    sum2 = add(a, b);
}
```

```
//Function Definition
int add ( int x, int y )
{
    int sum;
    sum = x + y;
    return sum;
}
```

Formal and Actual Parameters

```
#include<stdio.h>

int add(int, int);

main(){
    int a, b, ans, ans2;
    a=2;
    b=3;
    ans = add(1, 2);
    ans2 = add(a, b);
}
```

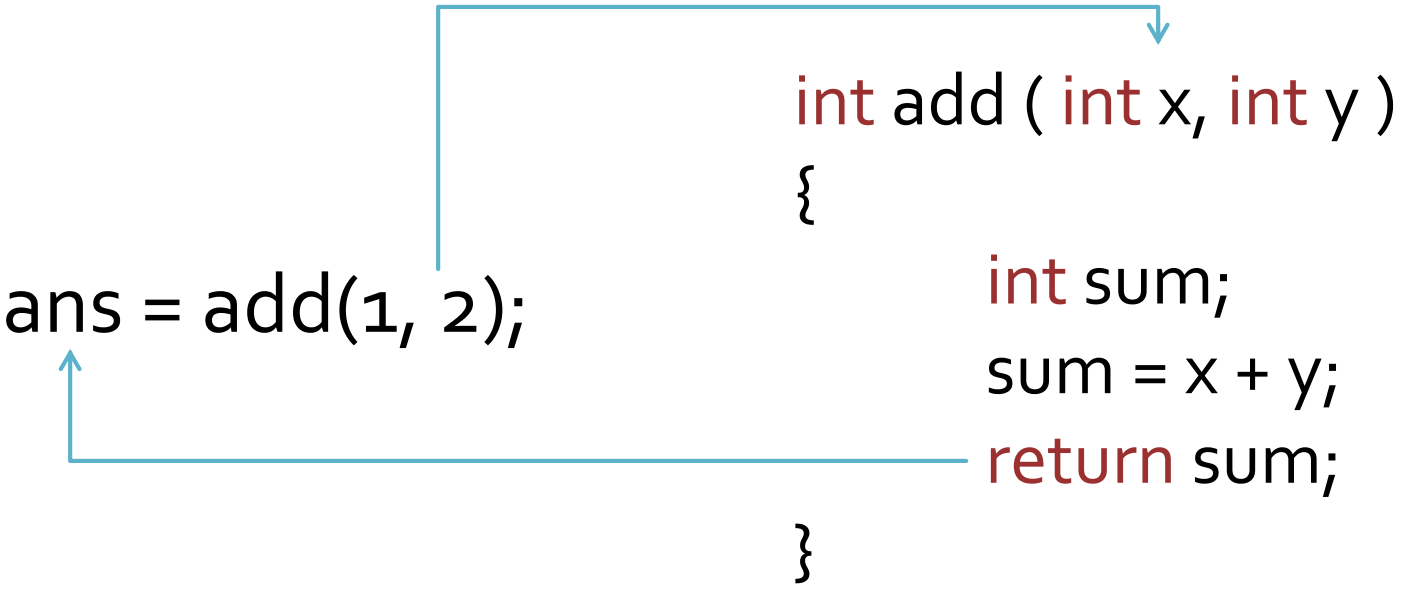
Actual Parameters

Formal Parameters and their types

```
int add ( int x, int y )
{
    int sum;
    sum = x + y;
    return sum;
}
```

More about Parameters and Return Statements

- Parameters serve as input to a function
- The output is returned using the return statement



More Examples

```
#include<stdio.h>
//Function Headings
float square(float x);
float cube(float x);

main(){
    float a, num2, num3;
    scanf("%d", &a);

    num2 = square(a);
    num3 = cube(a)
}
```

```
//Function Definitions
float square( float x )
{
    return x*x;
}

float cube( float x )
{
    return x*x*x;
}
```

More Examples

```
float max( float x, float y )
{  // find and return the larger value of 2 numbers
   if (x >= y) return x;
   else return y;
}

float maxof3 ( float x, float y, float z )
{  // find and return the largest among 3 numbers
   float temp = max( x, y );
   return max(temp, z);
}
```

Objectives

At the end of the meeting, students should be able to:

- explain what a pointer is
- use pointer operators in a program
- use pointers as function parameters; and
- explain the importance of modular programming

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void increase(int x);

main(){
    int age = 16;

    increase(age);
    printf("%d\n", age);
}

void increase(int x)
{
    x = x + 1;
}
```

Memory Address	Variable Name	Value
… 100 101 … 200 201 …	age	16

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void increase(int x);

main(){
    int age = 16;

    increase(age);
    printf("%d\n", age);
}

void increase(int x)
{
    x = x + 1;
}
```

Memory Address	Variable Name	Value
… 100 101 … 200 201 …	age	16

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void increase(int x);

main(){
    int age = 16;

    increase(age);
    printf("%d\n", age);
}

void increase(int x)
{
    x = x + 1;
}
```

Memory Address	Variable Name	Value
…		
100	age	16
101		
…		
200	x	16
201		
…		

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void increase(int x);

main(){
    int age = 16;

    increase(age);
    printf("%d\n", age);
}

void increase(int x)
{
    x = x + 1;
}
```

Memory Address	Variable Name	Value
…		
100	age	16
101		
…		
200	x	17
201		
…		

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void increase(int x);

main(){
    int age = 16;

    increase(age);
    printf("%d\n", age);
}

void increase(int x)
{
    x = x + 1;
}
```

Memory Address	Variable Name	Value
...		
100	age	16
101		
...		
200	x	17
201		
...		

value of age is **unchanged**, even after the function call to increase()

Pointers

- Pointers are variables that store memory addresses
- To declare a pointer, we place an * before the variable name (e.g. int *x)

Output parameters

- A parameter can be changed by passing the **address** of a variable (instead of its value)
  - Address Operator **&**
    - to obtain memory address of a variable
  - Indirection Operator ***
    - used to access the value of the variable pointed to by a pointer

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void increase(int *x);

main(){
    int age = 16;

    increase(&age);
    printf("%d\n", age);
}

void increase(int *x)
{
    *x = *x + 1;
}
```

Memory Address	Variable Name	Value
… 100 101 … 200 201 …	age	16

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void increase(int *x);

main(){
    int age = 16;

    increase(&age);
    printf("%d\n", age);
}

void increase(int *x)
{
    *x = *x + 1;
}
```

Memory Address	Variable Name	Value
… 100 101 … 200 201 …	age	16

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void increase(int *x);

main(){
    int age = 16;

    increase(&age);
    printf("%d\n", age);
}

void increase(int *x)
{
    *x = *x + 1;
}
```

Memory Address	Variable Name	Value
…		
100	age	16
101		
…		
200	x	100
201		
…		

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void increase(int *x);

main(){
    int age = 16;

    increase(&age);
    printf("%d\n", age);
}

void increase(int *x)
{
    *x = *x + 1;
}
```

Memory Address	Variable Name	Value
…		
100	age	17
101		
…		
200	x	100
201		
…		

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void increase(int *x);

main(){
    int age = 16;

    increase(&age);
    printf("%d\n", age);
}

void increase(int *x)
{
    *x = *x + 1;
}
```

Memory Address	Variable Name	Value
…		
100	age	17
101		
…		
200	x	100
201		
…		

value of age is **changed** after the function call to increase()

Example

- How do you swap the values of two numbers?

Example: swapping values

```
main()
{
    int a=1, b=2;
    printboth( a, b );
    swap( a, b );
    printboth( a, b );
}
```

```
void printboth( int x, int y )
{
    printf("%d and %d \n", x, y);
}

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

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void swap( int x, int y );

main(){
        int a=1, b=2;
        …
        swap( a, b );
        …
}

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

Memory Address	Variable Name	Value
…		
100	a	1
101	b	2
102		
…		
200		
201		
202		
…		

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void swap( int x, int y );

main(){
        int a=1, b=2;
        …
        swap( a, b );
        …
}

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

Memory Address	Variable Name	Value
…		
100	a	1
101	b	2
102		
…		
200		
201		
202		
…		

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void swap( int x, int y );

main(){
        int a=1, b=2;
        …
        swap( a, b );
        …
}

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

Memory Address	Variable Name	Value
…		
100	a	1
101	b	2
102		
…		
200	x	1
201	y	2
202		
…		

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void swap( int x, int y );

main(){
        int a=1, b=2;
        …
        swap( a, b );
        …
}

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

Memory Address	Variable Name	Value
…		
100	a	1
101	b	2
102		
…		
200	x	2
201	y	1
202		
…		

Example: swapping values

```
main()
{
    int a=1, b=2;
    printboth( a, b );
    swap( &a, &b );
    printboth( a, b );
}
```

```
printboth( int x, int y )
{
    printf("%d and %d \n", x, y);
}

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

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void swap(int *x, int *y);

main(){
        int a=1, b=2;
        …
        swap( &a, &b );
        …
}

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

Memory Address	Variable Name	Value
…		
100	a	1
101	b	2
102		
…		
200		
201		
202		
…		

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void swap(int *x, int *y);

main(){
        int a=1, b=2;
        …
        swap( &a, &b );
        …
}

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

Memory Address	Variable Name	Value
…		
100	a	1
101	b	2
102		
…		
200		
201		
202		
…		

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void swap(int *x, int *y);

main(){
        int a=1, b=2;
        …
        swap( &a, &b );
        …
}

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

Memory Address	Variable Name	Value
…		
100	a	1
101	b	2
102		
…		
200	x	100
201	y	101
202		
…		

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void swap(int *x, int *y);

main(){
        int a=1, b=2;
        …
        swap( &a, &b );
        …
}

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

Memory Address	Variable Name	Value
…		
100	a	1
101	b	2
102		
…		
200	x	100
201	y	101
202	temp	1
…		

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void swap(int *x, int *y);

main(){
        int a=1, b=2;
        …
        swap( &a, &b );
        …
}

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

Memory Address	Variable Name	Value
…		
100	a	2
101	b	2
102		
…		
200	x	100
201	y	101
202	temp	1
…		

More on parameter-passing (call-by-value)

```
#include<stdio.h>

void swap(int *x, int *y);

main(){
        int a=1, b=2;
        …
        swap( &a, &b );
        …
}

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

Memory Address	Variable Name	Value
…		
100	a	2
101	b	1
102		
…		
200	x	100
201	y	101
202	temp	1
…		

Example: swapping values

```
main()
{
    int a=1, b=2;
    printboth( a, b );
    swap( &a, &b );
    printboth( a, b );
}
```

```
printboth( int x, int y )
{
    printf("%d and %d \n", x, y);
}

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

Sorting any three numbers in ascending order

```
main()
{
    int a, b, c;
    printf("input any 3 numbers: ");
    scanf("%d %d %d", &a, &b, &c);
    if (a > b) swap(&a, &b);
    if (b > c) swap(&b, &c);
    if (a > b) swap(&a, &b);
    printf("sorted: %d %d %d\n", a, b, c);
}
```

Advantages of functions and modular programming

- **Avoid redundancy** – lengthy code that is repeated at different parts of a program need to be written only once
- **Encourage re-usability** – frequently-used functions can be added to a library (e.g., frequently used math or string functions)

Advantages of functions and modular programming

- **Improve readability** – implementation details are hidden in the functions
- **Manage complexity** – large software engineering projects are split into logical modules that can be developed and tested separately (simultaneous development is even possible with teams of programmers)

Good programming style with functions

- Use predefined functions when available (don't reinvent the wheel – except when you want to know how wheels are made)
- If you have to write your own function, consider using appropriate parameters to make it more useful