EECS 10: Computational Methods in Electrical and Computer Engineering

Review of Lectures 19 - 25

Rainer Dömer

doemer@uci.edu

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

Review of Lectures 19 - 25

- Lecture 19: Recursion
- Lecture 20: Structures, unions, enumerators
- Lecture 21: Binary data representation
- Lecture 22: Memory objects, pointers
- Lecture 23: Pointer operations, string operations
- Lecture 24: File processing
- Lecture 25: Program modules

Recursion

- Introduction
  - *Recursion* is an alternative to *Iteration*
  - Recursion is a very simple concept, yet very powerful
  - Recursion is present in nature
    - Trees have branches, which have branches, which have branches, ... which have leaves.
  - Recursion is traversal of hierarchy
    - *Traverse* (climb) a tree to the top:
      - start at the root
      - at a leaf, stop
      - at a branch, *traverse* one branch
    - *Traverse* a file system on a computer
      - start at the current directory
      - at a file, process the file
      - at a directory, *traverse* the directory

Recursion

- Recursive Function
  - Function that calls itself ...
    - ... directly, or
    - ... indirectly

```
int f(...)
  { ...
    f(...);
    ...
  }
```

```
int a(...)
  { ...
    b(...);
    ...
  }
int b(...)
  { ...
    a(...);
    ...
  }
```

- Concept of Recursion
  - Trivial *base case*
    - Return value defined for simple case
    - Example: **`if (arg == 0) {return 1; }`**
  - *Recursion step*
    - Reduce the problem towards the base case
    - Make a recursive function call
    - Example: **`if (arg > 0) { return ...fct(arg-1); }`**
- Termination of Recursion
  - Converging of recursive calls to the base case
  - Recursive call must be “simpler” than current call

Recursion

- Example: Factorial function *n!*
  - The factorial of a non-negative integer is
    - $n! = n * (n-1) * (n-2) * (n-3) * ... * 1$
  - This can be written as
    - $n! = n * ((n-1) * ((n-2) * ((n-3) * (... * 1))))$
  - *Recursive* definition:
    - $n=1: 1! = 1$ (base case)
    - $n>1: n! = n * (n-1)!$ (recursion step)
  - Example computation:

$$
\begin{aligned}
5! &= 5 * 4! \\
&= 5 * (4 * 3!) \\
&= 5 * (4 * (3 * 2!)) \\
&= 5 * (4 * (3 * (2 * 1!))) \\
&= 5 * (4 * (3 * (2 * 1))) \\
&= 5 * (4 * (3 * 2)) \\
&= 5 * (4 * 6) \\
&= 5 * 24 \\
&= 120
\end{aligned}
$$

Recursion

- Program example: **`Factorial.c`** (part 1/2)

```
/* Factorial.c: example demonstrating recursion */
/* author: Rainer Doemer                         */
/* modifications:                                */
/* 11/14/04 RD  initial version                  */

#include <stdio.h>

/* function definition */

long factorial(long n)
{
   if (n == 1)          /* base case */
      { return 1;
       } /* fi */
   else                 /* recursion step */
      { return n * factorial(n-1);
       } /* esle */
} /* end of factorial */

...
```

Recursion

- Program example: **`Factorial.c`** (part 2/2)

```
...
int main(void)
{
   /* variable definitions */
   long int n, f;

   /* input section */
   printf("Please enter value n: ");
   scanf("%ld", &n);

   /* computation section */
   f = factorial(n);

   /* output section */
   printf("The factorial of %ld is %ld.\n", n, f);

   /* exit */
   return 0;
} /* end of main */

/* EOF */
```


Recursion

- Example session: **`Factorial.c`**

```
% vi Factorial.c
% gcc Factorial.c -o Factorial -Wall -ansi
% Factorial
Please enter value n: 1
The factorial of 1 is 1.
% Factorial
Please enter value n: 2
The factorial of 2 is 2.
% Factorial
Please enter value n: 3
The factorial of 3 is 6.
% Factorial
Please enter value n: 5
The factorial of 5 is 120.
% Factorial
Please enter value n: 10
The factorial of 10 is 3628800.
%
```

Recursion vs. Iteration

- Example: Factorial function *n!*
  - The factorial of a non-negative integer is
    - $n! = n * (n-1) * (n-2) * (n-3) * ... * 1$
  - This can be written as
    - $n! = n * ((n-1) * ((n-2) * ((n-3) * (... * 1))))$
  - *Recursive* definition:
    - $n=1: 1! = 1$ (base case)
    - $n>1: n! = n * (n-1)!$ (recursion step)
  - *Iterative* implementation:
    - Compute *n* products in a loop
    - $n! = n * (n-1) * (n-2) * (n-3) * ... * 1$
    -
```
p = n;
for (f=n-1; f>=1; f--)
   { p = p * f; }
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 9

Recursion vs. Iteration

- Program example: **`Factorial2.c`** (part 1/2)

```
/* Factorial2.c: example demonstrating iteration       */
/* author: Rainer Doemer                               */
/* modifications:                                      */
/* 11/14/04 RD  initial version (based on Factorial.c) */

#include <stdio.h>

/* function definition */

long factorial(long n)
{
   long product, factor;

   product = n;
   for(factor = n-1; factor >=1; factor--)
      { product *= factor;
       } /* rof */
   return product;
} /* end of factorial */

...
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 10

Recursion vs. Iteration

- Program example: **`Factorial2.c`** (part 2/2)

```
...
int main(void)
{
   /* variable definitions */
   long int n, f;

   /* input section */
   printf("Please enter value n: ");
   scanf("%ld", &n);

   /* computation section */
   f = factorial(n);

   /* output section */
   printf("The factorial of %ld is %ld.\n", n, f);

   /* exit */
   return 0;
} /* end of main */

/* EOF */
```


Recursion vs. Iteration

- Example session: **`Factorial2.c`**

```
% cp Factorial.c Factorial2.c
% vi Factorial2.c
% gcc Factorial2.c -o Factorial2 -Wall -ansi
% Factorial2
Please enter value n: 1
The factorial of 1 is 1.
% Factorial2
Please enter value n: 2
The factorial of 2 is 2.
% Factorial2
Please enter value n: 3
The factorial of 3 is 6.
% Factorial2
Please enter value n: 5
The factorial of 5 is 120.
% Factorial2
Please enter value n: 10
The factorial of 10 is 3628800.
%
```

Recursion

- Example 2: Fibonacci series
  - Sequence of integers
    - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...
  - Mathematical properties
    - The first two numbers are 0 and 1
    - Every subsequent Fibonacci number is the sum of the previous two Fibonacci numbers
  - Ratio of successive Fibonacci numbers is ...
    - ... converging to constant value 1.618...
    - ... called *Golden Ratio* or *Golden Mean*
  - Recursive definition:
    - Base case: *fibonacci(0) = 0*
      fibonacci(1) = 1
    - Recursion step: *fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)*

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 13

Recursion

- Program example: **Fibonacci.c** (part 1/2)

```
/* Fibonacci.c: example demonstrating recursion */
/* author: Rainer Doemer                        */
/* modifications:                               */
/* 11/14/04 RD  initial version                 */

#include <stdio.h>

/* function definition */

long fibonacci(long n)
{
   if (n <= 1)  /* base case */
      { return n;
       } /* fi */
   else         /* recursion step */
      { return fibonacci(n-1) + fibonacci(n-2);
       } /* esle */
} /* end of fibonacci */

/* main function */
...
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 14

Recursion

- Program example: **`Fibonacci.c`** (part 2/2)

```
...
int main(void)
{
   /* variable definitions */
   long int n, f;

   /* input section */
   printf("Please enter value n: ");
   scanf("%ld", &n);

   /* computation section */
   f = fibonacci(n);

   /* output section */
   printf("The %ld-th Fibonacci number is %ld.\n", n, f);

   /* exit */
   return 0;
} /* end of main */

/* EOF */
```


Recursion

- Example session: **`Fibonacci.c`**

```
% cp Factorial.c Fibonacci.c
% vi Fibonacci.c
% gcc Fibonacci.c -o Fibonacci -Wall -ansi
% Fibonacci
Please enter value n: 1
The 1-th Fibonacci number is 1.
% Fibonacci
Please enter value n: 10
The 10-th Fibonacci number is 55.
% Fibonacci
Please enter value n: 20
The 20-th Fibonacci number is 6765.
% Fibonacci
Please enter value n: 30
The 30-th Fibonacci number is 832040.
% Fibonacci
Please enter value n: 40
The 40-th Fibonacci number is 102334155.
%
```

Data Structures

- Overview
  - Arrays
  - Structures
    - Declaration and definition
    - Instantiation and initialization
    - Member access
  - Unions
    - Declaration and definition
    - Member access
  - Enumerators
    - Declaration and definition
  - Type definitions

Data Structures

- Structures (aka. records)
  - User-defined, composite data type
    - Type is a composition of (different) sub-types
  - Fixed set of members
    - Names and types of members are fixed at structure definition
  - Member access by name
    - Member-access operator: ***structure_name.member_name***
- Example:

```
struct S { int i; float f;} s1, s2;

s1.i = 42;      /* access to members */
s1.f = 3.1415;
s2 = s1;        /* assignment */
s1.i = s1.i + 2*s2.i;
```

Data Structures

- Structure Declaration
  - Declaration of a user-defined data type

- Example:

```
struct Student;          /* declaration */
```


Data Structures

- Structure Declaration
  - Declaration of a user-defined data type
- Structure Definition
  - Definition of structure members and their type

- Example:

```
struct Student;          /* declaration */

struct Student           /* definition */
{ int   ID;              /* members */
  char  Name[40];
  char  Grade;
};
```

Data Structures

- Structure Declaration
  - Declaration of a user-defined data type
- Structure Definition
  - Definition of structure members and their type
- Structure Instantiation
  - Definition of a variable of structure type
- Example:

```
struct Student;           /* declaration */

struct Student            /* definition */
{ int   ID;               /* members */
  char  Name[40];
  char  Grade;
};

struct Student Jane;      /* instantiation */
```


Data Structures

- Structure Declaration
  - Declaration of a user-defined data type
- Structure Definition
  - Definition of structure members and their type
- Structure Instantiation and Initialization
  - Definition of a variable of structure type
  - Initializer list defines initial values of members
- Example:

```
struct Student;           /* declaration */

struct Student            /* definition */
{ int   ID;               /* members */
  char  Name[40];
  char  Grade;
};

struct Student Jane =     /* instantiation */
{1001, “Jane Doe”, ‘A’}; /* initialization */
```

Data Structures

- Structure Access
  - Members are accessed by their name
  - Member-access operator **.**
- Example:

```
struct Student
{  int  ID;
   char Name[40];
   char Grade;
};

struct Student Jane =
{1001, "Jane Doe", 'A'};

void PrintStudent(struct Student s)
{
   printf("ID:    %d\n", s.ID);
   printf("Name:  %s\n", s.Name);
   printf("Grade: %c\n", s.Grade);
}
```

Jane

ID	**1001**
Name	**"Jane Doe"**
Grade	**'A'**

```
ID:    1001
Name:  Jane Doe
Grade: A
```

EECS10: Computational Methods in ECE, Review 19-25   (c) 2004 R. Doemer   23

Data Structures

- Unions
  - User-defined, composite data type
    - Type is a composition of (different) sub-types
  - Fixed set of mutually exclusive members
    - Names and types of members are fixed at union definition
  - Member access by name
    - Member-access operator: ***union_name.member_name***
  - *Only one member may be used at a time!*
    - *All members share the same location in memory!*
- Example:

```
union U { int i; float f;} u1, u2;

u1.i = 42;      /* access to members */
u2.f = 3.1415;
u1.f = u2.f;    /* destroys u1.i! */
```

EECS10: Computational Methods in ECE, Review 19-25   (c) 2004 R. Doemer   24

Data Structures

- Union Declaration
  - Declaration of a user-defined data type
- Example:

```
union HeightOfTriangle;  /* declaration */
```


Data Structures

- Union Declaration
  - Declaration of a user-defined data type
- Union Definition
  - Definition of union members and their type
- Example:

```
union HeightOfTriangle;  /* declaration */

union HeightOfTriangle   /* definition */
{ int   Height;          /* members */
  int   LengthOfSideA;
  float AngleBeta;
};
```

Data Structures

- Union Declaration
  - Declaration of a user-defined data type
- Union Definition
  - Definition of union members and their type
- Union Instantiation
  - Definition of a variable of union type
- Example:

```
union HeightOfTriangle;   /* declaration */

union HeightOfTriangle    /* definition */
{ int   Height;           /* members */
  int   LengthOfSideA;
  float AngleBeta;
};

union HeightOfTriangle H;/* instantiation */
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 27

Data Structures

- Union Declaration
  - Declaration of a user-defined data type
- Union Definition
  - Definition of union members and their type
- Union Instantiation and Initialization
  - Definition of a variable of union type
  - *Single* initializer defines value of first member
- Example:

```
union HeightOfTriangle;   /* declaration */

union HeightOfTriangle    /* definition */
{ int   Height;           /* members */
  int   LengthOfSideA;
  float AngleBeta;
};

union HeightOfTriangle H /* instantiation */
= { 42 };                /* initialization */
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 28

Data Structures

- Union Access
  - Members are accessed by their name
  - Member-access operator `.`
- Example:

```
union HeightOfTriangle
{ int   Height;
  int   SideA;
  float Beta;
};

union HeightOfTriangle t1, t2, t3
= { 42 };

void SetHeight(void)
{
   t1.Height = 10;
   t2.SideA = t1.Height / 2;
   t3.Beta = 90.0;
}
```

	t1
Height/ SideA/ Beta	10

	t2
Height/ SideA/ Beta	5

	t3
Height/ SideA/ Beta	90.0

Data Structures

- Enumerators
  - User-defined data type
    - Members are an enumeration of integral constants
  - Fixed set of members
    - Names and values of members are fixed at enumerator definition
  - Members are constants
    - Member values cannot be changed after definition
- Example:

```
enum E { red, yellow, green };
enum E LightNS, LightEW;

LightEW = green;        /* assignment */
if (LightNS == green)   /* comparison */
   { LightEW = red; }
```

Data Structures

- Enumerator Declaration
  - Declaration of a user-defined data type
- Example:

```
enum Weekday;             /* declaration */
```


Data Structures

- Enumerator Declaration
  - Declaration of a user-defined data type
- Enumerator Definition
  - Definition of enumerator members and their value
- Example:

```
enum Weekday;             /* declaration */

enum Weekday              /* definition */
{ Monday, Tuesday,        /* members */
  Wednesday, Thursday,
  Friday, Saturday, Sunday;
};
```

Data Structures

- Enumerator Declaration
  - Declaration of a user-defined data type
- Enumerator Definition
  - Definition of enumerator members and their value
- Enumerator Instantiation
  - Definition of a variable of enumerator type
- Example:

```
enum Weekday;              /* declaration */

enum Weekday               /* definition */
{ Monday, Tuesday,         /* members */
  Wednesday, Thursday,
  Friday, Saturday, Sunday;
};

enum Weekday Today;        /* instantiation */
```


Data Structures

- Enumerator Declaration
  - Declaration of a user-defined data type
- Enumerator Definition
  - Definition of enumerator members and their value
- Enumerator Instantiation and Initialization
  - Definition of a variable of enumerator type
  - Initializer should be one member of the enumerator
- Example:

```
enum Weekday;              /* declaration */

enum Weekday               /* definition */
{ Monday, Tuesday,         /* members */
  Wednesday, Thursday,
  Friday, Saturday, Sunday;
};

enum Weekday Today         /* instantiation */
= Wednesday;               /* initialization */
```

Data Structures

- Enumerator Values
  - Enumerator values are integer constants
  - By default, enumerator values start at 0 and are incremented by 1 for each following member
- Example:

Today

Wednesday

```
Day of week: 2
```

```
enum Weekday
{ Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday;
};
enum Weekday Today
= Wednesday;
void PrintWeekday(
     enum Weekday d)
{
  printf("Day: %d\n", d);
}
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 35

Data Structures

- Enumerator Values
  - Enumerator values are integer constants
  - By default, enumerator values start at 0 and are incremented by 1 for each following member
  - Specific enumerator values may be defined by the user
- Example:

Today

Wednesday

```
Day of week: 3
```

```
enum Weekday
{ Monday = 1,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday;
};
enum Weekday Today
= Wednesday;
void PrintWeekday(
     enum Weekday d)
{
  printf("Day: %d\n", d);
}
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 36

Data Structures

- Enumerator Values
  - Enumerator values are integer constants
  - By default, enumerator values start at 0 and are incremented by 1 for each following member
  - Specific enumerator values may be defined by the user
- Example:

Today

Wednesday

Day of week: 4

```
enum Weekday
{ Monday = 2,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday = 1;
};
enum Weekday Today
= Wednesday;
void PrintWeekday(
     enum Weekday d)
{
  printf("Day: %d\n", d);
}
```

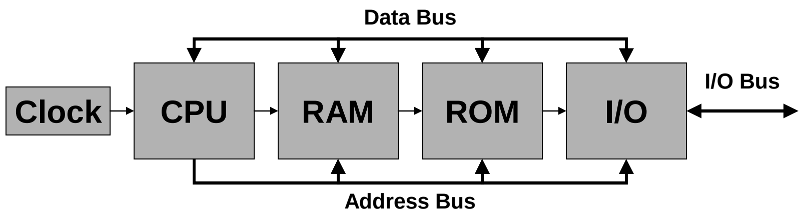

Data Structures

- Type definitions
  - A *typedef* can be defined as an alias type for another type
  - A *typedef* definition follows the same rules as a variable definition
  - Type definitions are usually used to abbreviate access to user-defined types
- Examples:

```
typedef long MyInteger;

typedef enum Weekday Day;
Day Today;

typedef struct Student Scholar;
Scholar Jane, John;
```

Basic Computer Architecture

- Main Computer Components
  – Central Processing Unit (CPU)
    - e.g. Intel Pentium, Motorola PowerPC, Sun SPARC, ...
  – Random Access Memory (RAM)
    - storage for program and data, read and write access
  – Read Only Memory (ROM)
    - fixed storage for basic input/output system (BIOS)
  – I/O Units
    - Input/output units connecting to peripherals



EECS10: Computational Methods in ECE, Review 19-25   (c) 2004 R. Doemer   39

Binary Data Representation

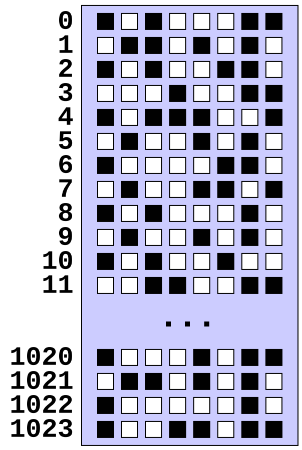
- Programs and data in a computer are represented in binary format
  – 1 *bit* (binary digit), 2 possible values   ■ □
    - 0 (false, “no”, current off, “empty”, ...)
    - 1 (true, “yes”, current on, “solid”, ...)
  – 1 *byte* = 8 bits (2^8 = 256 values)   ■□■□□□■□
    - in C, type **`char`** equals one byte
  – 1 *word* = 4* bytes (2^{32} = 4294967296 values)   ■□■□□□■■ □■■□■□■□ ■□■□□■■□ ■□■■□□■■
    - in C, type **`int`** equals one word
- Memory size is measured in Bytes
  – 1 KB = 1024 byte = 1 “kilo byte”
  – 1 MB = 1024*1024 byte = 1 “mega byte”
  – 1 GB = 1024*1024*1024 byte = 1 “giga byte”

*(*architecture dependent!)*

EECS10: Computational Methods in ECE, Review 19-25   (c) 2004 R. Doemer   40

Binary Data Representation

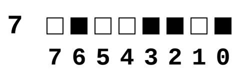
- Memory is composed of addressable bytes
  - Example:
    1 KB of memory



EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 41

Binary Data Representation

- Memory is composed of addressable bytes
  - Example:
    1 KB of memory
  - What is the value at address 7?

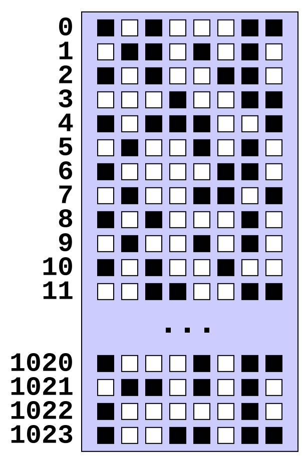


$= 0*2^7 + 1*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$

$= 0*128 + 1*64 + 0*32 + 0*16 + 1*8 + 1*4 + 0*2 + 1*1$

$= 64 + 8 + 4 + 1$

$= 77$



EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 42

Binary Data Representation

- Number Systems
  - DEC: Decimal numbers
    - Base 10, digits 0, 1, 2, 3, ..., 9
    - e.g. $157 = 1*10^2 + 5*10^1 + 7*10^0$
  - BIN: Binary numbers
    - Base 2, digits 0, 1
    - e.g. $10011101_2 = 1*2^7 + 0*2^6 + ... + 1*2^0$
  - OCT: Octal numbers
    - Base 8, digits 0, 1, 2, 3, ..., 7
    - e.g. $235_8 = 2*8^2 + 3*8^1 + 5*8^0$
  - HEX: Hexadecimal numbers
    - Base 16, digits 0, 1, 2, 3, ..., 9, A, B, C, ..., F
    - e.g. $9D_{16} = 9*16^1 + 13*16^0$

Binary Data Representation

- Number Systems

DEC	BIN	OCT	HEX
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Binary Data Representation

- Number Systems (signed vs. unsigned)

SDEC	UDEC	BIN	OCT	HEX
0	0	0000	0	0
1	1	0001	1	1
2	2	0010	2	2
3	3	0011	3	3
4	4	0100	4	4
5	5	0101	5	5
6	6	0110	6	6
7	7	0111	7	7
-8	8	1000	10	8
-7	9	1001	11	9
-6	10	1010	12	A
-5	11	1011	13	B
-4	12	1100	14	C
-3	13	1101	15	D
-2	14	1110	16	E
-1	15	1111	17	F

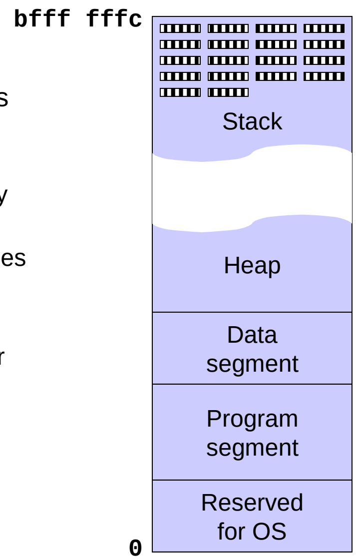
Binary Data Representation

- Number Systems
  - Signed representation: *two's complement*
    - to obtain the negative of any number in binary representation, ...
      - ... invert all bits,
      - ... and add 1
  - Example: 4-bit two's complement

SDEC	UDEC	BIN	OCT	HEX
...	...	...	...	...
7	7	0111	7	7
-8	8	1000	10	8
-7	9	1001	11	9
...	...	...	...	...

Binary Data Representation

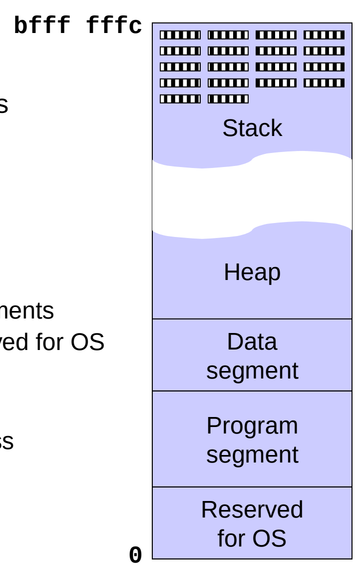
- Memory Segmentation
  - typical (virtual) memory layout on processor with 4-byte words and 1 GB of memory
  - Stack
    - grows and shrinks dynamically
    - function call hierarchy
    - stack frames with local variables
  - Heap
    - “free” storage
    - dynamic allocation by the user
  - Data segment
    - global (and static) variables
  - Program segment
    - stores binary program code



EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 47

Binary Data Representation

- Memory Segmentation
  - typical (virtual) memory layout on processor with 4-byte words and 1 GB of memory
- Memory errors
  - *Out of memory*
    - Stack and heap collide
  - *Segmentation fault*
    - access outside allocated segments
    - e.g. access to segment reserved for OS
  - *Bus error*
    - mis-aligned word access
    - e.g. word access to an address that is not divisible by 4



EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 48

Objects in Memory

- Data in memory is organized as objects
- Every object has ...
  - ... a *type* (e.g. **`int`**, **`double`**, **`char[ ]`**)
    - type is known to the compiler at compile time
  - ... a *value* (e.g. **`42`**, **`3.1415`**, **`“text”`**)
    - value is used for computation of expressions
  - ... a *size* (number of bytes in the memory)
    - in C, the **`sizeof`** operator returns the size of a variable or type
  - ... a *location* (address in the memory)
    - in C, the “address-of” operator (**&**) returns the address of an object
- Variables ...
  - ... serve as identifiers for objects
  - ... are bound to objects
  - ... give objects a name

Objects in Memory

- Example: Variable values, addresses, and sizes

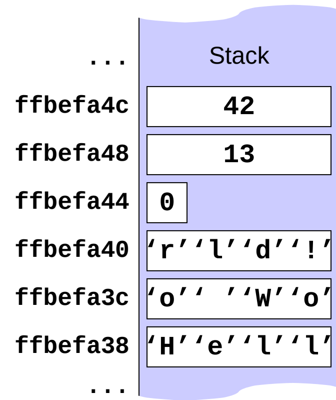
```
int x = 42;
int y = 13;
char s[] = "Hello World!";

printf("Value   of x    is %d.\n", x);
printf("Address of x    is %p.\n", &x);
printf("Size    of x    is %u.\n", sizeof(x));
printf("Value   of y    is %d.\n", y);
printf("Address of y    is %p.\n", &y);
printf("Size    of y    is %u.\n", sizeof(y));
printf("Value   of s    is %s.\n", s);
printf("Address of s    is %p.\n", &s);
printf("Size    of s    is %u.\n", sizeof(s));
printf("Value   of s[1] is %c.\n", s[1]);
printf("Address of s[1] is %p.\n", &s[1]);
printf("Size    of s[1] is %u.\n", sizeof(s[1]));
```

Objects in Memory

- Example: Variable values, addresses, and sizes

```
int x = 42;
int y = 13;
char s[] = "Hello World!";
...
```

```
Value   of x    is 42.
Address of x    is ffbefa4c.
Size    of x    is 4.
Value   of y    is 13.
Address of y    is ffbefa48.
Size    of y    is 4.
Value   of s    is Hello World!.
Address of s    is ffbefa38.
Size    of s    is 13.
Value   of s[1] is e.
Address of s[1] is ffbefa39.
Size    of s[1] is 1.
```



Pointers

- *Pointers* are variables whose values are *addresses*
  - The "address-of" operator (**&**) returns a pointer!
- Pointer Definition
  - The unary ***** operator indicates a pointer type in a definition

```
int x = 42;    /* regular integer variable */
int *p;        /* pointer to an integer */
```

- Pointer initialization or assignment
  - A pointer may be set to the "address-of" another variable

```
p = &x;        /* p points to x */
```

  - A pointer may be set to **0** (points to no object)

```
p = 0;         /* p points to no object */
```

  - A pointer may be set to **NULL** (points to "NULL" object)

```
#include <stdio.h>   /* defines NULL as 0 */
p = NULL;      /* p points to no object */
```

Pointers

- Pointer Dereferencing
  - The unary ***** operator dereferences a pointer to the value it points to ("content-of" operator)

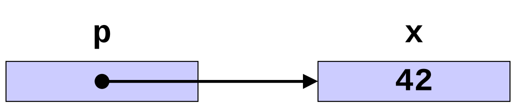
```
#include <stdio.h>

int  x = 42;   /* regular integer variable */
int *p = NULL; /* pointer to an integer */
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 53

Pointers

- Pointer Dereferencing
  - The unary ***** operator dereferences a pointer to the value it points to ("content-of" operator)

```
#include <stdio.h>

int  x = 42;   /* regular integer variable */
int *p = NULL; /* pointer to an integer */

p = &x;        /* make p point to x */
```



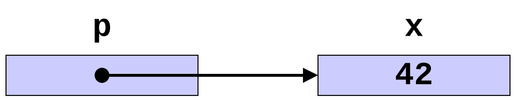
EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 54

Pointers

- Pointer Dereferencing
  - The unary ***** operator dereferences a pointer to the value it points to (“content-of” operator)

```
#include <stdio.h>

int  x = 42;   /* regular integer variable */
int *p = NULL; /* pointer to an integer */

p = &x;        /* make p point to x */
printf(“x is %d, content of p is %d\n”, x, *p);
```

```
x is 42, content of p is 42
```

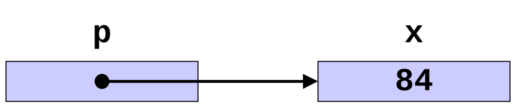


Pointers

- Pointer Dereferencing
  - The unary ***** operator dereferences a pointer to the value it points to (“content-of” operator)

```
#include <stdio.h>

int  x = 42;   /* regular integer variable */
int *p = NULL; /* pointer to an integer */

p = &x;        /* make p point to x */
printf(“x is %d, content of p is %d\n”, x, *p);
*p = 2 * *p;   /* multiply content of p by 2 */
printf(“x is %d, content of p is %d\n”, x, *p);
```

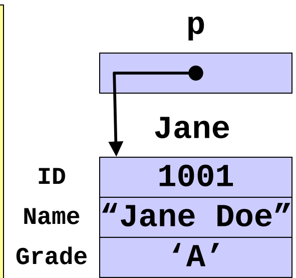
```
x is 42, content of p is 42
x is 84, content of p is 84
```



Pointers

- Pointer Dereferencing
  - The **->** operator dereferences a pointer to a structure to the content of a structure member

```
struct Student
{  int  ID;
   char Name[40];
   char Grade;
};

struct Student Jane =
{1001, “Jane Doe”, ‘A’};

struct Student *p = &Jane;

void PrintStudent(void)
{
   printf(“ID:    %d\n”, p->ID);
   printf(“Name:  %s\n”, p->Name);
   printf(“Grade: %c\n”, p->Grade);
}
```




```
ID:    1001
Name:  Jane Doe
Grade: A
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 57

Pointers

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element

```
int x[5] = {10,20,30,40,50}; /* array of 5 integers */
int *p;                      /* pointer to integer */

p = &x[1];                   /* point p to x[1] */
printf(“%d, ”, *p);          /* print content of p */
```

```
20,
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 58

Pointers

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element

```
int x[5] = {10,20,30,40,50}; /* array of 5 integers */
int *p;                      /* pointer to integer */
p = &x[1];                   /* point p to x[1] */
printf("%d, ", *p);          /* print content of p */
p++;                         /* increment p by 1 */
printf("%d, ", *p);          /* print content of p */
```

```
20, 30,
```


Pointers

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element

```
int x[5] = {10,20,30,40,50}; /* array of 5 integers */
int *p;                      /* pointer to integer */
p = &x[1];                   /* point p to x[1] */
printf("%d, ", *p);          /* print content of p */
p++;                         /* increment p by 1 */
printf("%d, ", *p);          /* print content of p */
p--;                         /* decrement p by 1 */
printf("%d, ", *p);          /* print content of p */
```

```
20, 30, 20,
```

Pointers

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element

```
int x[5] = {10,20,30,40,50}; /* array of 5 integers */
int *p;                      /* pointer to integer */

p = &x[1];                   /* point p to x[1] */
printf("%d, ", *p);          /* print content of p */
p++;                         /* increment p by 1 */
printf("%d, ", *p);          /* print content of p */
p--;                         /* decrement p by 1 */
printf("%d, ", *p);          /* print content of p */
p += 2;                      /* increment p by 2 */
printf("%d, ", *p);          /* print content of p */
```

```
20, 30, 20, 40,
```


Pointers

- Pointer Comparison
  - Pointers may be compared for equality
    - operators **==** and **!=** are useful to determine *identity*
    - operators **<**, **<=**, **>=**, and **>** are *not* applicable

```
int x[5] = {10,20,10,20,10}; /* array of 5 integers */
int *p1, *p2;                /* pointers to integer */

p1 = &x[1]; p2 = &x[3];      /* point to x[1], x[3] */

if (p1 == p2)
  { printf("p1 and p2 are identical!\n");
   }
if (*p1 == *p2)
  { printf("Contents of p1 and p2 are same!\n");
   }
```

```
Contents of p1 and p2 are same!
```

Pointers

- Pointer Comparison
  - Pointers may be compared for equality
    - operators **==** and **!=** are useful to determine *identity*
    - operators **<**, **<=**, **>=**, and **>** are *not* applicable

```
int x[5] = {10,20,10,20,10}; /* array of 5 integers */
int *p1, *p2;                /* pointers to integer */

p1 = &x[1]; p2 = &x[3];      /* point to x[1], x[3] */
p1 += 2;                     /* increment p1 by 2 */
if (p1 == p2)
  { printf("p1 and p2 are identical!\n");
   }
if (*p1 == *p2)
  { printf("Contents of p1 and p2 are same!\n");
   }
```

```
p1 and p2 are identical!
Contents of p1 and p2 are same!
```


Pointers

- String Operations using Pointers
  - Example: String length

```
int Length(char *s)
{
   int l = 0;
   char *p = s;

   while(*p != 0)
   { p++;
     l++;
   }
   return l;
}
```

```
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("Length of %s is %d\n",
            s1, Length(&s1[0]));
printf("Length of %s is %d\n",
            s2, Length(&s2[0]));
```

```
Length of ABC is 3
Length of Hello World! is 12
```

Pointers

- String Operations using Pointers
  - Example: String length

```
int Length(char *s)
{
   int l = 0;
   char *p = s;

   while(*p != 0)
   { p++;
     l++;
   }
   return l;
}
```

```
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("Length of %s is %d\n",
            s1, Length(&s1[0]));
printf("Length of %s is %d\n",
            s2, Length(s2));
```

```
Length of ABC is 3
Length of Hello World! is 12
```

  - Array and pointer types are equivalent
    - **s2** is an array, but can be passed as a pointer argument
    - Character array **s2** is same as character pointer **&s2[0]**

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 65

Pointers

- String Operations using Pointers
  - Example: String length

```
int Length(char *s)
{
   int l = 0;
   char *p = s;

   while(*p != 0)
   { p++;
     l++;
   }
   return l;
}
```

```
char s1[] = "ABC";
char *s2  = "Hello World!";

printf("Length of %s is %d\n",
            s1, Length(s1));
printf("Length of %s is %d\n",
            s2, Length(s2));
```

```
Length of ABC is 3
Length of Hello World! is 12
```

  - Array and pointer types are equivalent
    - **s1** is an array of characters, **s2** is a pointer to character
    - Both **s1** and **s2** can be passed to character pointer **s**

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 66

Pointers

- String Operations using Pointers
  - Example: String length

```
int Length(char s[])
{
   int l = 0;
   char *p = s;

   while(*p != 0)
   { p++;
     l++;
   }
   return l;
}
```

```
char s1[] = "ABC";
char *s2  = "Hello World!";

printf("Length of %s is %d\n",
            s1, Length(s1));
printf("Length of %s is %d\n",
            s2, Length(s2));
```

```
Length of ABC is 3
Length of Hello World! is 12
```

  - Array and pointer types are equivalent
    - **s1** is an array of characters, **s2** is a pointer to character
    - Both **s1** and **s2** can be passed to character array **s**

Pointers

- String Operations using Pointers
  - Example: String copy

```
void Copy(
     char *Dst,
     char *Src)
{
 do{
     *Dst = *Src;
     Dst++;
   } while(*Src++);
}
```

```
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("s1 is %s, s2 is %s\n",
               s1, s2);
```

```
s1 is ABC, s2 is Hello World!
```

Pointers

- String Operations using Pointers
  - Example: String copy

```
void Copy(
     char *Dst,
     char *Src)
{
 do{
     *Dst = *Src;
     Dst++;
   } while(*Src++);
}
```

```
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("s1 is %s, s2 is %s\n",
               s1, s2);
Copy(s2, s1);
printf("s1 is %s, s2 is %s\n",
               s1, s2);
```

```
s1 is ABC, s2 is Hello World!
s1 is ABC, s2 is ABC
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 69

Pointers

- String Operations using Pointers
  - Example: String copy

```
void Copy(
     char *Dst,
     char *Src)
{
 do{
     *Dst = *Src;
     Dst++;
   } while(*Src++);
}
```

```
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("s1 is %s, s2 is %s\n",
               s1, s2);
Copy(s2, s1);
printf("s1 is %s, s2 is %s\n",
               s1, s2);
```

```
s1 is ABC, s2 is Hello World!
s1 is ABC, s2 is ABC
```


  - Passing pointers as arguments to functions
    - Function can modify caller data by pointer dereferencing
    - Passing pointers = Pass by reference!

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 70

Pointers

- String Operations using Pointers
  - Example: String copy

```
void Copy(
     char *Dst,
     const char *Src)
{
 do{
     *Dst = *Src;
     Dst++;
   } while(*Src++);
}
```

```
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("s1 is %s, s2 is %s\n",
               s1, s2);
Copy(s2, s1);
printf("s1 is %s, s2 is %s\n",
               s1, s2);
```

```
s1 is ABC, s2 is Hello World!
s1 is ABC, s2 is ABC
```

  - Passing pointers as arguments to functions
    - Function can modify caller data by pointer dereferencing
    - Type qualifier **const**:
      Modification by pointer derefencing *not* allowed!

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 71

Pointers

- String Operations using Pointers
  - Example: String copy

```
void Copy(
     const char *Dst,
     const char *Src)
{
 do{
     *Dst = *Src;
     Dst++;
   } while(*Src++);
```

Error!
Write access to **const** data!

```
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("s1 is %s, s2 is %s\n",
               s1, s2);
Copy(s2, s1);
printf("s1 is %s, s2 is %s\n",
               s1, s2);
```

```
s1 is ABC, s2 is Hello World!
s1 is ABC, s2 is ABC
```

  - Passing pointers as arguments to functions
    - Function can modify caller data by pointer dereferencing
    - Type qualifier **const**:
      Modification by pointer derefencing *not* allowed!

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 72

Standard Library Functions

- Functions declared in **`string.h`** (part 1/2)
  - **`typedef unsigned int size_t;`**
    - type definition for length of strings
  - **`size_t strlen(const char *s);`**
    - returns the length of string **s**
  - **`int strcmp(const char *s1, const char *s2);`**
    - alphabetically compares string **s1** with string **s2**
    - returns -1 / 0 / 1 for less-than / equal-to / greater-than
  - **`int strncmp(const char *s1, const char *s2, size_t n);`**
    - same as previous, but compares maximal **n** characters
  - **`int strcasecmp(const char *s1, const char *s2);`**
  - **`int strncasecmp(const char *s1, const char *s2, size_t n);`**
    - same as string comparisons above, but case-insensitive

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 73

Standard Library Functions

- Functions declared in **`string.h`** (part 2/2)
  - **`char *strcpy(char *s1, const char *s2);`**
    - copies string **s2** into string **s1**
  - **`char *strncpy(char *s1, const char *s2, size_t n);`**
    - copies maximal **n** characters of string **s2** into string **s1**
  - **`char *strcat(char *s1, const char *s2);`**
    - concatenates string **s2** to string **s1**
  - **`char *strncat(char *s1, const char *s2, size_t n);`**
    - concatenates maximal **n** characters of string **s2** to string **s1**
  - **`char *strchr(const char *s, int c);`**
    - returns a pointer to the first character **c** in string **s**, or **NULL** if not found
  - **`char *strrchr(const char *s, int c);`**
    - returns a pointer to the last character **c** in string **s**, or **NULL** if not found
  - **`char *strstr(const char *s1, const char *s2);`**
    - returns a pointer to the first appearance of **s2** in string **s1** (or **NULL**)

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 74

File Processing

- Introduction
  - Up to now, all data processed is available only during program run time
    - when the program terminates, all data is lost
  - *Persistent data* is stored even after a program exits
  - Persistent data is stored in files
    - on the harddisk
    - on a removable disk (floppy disk, etc.)
    - on a tape, ...
  - Input and output from/to files is organized as *I/O streams*

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 75

File Processing

- I/O Streams
  - Standard I/O streams (opened by the system)
    - **`stdin`** standard input stream (i.e. **`scanf()`**)
    - **`stdout`** standard output stream (i.e. **`printf()`**)
    - **`stderr`** standard error stream (i.e. **`perror()`**)
  - File I/O streams (explicitly opened by a program)
    - Open a file **`fopen()`**
    - Write data to a file **`fprintf()`**, **`fputs()`**, etc.
    - Read data from a file **`fscanf()`**, **`fgets()`**, etc.
    - Close a file **`fclose()`**
  - In C, all I/O functions are ...
    - ... declared in header file **`stdio.h`**
    - ... implemented in the standard C library

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 76

Standard Library Functions

- Functions declared in **stdio.h** (part 1/4)
  - **int printf(const char *fmt, ...);**
  - **int scanf(const char *fmt, ...);**
    - formatted output/input to/from stream **stdin**/**stdout**
  - **int sprintf(char *s, const char *fmt, ...);**
  - **int sscanf(const char *s, const char *fmt, ...);**
    - formatted output/input to/from a string **s**
  - **int getchar(void);**
  - **int putchar(int c);**
    - input/output of a single character to/from stream **stdin**/**stdout**
  - **char *gets(char *s);**
  - **int puts(const char *s);**
    - input/output of strings to/from stream **stdin**/**stdout**

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 77

Standard Library Functions

- Functions declared in **stdio.h** (part 2/4)
  - **typedef __FILE FILE;**
    - opaque type for a file handle
  - **FILE *fopen(const char *n, const char *m);**
    - open file named **n** for input (**"r"**), output (**"w"**), or append (**"a"**)
    - returns a file handle, or **NULL** in case of an error
  - **int fclose(FILE *f);**
    - closes an open file handle
  - **int fflush(FILE *f);**
    - flushes any unwritten data from a buffer into the file
  - **int fprintf(FILE *f, const char *fmt, ...);**
  - **int fscanf(FILE *f, const char *fmt, ...);**
  - **int fgetc(FILE *f);**
  - **char *fgets(char *s, int n, FILE *f);**
  - **int fputc(int c, FILE *f);**
  - **int fputs(const char *s, FILE *f);**
    - input/output functions from/to stream **f**

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 78

Standard Library Functions

- Functions declared in **`stdio.h`** (part 3/4)
  - **`typedef unsigned int size_t;`**
    - type for size of a piece of memory
  - **`size_t fread(void *p, size_t s, size_t n, FILE *f);`**
    - binary input to memory location **`p`** for **`n`** times **`s`** bytes from file **`f`**
  - **`size_t fwrite(const void *p, size_t s, size_t n, FILE *f);`**
    - binary output from memory location **`p`** for **`n`** times **`s`** bytes to file **`f`**
  - **`int fseek(FILE *f, long pos, int w);`**
    - move to position **`pos`** in file **`f`** (from beginning/current pos/end)
  - **`long ftell(FILE *f);`**
    - return the current position in file **`f`** (from beginning)
  - **`void rewind(FILE *f);`**
    - move to beginning of file **`f`**
  - **`int feof(FILE *f);`**
    - check if end of file **`f`** is reached

Standard Library Functions

- Functions declared in **`stdio.h`** (part 4/4)
  - **`int ferror(FILE *f);`**
    - returns the current error status for file **`f`**
  - **`void perror(const char *prg);`**
    - print current error for program **`prg`** to stream **`stderr`**
  - **`int remove(const char *filename);`**
    - delete file **`filename`**
  - **`int rename(const char *old, const char *new);`**
    - rename file **`old`** to new name **`new`**

File Processing

- Program example: **`PhotoLab.c`** (part 1/8)

```
/**********************************************************/
/* PhotoLab.c: final assignment for EECS 10 in Fall 2004 */
/*                                                        */
/* modifications: (most recent first)                     */
/* 11/28/04 RD   adjusted for lecture usage               */
/**********************************************************/

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

/*** global definitions ***/

#define WIDTH  640      /* width of photo */
#define HEIGHT 480      /* height of photo */
#define SLEN    80      /* max. string length */

...
```


File Processing

- Program example: **`PhotoLab.c`** (part 2/8)

```
...
/* write a photo to the specified file from the       */
/* data structure; return 0 for success, >0 for error */

int WritePhotoPPM(
        char Filename[SLEN],
        unsigned char R[WIDTH][HEIGHT],
        unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT])
{
    FILE *File;
    int  x, y;

    File = fopen(Filename, "w");
    if (!File)
    {   printf("\nCannot open file \"%s\" for writing!\n",
                                              Filename);
        return(1);
    }
...
```

File Processing

- Program example: **`PhotoLab.c`** (part 3/8)

```
...
    fprintf(File, "P6\n");
    fprintf(File, "%d %d\n", WIDTH, HEIGHT);
    fprintf(File, "255\n");
    for(y=0; y<HEIGHT; y++)
    {   for(x=0; x<WIDTH; x++)
        {   fputc(R[x][y], File);
            fputc(G[x][y], File);
            fputc(B[x][y], File);
        }
    }
    if (ferror(File))
    {   printf("\nFile error while writing to file!\n");
        return(2);
    }
    fclose(File);
    return(0);  /* success! */
} /* end of WritePhotoPPM */
...
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 83

File Processing

- Program example: **`PhotoLab.c`** (part 4/8)

```
...
/* read a photo from the specified file into the      */
/* data structure; return 0 for success, >0 for error */

int ReadPhotoPPM( char Filename[SLEN],
        unsigned char R[WIDTH][HEIGHT],
        unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT])
{
    FILE *File;
    char Type[SLEN];
    int  Width, Height, MaxValue, x, y;

    File = fopen(Filename, "r");
    if (!File)
    {   printf("\nCannot open file \"%s\" for reading!\n",
                                        Filename);
        return(1);
    }
...
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 84

File Processing

- Program example: **`PhotoLab.c`** (part 5/8)

```
...
    fscanf(File, "%79s", Type);
    if (Type[0] != 'P' || Type[1] != '6' || Type[2] != 0)
    {   printf("\nUnsupported file format!\n");
        return(2);
    }
    fscanf(File, "%d", &Width);
    if (Width != WIDTH)
    {   printf("\nUnsupported image width %d!\n", Width);
        return(3);
    }
    fscanf(File, "%d", &Height);
    if (Height != HEIGHT)
    {   printf("\nUnsupported image height %d!\n", Height);
        return(4);
    }
...
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 85

File Processing

- Program example: **`PhotoLab.c`** (part 6/8)

```
...
    fscanf(File, "%d", &MaxValue);
    if (MaxValue != 255)
    {   printf("\nUnsupported maximum %d!\n", MaxValue);
        return(5);
    }
    if ('\n' != fgetc(File))
    {   printf("\nCarriage return expected!\n");
        return(6);
    }
    for(y=0; y<HEIGHT; y++)
    {   for(x=0; x<WIDTH; x++)
        {   R[x][y] = fgetc(File);
            G[x][y] = fgetc(File);
            B[x][y] = fgetc(File);
        }
    }
...
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 86

File Processing

- Program example: **PhotoLab.c** (part 7/8)

```
...
    if (ferror(File))
    {   printf("\nFile error while reading from file!\n");
        return(7);
    }
    fclose(File);
    return(0);  /* success! */
} /* end of ReadPhotoPPM */

...
```


File Processing

- Program example: **PhotoLab.c** (part 8/8)

```
...
/*** main program ***/

int main(void)
{
    unsigned char R[WIDTH][HEIGHT];
    unsigned char G[WIDTH][HEIGHT];
    unsigned char B[WIDTH][HEIGHT];

    ReadPhotoPPM("SimonsToys.ppm", R, G, B);
    /* do something to the picture ... */
    WritePhotoPPM("Output.ppm", R, G, B);

    return 0;
} /* end of main */

/* EOF */
```

File Processing

- Example session:

```
% vi PhotoLab.c
% gcc PhotoLab.c -o PhotoLab -Wall -ansi
% PhotoLab
% pnmtojpeg Output.ppm > Output.jpg
%
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 89

File Processing

- Example session:

```
% vi PhotoLab.c
% gcc PhotoLab.c -o PhotoLab -Wall -ansi
% PhotoLab
% pnmtojpeg Output.ppm > Output.jpg
%
% vi PhotoLab.c        (exchange R and G when writing)
% gcc PhotoLab.c -o PhotoLab -Wall -ansi
% PhotoLab
% pnmtojpeg Output.ppm > Output2.jpg
%
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 90

Program Modules

- C programs can be partitioned into *modules*
- Modules typically consist of
  - Module header file (file suffix **.h**)
  - Module program file (file suffix **.c**)
  - Module object file (file suffix **.o**)
- Modules can be *linked* together
  - Linker combines object files and libraries into an executable file
- C compiler consists of separate components
  - Preprocessor (processes # directives)
  - C language front end (processes C syntax and semantics)
  - Assembler (processes machine instructions)
  - Linker (processes object files and libraries)

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 91

Program Modules

- Source files
  - Header files: **name.h**
    - Inclusion of required header files
    - Definitions of exported constants
    - Declarations of exported global variables
    - Declarations of exported functions
  - Program files: **name.c**
    - Inclusion of required header files
    - Declaration and definition of local variables
    - Declaration and definition of local functions
    - Definitions of exported global variables
    - Definitions of exported functions

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 92

Program Modules

- Object files
  - **name.o**
    - Compiled object code of source file **name.c**
    - Use option **-c** in GNU compiler call to create object files
      gcc -c name.c -o name.o -Wall -ansi
- Executable file
  - **name**
    - Object files and libraries linked together into a complete file ready for execution
    - GNU compiler recognizes object files by .o suffix, so object files can be processed directly
      gcc obj1.o obj2.o -llib1 -llib2 -o name

Program Modules

- Module example: **Constants.h**

```
/**********************************************************/
/* Constants.h: header file for constant definitions     */
/* author: Rainer Doemer                                  */
/* modifications: (most recent first)                     */
/* 11/30/04 RD  initial version                           */
/**********************************************************/

#ifndef CONSTANTS_H
#define CONSTANTS_H

/*** global definitions ***/

#define WIDTH  640      /* width of photo */
#define HEIGHT 480      /* height of photo */
#define SLEN    80      /* max. string length */

#endif /* CONSTANTS_H */

/* EOF Constants.h */
```

Program Modules

- Module example: **FileIO.h**

```
/*********************************************************/
/* FileIO.h: header file for I/O module                */
/*********************************************************/
#ifndef FILE_IO_H
#define FILE_IO_H

#include "Constants.h"

int ReadPhotoPPM(       /* read a photo from file */
        char Filename[SLEN],
        unsigned char R[WIDTH][HEIGHT],
        unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);

int WritePhotoPPM(      /* write a photo to file */
        char Filename[SLEN],
        unsigned char R[WIDTH][HEIGHT],
        unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);

#endif /* FILE_IO_H */
/* EOF FileIO.h */
```

Program Modules

- Module example: **FileIO.c**

```
/*********************************************************/
/* FileIO.c: program file for I/O module               */
/*********************************************************/

#include <stdio.h>
#include "FileIO.h"

/*** function definitions ***/

int ReadPhotoPPM(char Filename[SLEN],
        unsigned char R[WIDTH][HEIGHT],
        unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT])
{   /* ... function body ... */
}
int WritePhotoPPM(char Filename[SLEN],
        unsigned char R[WIDTH][HEIGHT],
        unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT])
{   /* ... function body ... */
}
/* EOF FileIO.c */
```

Program Modules

- Module example: **`Mirror.h`**

```
/************************************************************/
/* Mirror.h: header file for mirror operation             */
/************************************************************/

#ifndef MIRROR_H
#define MIRROR_H

/*** header files ***/

#include "Constants.h"

/*** function declarations ***/

void Mirror(     /* flip the image horizontally */
        unsigned char R[WIDTH][HEIGHT],
        unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);

#endif /* MIRROR_H */

/* EOF Mirror.h */
```


Program Modules

- Module example: **`Mirror.c`**

```
/************************************************************/
/* Mirror.c: program file for mirror operation            */
/************************************************************/

#include "Mirror.h"

/*** function definitions ***/

/* mirror effect: flip the image horizontally */

void Mirror(
        unsigned char R[WIDTH][HEIGHT],
        unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT])
{
    /* ... function body ... */
}

/* EOF Mirror.c */
```

Program Modules

- Module example: **Main.c**

```
/**********************************************************/
/* Main.c: main program file                              */
/**********************************************************/

#include "Constants.h"
#include "FileIO.h"
#include "Mirror.h"

int main(void)
{
    unsigned char R[WIDTH][HEIGHT];
    unsigned char G[WIDTH][HEIGHT];
    unsigned char B[WIDTH][HEIGHT];

    ReadPhotoPPM("SimonsToys.ppm", R, G, B);
    Mirror(R, G, B);
    WritePhotoPPM("Output.ppm", R, G, B);

    return 0;
} /* end of main */

/* EOF Main.c */
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 99

Program Modules

- Example session:

```
% vi Constants.h
% vi FileIO.h
% vi FileIO.c
% vi Mirror.h
% vi Mirror.c
% vi Main.c
% gcc -c FileIO.c -o FileIO.o -Wall –ansi
% gcc -c Mirror.c -o Mirror.o -Wall –ansi
% gcc -c Main.c -o Main.o -Wall –ansi
% gcc FileIO.o Mirror.o Main.o -o PhotoLab
% PhotoLab
% pnmtojpeg Output.ppm > Output.jpg
%
```

EECS10: Computational Methods in ECE, Review 19-25 (c) 2004 R. Doemer 100