Introduction

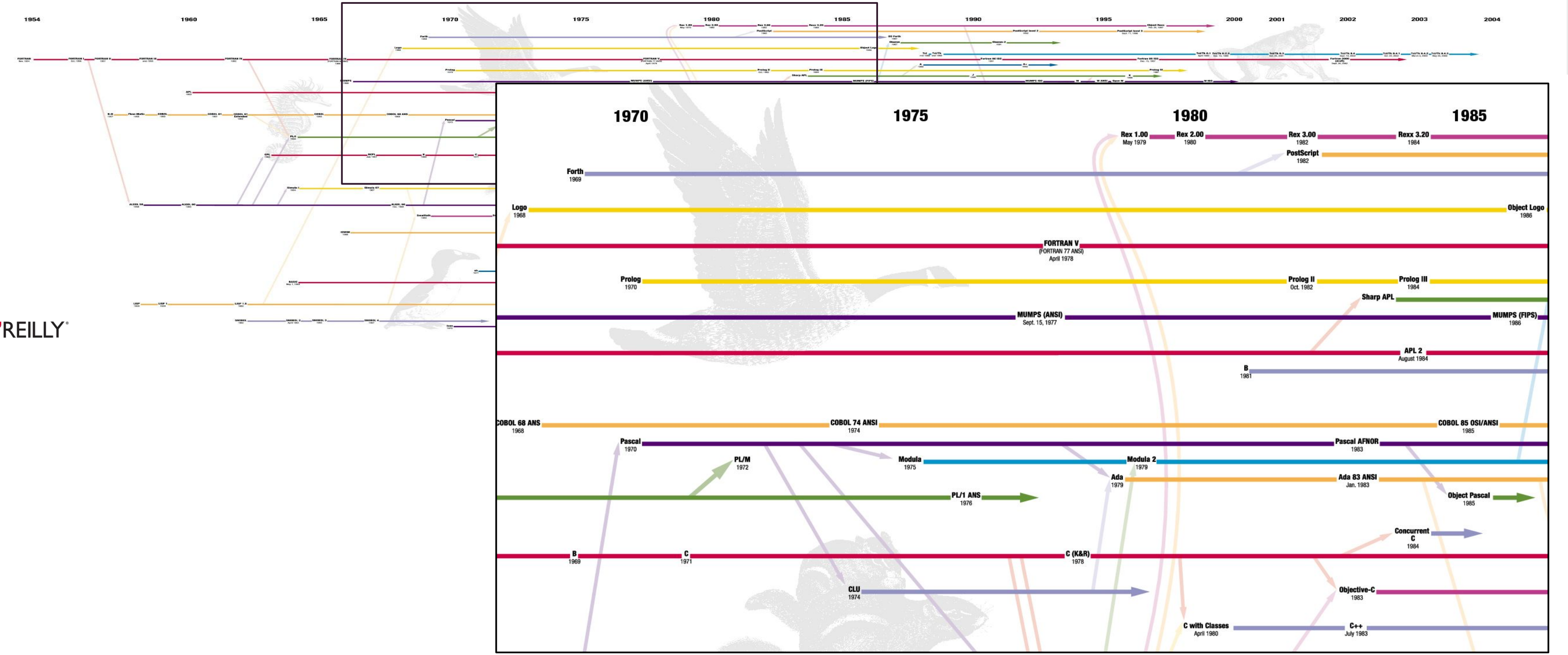
Outline

- Programming Languages
  - Object Oriented Programming
  - Procedural Programming
- What is C?
  - Short history
  - Features, Strengths and weaknesses
  - Relationships to other languages
- Writing C Programs
  - Editing
  - Compiling
- Structure of C Programs
  - Comments
  - Variables
  - Functions: main, function prototypes and functions
  - Expressions and Statements

Programming Languages

- Many programming languages exist, each intended for a specific purpose
  - Over 700 programming language entries on wikipedia
  - Should we learn all?
- Which is the best language? None!
- Choose the right tool for the job based on:
  - problem scope,
  - target hardware/software,
  - memory and performance considerations,
  - portability,
  - concurrency.

Programming Languages



O'REILLY

Object Oriented Programming

- ❑ Very useful to organize large software projects
- ❑ The program is organized as classes
- ❑ The data is broken into ‘objects’ and the sequence of commands becomes the interactions between objects:
  - Decide which classes you need
  - provide a full set of operations for each class
  - and make commonality explicit by using inheritance.
- ❑ Covered in CSC111 and CSC113

Procedural Programming

- The program is divided up into subroutines or procedures
- Allows code to become structured
- The programmer must think in terms of actions:
    - decide which procedures and data structures you want
- Procedural languages include:
    - Fortran
    - BASIC
    - Pascal
    - C

What is C?

- History:
  - 1972 - Dennis Ritchie – AT&T Bell Laboratories
  - 16-bit DEC PDP-11 computer (right)
  - 1978 - Published; first specification of language
  - 1989 - C89 standard (known as ANSI C or Standard C)
  - 1990 - ANSI C adopted by ISO, known as C90
  - 1999 - C99 standard: mostly backward-compatible, not completely implemented in many compilers
  - 2007 - work on new C standard C1X announced
- In this course: ANSI/ISO C (C89/C90)

What is C?

- Features:
  - Provides low -level access to memory
  - Provides language constructs that map efficiently to machine instructions
  - Few keywords (32 in ANSI C)
  - Structures, unions – compound data types
  - Pointers - memory, arrays
  - External standard library – I/O, other facilities
  - Compiles to native code
  - Systems programming:
    - OSes, like Linux
    - microcontrollers: automobiles and airplanes
    - embedded processors: phones, portable electronics, etc.
    - DSP processors: digital audio and TV systems
    - . . . Macro preprocessor
  - Widely used today, Extends to newer system architectures

What is C?

- Strengths:
  - Efficiency: intended for applications where assembly language had traditionally been used
  - Portability: hasn’t splintered into incompatible dialects; small and easily written
  - Power: large collection of data types and operators
  - Flexibility: not only for system but also for embedded system commercial data processing
  - Standard library
  - Integration with UNIX

- Weaknesses
  - Error-prone:
    - Error detection left to the programmer
  - Difficult to understand
    - Large programmes
    - Difficult to modify
  - Memory management
    - Memory management is left to the programmer

Relationship to Other Languages

- More recent derivatives: C++, Objective C, C#
- Influenced: Java, Perl, Python (quite different)
- C lacks:
  - Exceptions
  - Range-checking
  - Memory management and garbage collection.
  - Objects and object-oriented programming
  - Polymorphism
- Shares with Java:
  - /* Comments */
  - Variable declarations
  - if / else statements
  - for / while loops
  - function definitions (like methods)
  - Main function starts program

C Programs

- Editing:
  - C source code files has .c extension
  - Text files that can be edited using any text editor: Example `product.c`

```
#include <stdio.h>
main() {
  int a, b, c;
  a = 3; b = 2; c = a * b;
  printf("The product is %d", c);
}
```

- Compiling:
  - `gcc -o product product.c`
    - "`-o`" place the output in file product
    - "`product`" is the executable file
  - To execute the program:
    - `product` on windows or `./product` on Linux and Linux-like

C Compilers

- Several compilers
  - Microsoft compiler
  - GNU Compiler Collection (GCC)
  - : (see a List of C compilers)
- How to install GCC on windows:
  - MinGW: from https://nuwen.net/mingw.html
  - Cygwin: from https://cygwin.com/install.html
  - Don't forget to update the path!
- Compilation options:
  - `gcc -ansi product.c` : check the program compatibility with ANSI C
  - `gcc -Wall product.c` : enables all the warnings that are easy to avoid
  - In this course we will always use:

    `gcc -Wall -ansi -o product product.c`
- Cross Compilation: compiling on one platform to run on another

Structure of .c File

```
/* Begin with comments about file contents */

/* Insert #include statements and preprocessor definitions */

/* Function prototypes and variable declarations */

/* Define main() function {
      Function body
   }
*/

/* Define other function(s) {
      Function body
   }
*/
```

Structure of .c File: Comments

- /* this is a simple comment */
- Can span multiple lines

```
/* This comment
   Spans
   m u l t i p l e l i n e s */
```

- Completely ignored by compiler
- Can appear almost anywhere|

```
/* h e l l o . c –
   our f i r s t C program
   Created for CSC215 */
```

Structure of .c File: #include Preprocessor

- `#include` is a preprocessor:
  - Header files: constants, functions, other declarations
  - #include: read the contents of the header file stdio.h
- `stdio.h`: standard I/O functions for console and files

```
#include <stdio.h>
/* basic I/O facilities */
```

  - stdio.h – part of the C Standard Library
- other important header files:

assert.h	ctype.h	errno.h	float.h	limits.h	locale.h	math.h
signal.h	setjmp.h	stdarg.h	stddef.h	stdlib.h	string.h	time.h

- Included files must be on include path
  - standard include directories assumed by default
  - #include "stdio.h" – searches ./ for stdio.h first

Structure of .c File: #Variables and Constants

- Variables: named spaces in memory that hold values
  - Refer to these spaces using their names rather than memory addresses
  - Names selection adheres to some rules
  - Defined with a type that determines their domains and operations
  - Variable must be declared prior to their use
  - Can change their values after initialization

- Constants:
  - Do not change their values after initialization
  - Can be of any basic or enumerated data type
  - Declared by assigning a literal to a typed name, with the use of the keyword `const`

    ```
    const int LENGTH = 10;
    Const char NEWLINE = '\n';
    ```

  - Can also use the `#define` preprocessor

    ```
    #define LENGTH 10
    #define NEWLINE '\n'
    ```

Structure of .c File: Function Prototype

- Functions also must be declared before use
- Declaration called function prototype
- Function prototypes:

```
int factorial(int);
int factorial(int n);
```

- Prototypes for many common functions in header files for C Standard Library
- General form:

```
return_type function_name(arg1,arg2,...);
```

- Arguments: local variables, values passed from caller
- Return value: single value returned to caller when function exits
- void – signifies no return value/arguments int rand(void);

Structure of .c File: Function main

- `main()`: entry point for C program
- Simplest version:
  - no inputs,
  - outputs 0 when successful,
  - and nonzero to signal some error int main(void);
- Two-argument form of main():
  - access command-line arguments int main(int argc, char **argv);
  - More on the `char **argv` notation later

Structure of .c File: Function Definitions

- Function declaration

```
<return_type> <function_name>(<list_of_parameters>){
  <declare_variables;>
  <program_statements;>
  return <expression>;
}
```

- Must match prototype (if there is one)
  - variable names don’t have to match
- No semicolon at end
- Curly braces define a block – region of code
  - Variables declared in a block exist only in that block
  - Variable declarations before any other statements

Console Input and Output

- stdout, stdin: console output and input streams
  - puts(<string_expression>) : prints string to stdout
  - putchar(<char_expression>) : prints character to stdout
  - <char_var> = getchar() : returns character from stdin
  - <string_var> = gets(<buffer>) : reads line from stdin into string
  - printf(control_string, arg1, arg2, …) to be discussed later

Structure of .c File: Expressions and statements

- Expression:
  - a sequence of characters and symbols that can be evaluated to a single data item.
  - consists of: literals, variables, subexpressions, interconnected by one or more *operators*
    - Numeric literals like 3 or 4.5
    - String literals like “Hello”
  - Example expressions:
    - Binary arithmetic

      `x+y , x-y , x*y , x/y , x%y`
- Statement:
  - A sequence of characters and symbols causes the computer to carry out some definite action
  - Not all statements have values
  - Example statement:

    `y = x+3*x/(y-4);`
  - Semicolon ends statement (not newline)

Output Statements

```
/* The main ( ) function */
int main (void)/* entry point */ {
   /* write message to console */
   puts( "Hello World!" );
   return 0; /* exit (0 => success) */
}
```

- `puts(<string>)`: output text to console window (stdout) and end the line
- String literal: written surrounded by double quotes
- return 0; exits the function, returning value 0 to caller

Variables, Types and Expressions

Outline

- Variables
- Datatypes
  - Basic data types
  - Derived data types
  - User-defined data types
- Expressions
  - Operators: arithmetic, relational, logical, assignment, inc-/dec- rement, bitwise
  - Evaluation
- Formatted input/output

Variables

- Named values
  - Naming rules:
    - Made up of letters, digits and the underscore character ‘_’
    - Must not begin with a digit
    - Must not be a special keyword

`auto`	`break`	`case`	`char`	`const`	`continue`
`default`	`do`	`double`	`else`	`enum`	`extern`
`float`	`for`	`goto`	`if`	`int`	`long`
`register`	`return`	`short`	`signed`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`	`unsigned`	`void`
`volatile`	`while`				

- Variable declaration:
  - Must declare variables before use
  - Variable declaration: `int n; float phi;`
    - int - integer data type
    - float - floating-point data type
  - Many other types
- Variable initialization:
  - Uninitialized variable assumes a default value
  - Variables initialized via assignment operator: `n = 3;`
  - Can also be initialized at declaration: `float phi = 1.6180339887;`
  - Can declare/initialize multiple variables at once: `int a, b, c = 0, d = 4;`

Basic Data Types

- Data type determines the variable's domain and applicable operations
- Four types: **char** **int** **float** **double**
- Modifiers: signed unsigned short long
- Combinations:

	Type	Bits	Range		
Char	[signed] char	8	-128	..	127
	unsigned char	8	0	..	259
int	[signed] int	16 (at least)	-2^{15}	..	$2^{15}-1$
	unsigned int	16 (at least)	0	..	$2^{16}-1$
	[signed] short [int]	16	-2^{15}	..	$2^{15}-1$
	unsigned short [int]	16	0	..	$2^{16}-1$
	[signed] long [int]	32 (at least)	-2^{31}	..	$2^{31}-1$
	unsigned long [int]	32 (at least)	0	..	$2^{32}-1$
float	float	32	1.2E-38 .. 3.4E+38 (6 dig-prec)		
double	double	64	2.3E-308 .. 1.7E+308 (15 dig-prec)		
	long double	80 (at least)	3.4E-4932 .. 1.1E+4932 (19 dig-prec)		

- What about boolean? strings?

Boolean?

- No special boolean type
- Evaluating boolean and logical expressions:
  - results in integer 1 if the logic is true
  - results in 0 if the logic is false
- Interpretation of integer as boolean:
  - 0 is perceived as false
  - any non-zero value is perceived as true

Strings ?

- Strings stored as character array
- Null-terminated (last character in array is ’\0’: null character)

```
char course[7] = {'C', 'S', 'C', '2', '1', '5', '\0'};
char course[] = {'C', 'S', 'C', '2', '1', '5', '\0'};
```

- Not written explicitly in string literals

```
char course[7] = "CSC215";
char course[] = "CSC215";
```

- Special characters specified using \ (escape character):
  - `\\` – backslash
  - `\'` – apostrophe
  - `\"` – quotation mark
  - `\b`, `\t`, `\r`, `\n` – backspace, tab, carriage return, linefeed
  - `\o`oo, `\x`hh – octal and hexadecimal ASCII character codes, e.g. \x41 – `'A'`, \060 – `'0'`

Initialization of Variables

- Local variables:
  - declared inside a function
  - are not initialized by default
- Global variables:
  - declared outside of functions
    - On top of the program
  - are initialized by default:

Type	**Default value**
int	0
char	'\0'
float	0
double	0
pointer	null
Derived types	apply recursively

Constants

- The previous examples can be rewritten as:

```
int main(void) /* entry point */ {
  const char msg [ ] = "Hello World!";
  /* write message to console */
  puts(msg);
}
```

- **const** keyword: qualifies variable as constant
- **char**: data type representing a single character; written in quotes: ’a’, ’3’, ’n’
- const char msg[]: a constant array of characters

Expressions

- Expression:
  - a sequence of characters and symbols that can be evaluated to a single data item.
  - consists of: literals, variables, subexpressions, interconnected by one or more *operators*

- Operator:
  - Can be unary, binary, and ternary
  - Categories:
    - Arithmetic: `+x, -x, x+y , x-y , x*y , x/y , x%y`
    - Relational `x==y, x!=y, x<y, x<=y, x>y, x>=y`
    - Logical `x&&y, x||y, !x`
    - Bitwise `x&y, x|y, x^y, x<<y, x>>y, ~x`
    - Assignment `x=y, x+=y, x-=y, x*=y, x/=y, x%=y`
      `x<<=y, x>>=y, x&=y, x|=y, x^=y`
    - inc-/dec- rement `++x, x++, --x, x--`
    - Conditional `x?y:z`
    - More: `*x, &x, (type)x, sizeof(x), sizeof(<type>)`

Arithmetic Operators

- 2 Unary operators: + -
- 5 Binary operators: + - * / %
  - If both operands are of type int, the result is of type int
- Example:

```
int main() {
  int a = 9, b = 4, c;
  c = a+b;
  printf("a+b = %d \n",c);
  c = a-b;
  printf("a-b = %d \n",c);
  c = a*b;
  printf("a*b = %d \n",c);
  c=a/b;
  printf("a/b = %d \n",c);
  c=a%b;
  printf("Remainder when a divided by b = %d \n",c);
  return 0;
}
```

Relational Operators

- 6 Binary operators: == != > >= < <=
- Checks the relationship between two operands:
  - if the relation is true, it yieldss 1
  - if the relation is false, it yields value 0
- Example:

```
int main(){
  int a = 5, b = 5, c = 10;
  printf("%d == %d = %d \n", a, b, a == b); /* true */
  printf("%d == %d = %d \n", a, c, a == c); /* false */
  printf("%d > %d = %d \n", a, b, a > b); /*false */
  printf("%d > %d = %d \n", a, c, a > c); /*false */
  printf("%d < %d = %d \n", a, b, a < b); /*false */
  printf("%d < %d = %d \n", a, c, a < c); /*true */
  printf("%d != %d = %d \n", a, b, a != b); /*false */
  printf("%d != %d = %d \n", a, c, a != c); /*true */
  printf("%d >= %d = %d \n", a, b, a >= b); /*true */
  printf("%d >= %d = %d \n", a, c, a >= c); /*false */
  printf("%d <= %d = %d \n", a, b, a <= b); /*true */
  printf("%d <= %d = %d \n", a, c, a <= c); /*true */
  return 0;
}
```

Logical Operators

- 1 Unary operator: `!` and 2 binary operators: `&&` `||`
- Example:

```
int main(){
  int a = 5, b = 5, c = 10, result;
  result = (a = b) && (c > b);
  printf("(a = b) && (c > b) equals to %d \n", result);
  result = (a = b) && (c < b);
  printf("(a = b) && (c < b) equals to %d \n", result);
  result = (a = b) || (c < b);
  printf("(a = b) || (c < b) equals to %d \n", result);
  result = (a != b) || (c < b);
  printf("(a != b) || (c < b) equals to %d \n", result);
  result = !(a != b);
  printf("!(a == b) equals to %d \n", result);
  result = !(a == b);
  printf("!(a == b) equals to %d \n", result);
  return 0;
}
```

Bitwise Operators

- 1 Unary operator ~ and 5 binary operators & | ^ << >>
- Examples:

```
int main(){
  int a = 12;
  int b = 25;
  printf("complement=%d\n",~35);
  printf("complement=%d\n",~-12);
  printf("Output = %d", a&b);
  printf("Output = %d", a|b);
  printf("Output = %d", a^b);

  int num=212;
  printf("Right shift by 3: %d\n", num>>3);
  printf("Left shift by 5: %d\n", num<<5);
  return 0;
}
```

```
 35 00000000 00100011  ~
-36 11111111 11011100
```

```
-12 11111111 11110100  ~
 11 00000000 00001011
```

```
12 00000000 00001100
25 00000000 00011001
   -----------------  &
 8 00000000 00001000
```

```
12 00000000 00001100
25 00000000 00011001
   -----------------  |
29 00000000 00011101
```

```
12 00000000 00001100
25 00000000 00011001
   -----------------  ^
21 00000000 00010101
```

```
212 00000000 11010100
 26 00000000 00011010  >>
```

```
 212 00000000 11010100
6784 00011010 10000000  <<
```

Assignment Operators

- 11 Binary operators: = += -= *= /= %= &= |= ^= <<= >>=
- Example:

```
int main(){
  int a = 5, c;
  c = a;
  printf("c = %d \n", c);
  c += a; /* c = c+a */
  printf("c = %d \n", c);
  c -= a; /* c = c-a */
  printf("c = %d \n", c);
  c *= a; /* c = c*a */
  printf("c = %d \n", c);
  c /= a; /* c = c/a */
  printf("c = %d \n", c);
  c %= a; /* c = c%a */
  printf("c = %d \n", c);
  return 0;
}
```

Increment/Decrement operators

- 2 Binary operators: ++ --
- Example:

```
int main(){
  int a = 10, b = 100;
  float c = 10.5, d = 100.5;
  printf("++a = %d \n", ++a); /* 11 */
  printf("b++ = %d \n", b++); /* 100 */
  printf("c-- = %f \n", c--); /* 10,500000 */
  printf("--d = %f \n", --d); /* 99.500000 */
  return 0;
}
```

Ternary Conditional Operator

- Syntax: `<conditionalExpression> ? <expression1> : <expression2>`
- The conditional operator works as follows:
    - <conditionalExpression> is evaluated first to non-zero (1) or false (0).
    - if <conditionalExpression> is true, <expression1> is evaluated
    - if <conditionalExpression> is false, <expression2> is evaluated.
- Example:

```
int main(){
  char February;
  int days;
  printf("If this year is leap year, enter 1. If not enter any integer: ");
  scanf("%c",&February);
  /* If test condition (February == 'l') is true, days equal to 29. */
  /* If test condition (February =='l') is false, days equal to 28. */
  days = (February == '1') ? 29 : 28;
  printf("Number of days in February = %d",days);
  return 0;
}
```

More Operators

- sizeof: unary operator returns data (constant, variable, array, structure...)
- Example:

```
int main(){
  int a, e[10];
  float b;
  double c;
  char d;
  printf("Size of int=%lu bytes\n",sizeof(a));
  printf("Size of float=%lu bytes\n",sizeof(b));
  printf("Size of double=%lu bytes\n",sizeof(c));
  printf("Size of char=%lu byte\n",sizeof(d));
  printf("Size of integer type array having 10 elements = %lu bytes\n", sizeof(e));
  return 0;
}
```

Evaluating Expressions

- Expression: A sequence of characters and symbols that can be evaluated to a single data item.
- Expression evaluation:
  - Order of operations:
    Use parenthesis to override order of evaluation
  - Example: Assume `x = 2.0 and y = 6.0.`
    Evaluate the statement:
    `float z = x+3*x/(y-4);`
    1. Evaluate expression in parentheses
       → `float z = x+3*x/2.0;`
    2. Evaluate multiplies and divides, from left-to-right
       → `float z = x+6.0/2.0;` → `float z = x+3.0;`
    3. Evaluate addition float:
       → `float z = 5.0;`
    4. Perform initialization with assignment Now, `z = 5.0.`
  - How do I insert parentheses to get z = 4.0?

<table>
<tr><th>Operator</th><th>Associativity</th></tr>
<tr><td><function>(), [ ], ->, .</td><td>left to right</td></tr>
<tr><td>!, ~, ++, --, +, -, *, (<type>), sizeof</td><td>right to left</td></tr>
<tr><td>*, /, %</td><td>left to right</td></tr>
<tr><td>+, - (unary)</td><td>left to right</td></tr>
<tr><td><<, >></td><td>left to right</td></tr>
<tr><td><, <=, >, >=</td><td>left to right</td></tr>
<tr><td>==, !=</td><td>left to right</td></tr>
<tr><td>&</td><td>left to right</td></tr>
<tr><td>^</td><td>left to right</td></tr>
<tr><td>|</td><td>left to right</td></tr>
<tr><td>&&</td><td>left to right</td></tr>
<tr><td>||</td><td>left to right</td></tr>
<tr><td>? :</td><td>left to right</td></tr>
<tr><td>= += -= *= /= %= &= ^= |= <<= >>=</td><td>right to left</td></tr>
<tr><td>,</td><td>lrft to right</td></tr>
</table>

Formatted Input and Output

- Function `printf`

  `printf(control_string, arg1, arg2, …);`

  - control_string is the control string or conversion specification consists of % followed by a specifier

    `%[flags][width][.precision][length]specifier`
  - Specifiers (place holders):
    - %d - int (same as %i)
    - %ld - long int (same as %li)
    - %f - decimal floating point
    - %lf - double or long double
    - %e - scientific notation (similar to %E)
    - %c - char
    - %s - string
    - %o - signed octal
    - %x - hexadecimal (similar to %X)
    - %p - pointer
    - %%- %
  - Optional width, length precision and flags

Flags	: -	+	#	0
Width	: *	number		
Length	: h	l	L	
Precision	: .*	.number		

Formatted Input and Output

- Numeric:

% [[<FLAG>] [<LENGTH>] [.<PRECISION>]] **<SPECIFIER>**

-	Left align
+	Prefix sign to the number
#	Prefix 0 to octal, 0x/0X to hexadecimal Force decimal point with e E f G g
0	Pad with leading zeros
□	Replace positive sign with space

<Number>	Minimum length
*	Passing it as an arg
Default:	All

<Number>	Decimal digits
*	Passing it as an arg
Default:	6

%d	int (same as %i)
%ld	long int (same as %li)
%f	decimal floating point
%lf	double or long double
%e	scientific notation (similar to %E)
%g	shorter of f and e
%c	char
%o	signed octal
%x	hexadecimal (similar to %X)

- String:

% [[<FLAG>] [<LENGTH>] [.] [<WIDTH>]] **<SPECIFIER>**

-	Left align

<Number>	Minimum length
*	Passing it as an arg
Default:	All

<Number>	Max number of characters to print
*	Passing it as an arg
Default:	0 with ., all if . is omitted

%s	string

Formatted Input and Output

- Function scanf

  `scanf(control_string, arg1, arg2, …);`

  - Control_string governs the conversion, formatting, and printing of the arguments
  - Each of the arguments must be a pointer to the variable in which the result is stored
  - So: `scanf("%d", &var);` is a correct one, while `scanf("%d", var);` is not correct
  - Place holders:
    - %d - int (same as %i)
    - %ld - long int (same as %li)
    - %f - float
    - %lf - double
    - %c - char
    - %s - string
    - %x - hexadecimal

Macros

- Preprocessor macros begin with # character
  - `#define msg "Hello World"`
    defines msg as “Hello World” throughout source file
- #define can take arguments and be treated like a function
  #define add3(x,y,z) ((x)+(y)+(z))
  - parentheses ensure order of operations
  - compiler performs inline replacement; not suitable for recursion
- #if, #ifdef, #ifndef, #else, #elif , #endif conditional preprocessor macros
  - can control which lines are compiled
  - evaluated before code itself is compiled, so conditions must be preprocessor defines or literals
  - the gcc option -Dname=value sets a preprocessor define that can be used
  - Used in header files to ensure declarations happen only once
- Conditional preprocessor macros:
  - #pragma preprocessor directive
  - #error, #warning trigger a custom compiler error/warning
  - #undef msg remove the definition of msg at compile time

Control Flow

Outline

- Blocks and compound statements
- Conditional statements
    - if - statement
    - if-else - statement
    - switch - statement
    - ? : opertator
    - Nested conditional statements
- Repetitive statements
    - for - statement
    - while - statement
    - do-while - statement
    - Nested repetitive statements
    - Break and continue statements
- Unconditional jump: `goto`

Blocks and Compound Statements

- A simple statement ends in a semicolon: `z = foo(x+y);`
- Consider the multiple statements:

```
temp = x+y ;
z = foo (temp) ;
```

  - Curly braces – combine into compound statement/block
  - Block can substitute for simple statement
  - Compiled as a single unit
  - Variables can be declared inside
  - No semicolon at end

```
{
   int temp = x+y;
   z = foo(temp);
}
```

- Block can be empty {}

Blocks and Compound Statements

- Blocks nested inside each other

```
{
  int temp = x+y ;
  z = foo ( temp ) ;
  {
    float temp2 = x*y ;
    z += bar ( temp2 ) ;
  }
}
```

- Variables declared inside a block are only visibly within this block and its internatl blocks

Conditional Statements

- **if** - Statement
- **if-else** - Statement
- **switch** - Statement
- **? :** Ternary operator
- No boolean type in ANSI C
  - introduced in C99
- Relational and logical expressions are evaluated to:
  - `1` if they are logically true
  - `0` if they are logically false
- Numeric expressions are considered false if they are evaluated to integer 0
- Pointer expressions are considered false if they are evaluated to null

if- Statement

- Syntax:

```
if (<condition>)
  <statement>;
```

- Example:

```
if ( x % 2 == 0)
  y += x / 2 ;
```

  - Evaluate condition: `(x % 2 == 0)`
    - If true, execute inner statement: `y += x/2;`
    - Otherwise, do nothing
  - Inner statements may be block

if-else - Statement

- Syntax:

```
if (<condition>)
  <statement1>;
else
  <statement2>;
```

- Example:

```
if ( x % 2 == 0)
  y += x / 2 ;
else
  y += ( x + 1 ) / 2;
```

  - Evaluate condition: `(x % 2 == 0)`
    - If true, execute first statement: `y += x/2;`
    - Otherwise, execute second statement: `y += ( x + 1 ) / 2;`
  - Either inner statements may be block

Nesting if/if-else Statements

- Can have additional alternative control paths by nesting `if` statements:

```
if (<condition>)
  <statement1>; /* can be an if or if-else statement*/
else
  <statement2>; /* can be an if or if-else statement*/
```

- Conditions are evaluated in order until one is met; inner statement then executed
  - if multiple conditions true, only first executed
- Example:

```
if ( x % 2 == 0)
  y += x / 2 ;
else if ( x % 4 == 1)
  y += 2 * (( x + 3 )/ 4 );
else
  y += ( x +1 )/ 2 ;
```

Nesting if/if-else Statements

- **Dangling `else`** , example:

```
if ( x % 4 == 0)
if ( x % 2 == 0)
y = 2;
else
y = 1;
```

```
if ( x % 4 == 0)
  if ( x % 2 == 0)
    y = 2;
  else
    y = 1;
```

```
if ( x % 4 == 0)
  if ( x % 2 == 0)
    y = 2;
else
  y = 1;
```

  - To which if statement does the else keyword belong? Belongs to the nearest if in the same block
  - To associate else with outer if statement: use braces

```
if ( x % 4 == 0) {
  if ( x % 2 == 0)

     y = 2;
} else
     y = 1;
```

switch - Statement

- Syntax:

```
switch (<int or char expression>) {
  case <literal1>:    <statements>
                      [break;]
  [more cases]
  [default:  <statements>]
}
```

- Provides multiple paths
- Case labels: different entry points into block
- Compares evaluated expression to each case:
  - When match found, starts executing inner code until `break;` reached
  - Execution “falls through” if `break;` is not included

switch - Statement

- Example:

```
switch ( ch ) {
  case 'Y' : /* ch == 'Y ' */
             /* do something */
             break ;
  case 'N' : /* ch == 'N ' */
             /* do something else */
             break ;
  default :  /* otherwise */
             /* do a third thing */
}
```

Loops (Iterative Statements)

- **while** - loop
- **for** - loop
- **do-while** - loop
- **break** and **continue** keywords

Loops: while - Statement

- Syntax:

```
while ( <condition> )
 <loop body>
```

- Simplest loop structure – evaluate body as long as condition is true
- Condition evaluated first, so body may never be executed
- Example:

```
int x = 0;
while ( x < 10 ) {     /* While x is less than 10 */
  printf( "%d\n", x );
  x++;                 /* Update x so the condition breaks eventually
*/
}
```

Loops: for - Statement

- Syntax:

```
for ( [<initialization>] ; [<condition>] ; [<modification>] )
   <loop body>
```

- Example:

```
int i , j = 1;
for ( i = 1; i <= n ; i ++)
   j *= i ;
printf(“%d\n”, j);
```

  - A “counting” loop
  - Inside parentheses, three expressions, separated by semicolons:
    - Initialization: i = 1
    - Condition: i <= n
    - Modification: i++

Loops: for - Statement

- Any expression can be empty (condition assumed to be “true”):

```
for (;;) /* infinite loop */
   <loop body>
```

- Compound expressions separated by commas
  - Comma: operator with lowest precedence, evaluated left-to-right

```
for ( i = 1 , j = 1; i <= n , j % 2 != 0 ; j *= i , i ++)
   <loop body>
```

- Equivalent to while loop:

```
<initialization>
while (<condition>) {
   <loop body>
   <modification>
}
```

Loops: do-while - Statement

- Syntax:

```
do {
   <loop body>
} while( <condition> );
```

- Differs from while loop – condition evaluated after each iteration
  - Body executed at least once
  - Note semicolon at end
- Example:

```
char c ;
do {
/ * loop body * /
puts( "Keep going? (y/n) " ) ;
c = getchar();
/ * other processing * /
} while ( c == 'y' && /* other conditions */ );
```

Loops: Nested Loops

- A nested loop is a loop within a loop
  - an inner loop within the body of an outer one.

    ```
    for ([<initialization>];[<condition>];[<modification>])
      <loop body> /* another loop here */
    ```

- Can nest any loop statement within the body of any loop statement
- Can have more than two levels of nested loops

Loops: break - Statement

- Sometimes want to terminate a loop early
  - break; exits innermost loop or switch statement to exit early
  - Consider the modification of the do-while example:

```
char c ;
do {
  /* loop body */
  puts ( "Keep going? (y/n) " ) ;
  c = getchar() ;
  if ( c != 'y')
    break ;
  /* other processing */
} while ( /* other conditions */ ) ;
```

Loops: continue - Statement

- Use to skip an iteration
  - continue; skips rest of innermost loop body, jumping to loop condition
- Example:

```
int i , ret = 1 , minval;
for ( i = 2; i <= (a > b? a:b); i++) {
    if ( a % i ) /* a not divisible by i  */
        continue;
    if ( b % i == 0) /* b and a are multiples of i  */
        ret = i;
}
printf("%d\n", ret);
```

Unconditional Jump

- `goto`: transfers program execution to a labeled statement in the current function
  - DISCOURAGED
  - easily avoidable
  - requires a label
- Label: a plain text, except C keywords, followed by a colon, prefixing a code line
  - may occur before or after the `goto` statement
- Example:

```
int main () {
  int a = 10;
  LOOP:do {
         if ( a == 15) {
           a = a + 1;
           goto LOOP;
         }
         printf("value of a: %d\n", a++);
       } while( a < 20 );
   return 0;
}
```

CSC215 Lecture

Functions and Modular Programming

Outline

- Functions:
  - Need, Definition
  - Defining functions
  - Calling functions
  - Prototypes
- Scopes
  - Scope and visibility
  - Storage classes
- Recursive functions
- Multiple source files
- Makefiles

Introduction

- Design your solution so that it keeps the flow of control as simple as possible
  - top-down design:
    decompose the problem into smaller problems each can be solved easily
- Some problems are complicated
  - break them down into smaller problems
  - conquer each sub problem independently
- Your programs will consist of a collection of user-defined functions
  - each function solves one of the small problems
  - you call (invoke) each function as needed

What is a Function?

- Function: a group of statements that together perform a task
  - divide up your code into separate functions such that each performs a specific task
  - every C program has at least one function, which is **main()**
  - most programs define additional functions
- Why
  - to avoid repetitive code : "reusability" written once, can be called infinitely
  - to organize the program : making it easy to code, understand, debug and collaborate
  - to hide details : "what is done" vs "how it is done"
  - to share with others
- Defining functions:
  - Predefined (library functions): We have already seen
    - main, printf, scanf, getchar, gets
  - User-defined

Defining Functions

- Syntax:

```
<return_type> <function_name>(<parameter_list>){
  <function_body>
}
```

  - Return_type: data type of the result
    - Use `void` if the function returns nothing
    - if no type is specified and void is not used: it defaults to `int`
  - Function_name: any valid identifier
  - Parameter_list:
    - declared variables: <param_type> <param_name>
    - comma separated
  - Function_body:
    - declaration statements
    - other processing statements
    - `return` statement, if not void

Example

- ❑ In many application, finding the greatest common factor is an important step
- ❑ GCF function:
  - takes two input integers
  - finds the greatest integer that divide both of them
  - returns the result to the calling context
  - Euclidean algorithm:
    - if a > b → gcf(a, b) = gcf(b, a mod b)
    - if b > a, swap a and b
    - Repeat until b is 0
- ❑ In c:

```
int gcf(int a, int b){
  /* if a < b swap them, to be discussed later*/
  while (b) {
    int temp = b ;
    b = a % b ;
    a = temp ;
  }
  return a;
}
```

Calling Functions

- Syntax:
  `<function name>(<argument list>)`
- A function is invoked (called) by writing:
  - its name, and
  - an appropriate list of arguments within parentheses
    - arguments must match the parameters in the function definition in:
      1- count , 2- type and 3- order
- Arguments are passed by value
  - each argument is evaluated, and
  - its value is copied to the corresponding parameter in the called function
- What if you need to pass the variable by reference?
  - you cannot
  - But you can pass its address by reference

Calling Functions

- Example:

```
/* Does not work as expected*/
void swap(int a, int b) {
   int temp = a;
   a = b;
   b = temp;
}

int main(){
   int a = 3, b = 5;
   swap(a, b);
   printf("a=%d, b=%d\n", a, b);
   return 0;
}
```

```
/* Works as expected*/
void swap(int *a, int *b){
   int temp = *a;
   *a = *b;
   *b = temp;
}

int main(){
   int a = 3, b = 5;
   swap(&a, &b);
   printf("a=%d, b=%d\n", a, b);
   return 0;
}
```

Calling Functions

- A function can be called from any function, not necessarily from `main`
- Example:

```
void swap(int *a, int *b){
  int temp = *a; *a = *b; *b = temp;
}
int gcf(int a, int b){
  if (b > a) swap(&a, &b);
  while (b) {
    int temp = b ;
    b = a % b ;
    a = temp ;
  }
  return a;
}
int main(){
  int a = 3, b = 5;
  printf("GCF of %d and %d is %d\n", a, b, gcf(a, b) );
  return 0;
}
```

Function Prototypes

- If function definition comes textually after use in program:
  - The compiler complains: `warning: implicit declaration of function`
- Declare the function before use: Prototype
  `<return_type> <function_name>(<parameters_list>);`
- Parameter_list does not have to name the parameters
- Function definition can be placed anywhere in the program after the prototypes.
- lIf a function definition is placed in front of main(), there is no need to include its function prototype.

Function Prototypes: Example

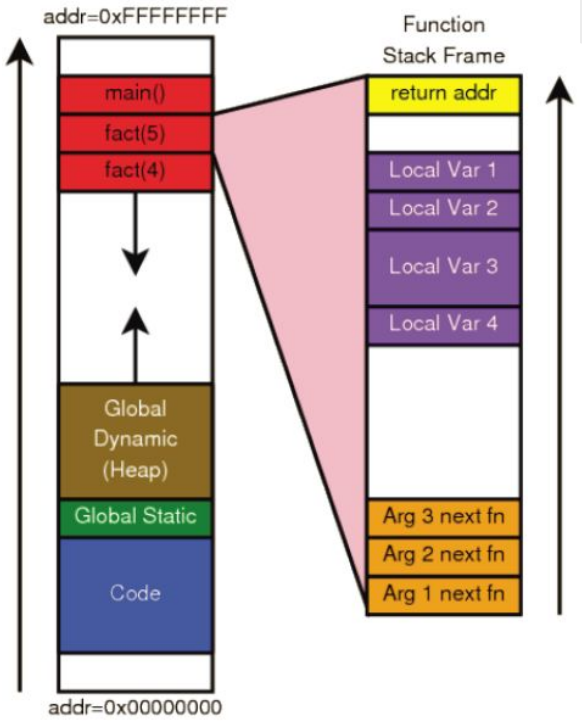
```
#include <stdio.h>
int gcf(int, int);
void swap(int*, int*);
int main(){
  int a = 33, b = 5;
  printf("GCF of %d and %d is %d\n", a, b, gcf(a, b) );
  return 0;
}
int gcf(int a, int b){
  if (b > a) swap(&a, &b);
  while (b) {
    int temp = b ;
    b = a % b ;
    a = temp ;
  }
  return a;
}
void swap(int *a, int *b){
  int temp = *a; *a = *b; *b = temp;
}
```

Function Stub

- A stub is a dummy implementation of a function with an empty body
    - A placeholder while building (other parts of) a program
        - so that it compiles correctly
    - Fill in one-stub at a time
    - Compile and test if possible

Memory Model

- Program code
  - Read only
  - May contain string literals
- Stack (automatic storage):
  - Function variables:
    - Local variables
    - Arguments for next function call
    - Return location
  - Destroyed when function ends
- Heap:
  - Dynamically allocated space
- Data segment:
  - Global variables
  - Static variables



Scopes

- Scope: the parts of the program where an identifier is valid
- A global variable:
  - A.K.A. external variable
  - defined outside of the local environment (outside of functions)
  - available anywhere within the file
- A local variable:
  - A.K.A. internal and automatic variable
  - defined within the local environment inside { }
  - local to that block, whether the block is a block within a function or the function itself
  - parameters in a function header are local to that function
  - it can mean different things in different contexts
  - if two variables share the same name but are in different blocks or functions
    - the variable declared in the current environment will be the one used in a reference

Scopes: Examples

Ex1:

```c
#include <stdio.h>

void doubleX(float x){
  x *= 2;
  printf("%f\n", x);
}

int main(){
  float x = 3;
  doubleX(x);
  printf("%f\n", x);
  return 0;
}
```

```
6.000000
3.000000
```

Ex2:

```c
#include <stdio.h>

float x = 10;

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

int main(){
  float x = 3;
  doubleX();
  printf("%f\n", x);
  return 0;
}
```

```
20.000000
3.000000
```

Ex3:

```c
#include <stdio.h>

float x = 10;

void doubleX(float x){
  x *= 2;
  printf("%f\n", x);
}
void printX(){
  printf("%f\n", x);
}
int main(){
  float x = 3;
  doubleX(x);
  printf("%f\n", x);
  printX();
  return 0;
}
```

```
6.000000
3.000000
10.000000
```

Ex4:

```c
#include <stdio.h>

int main(){
  int x = 5;
  if (x){
    int x = 10;
    x++;
    printf("%d\n", x);
  }
  x++;
  printf("%d\n", x);
  return 0;
}
```

```
11
6
```

Storage Classes

- Storage Classes: a modifier precedes the variable to define its scope and lifetime
- `auto`: the default for local variables
- `register`: advice to the compiler to store a local variable in a register
    - the advice is not necessarily taken by the compiler
- `static`: tells the compiler that the storage of that variable remains in existence
    - Local variables with static modifier remains in memory so that they can be accessed later
    - Global variables with static modifier are limited to the file where they are declared
- `extern`: points the identifier to a previously defined variable
- Initialization:
    - in the absence of explicit initialization:
        - static and external variables are set to 0
        - automatic and register variables contain undefined values (garbage)

Storage Classes: Examples

Ex1 :

```
#include <stdio.h>

int main(){
  float x = xx;     ✘

  return 0;
}

float xx;

void foo(){
  float x = xx;     ✓
}


/*
  main doesn't know
  about xx


*/
```

Ex1 correction :

```
#include <stdio.h>

int main(){
  extern float xx;  ✓
  float x = xx;     ✓

  return 0;
}

float xx;

void foo(){
  float x = xx;     ✓
}

/*
  declare xx in main
  as extern to point to
  the external xx, this
  will not create new xx
*/
```

Ex2 :

```
/*file1.c         */

#include <stdio.h>
int sp = 0;
double val[1000];

int main(){
  return 0;
}
```

```
/*file2.c         */

#include <stdio.h>

void foo(){
  printf("%d", sp);    ✘
}

int bar(){
  return (int)val[0];  ✘
}
```

Ex2 correction :

```
/*file1.c         */

#include <stdio.h>
int sp = 0;
double val[1000];

int main(){
  return 0;
}
```

```
/*file2.c         */

#include <stdio.h>
extern int sp;          ✓
extern double val[];    ✓

void foo(){
  printf("%d", sp);     ✓
}
int bar(){
  return (int)val[0];   ✓
}
```

Recursive Functions

- Recursive function: a function that calls itself (directly, or indirectly)
- Example:

```
void change (count){
   ..
   ..
   change(count);
   ..
}
```

- The algorithm needs to be written in a recursive style
  - a step or more uses the algorithm on a smaller problem size
- It must contain a base case that is not recursive
- Each function call has its own stack frame
  - consumes resources

Recursive Functions: Examples

- Multiply x × y:

```
int multiply(int x, int y){
  if (y == 1) return x;
  return x + multiply(x, y-1);
}
```

- Power x^y:

```
int power(int x, int y){
  if (y == 0) return 1;
  return x * multiply(x, y-1);
}
```

- Factorial `x!`:

```
int fac(int x){
  if (x == 1) return 1;
  return x * fac(x-1);
}
```

- Fibonacci:

```
int fib(int x) {
  if (x == 0) return 0;
  if (x == 1) return 1;
  return fib(x-1) + fib(x-2);
}
```

- Palindrome:

```
int isPal(char* s, int a, int b) {
  if (b >= a) return 1;
  if (s[a] == s[b])
    return isPal(s, a+1, b-1);
  return 0;
}
```

Optional Parameters

- C permits functions to have optional parameters
- Syntax: <returntype> <name>(<paramslist>, …)
  - … indicates that further parameters can be passed, must be listed only after the required parameters
  - since you specify the parameters as …, you do not know their names!
- How to use these additional parameters when they are passed?
  - `stdarg.h` file contains the definition of `va_list` (variable argument list)
  - declare a variable of type `va_list`
  - use the macro `va_start` which initializes your variable to the first of the optional params
  - use the function `va_arg` which returns the next argument

Optional Parameters

- Example:

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

int sum(int, ...);

int main(){
  printf("Sum of 15 and 56 = %d\n",  sum(2, 15, 56) );
  return 0;
}

int sum(int num_args, ...){
  int val = 0;
  va_list ap;
  int i;
  va_start(ap, num_args);
  for(i = 0; i < num_args; i++)
    val += va_arg(ap, int);
  va_end(ap);
  return val;
}
```

Multiple Source Files

- A typical C program: lot of small C programs, rather than a few large ones
    - each .c file contains closely related functions (usually a small number of functions)
    - header files to tie them together
    - Makefiles tells the compiler how to build them
- Example:
    - a calc program defines:
        - a stack structure and its:
        - pop and push functions
        - getch and ungetch to read one symbol at a time
        - getop function to parse numbers and operators
        - main function
    - main calls: getop, pop, and push
      getop calls: getch and ungetch
    - can be organized in 4 separate files:
    - Where to place prototypes and external declarations?
    - How to compile the program?

```
/*        stack.c       */
#include <stdio.h>
int sp = 0;
double val[1000];
void push(double x){
   ...
}
double pop(){
   ...
}
```

```
/*        getch.c       */
#include <stdio.h>

ch getch(){
   ...
}
void ungetch(char c){

}
```

```
/*        main.c        */
#include <stdio.h>

int main(){

}
```

```
/*        getop.c       */
#include <stdio.h>

int getop(char[] s){

}
```

Multiple Source Files: Header File

- Prototypes can be placed in a single file, called a header file
  - as well as all other shared definitions and declarations
  - typically contains definitions and declarations
    - but not executable code

- Example: calc program

  add a header file calc.h contains:
  - prototypes and
  - common declarations

  and **include** it where needed!

```
/*       getch.c      */
#include <stdio.h>
#include "calc.h"
ch getch(){
   ...
}
void ungetch(char c){

}
```

```
/*       main.c      */
#include <stdio.h>
#include "calc.h"
int main(){

}
```

```
/*       calc.H      */
void push(double);
double pop();
ch getch();
void ungetch(charc);
int getop(char[]);
```

```
/*       stack.c      */
#include <stdio.h>
#include "calc.h"
int sp = 0;
double val[1000];
void push(double x){
   ...
}
double pop(){
   ...
}
```

```
/*       getop.c      */
#include <stdio.h>
#include "calc.h"
int getop(char[] s){

}
```

File Inclusion

- Syntax:
  - **#include** <filename>
    - search for the file filename in paths according to the compiler defined rules
    - replaced by the content if the file filename
  - **#include** "filename"
    - search for the file filename in source program directory or according to the compiler rules
    - replaced by the content if the file filename
- When an included file is changed
  - all files depending on it must be recompiled
- Multiple inclusion of a file: problem
- Circular inclusion: problem

Conditional Inclusion

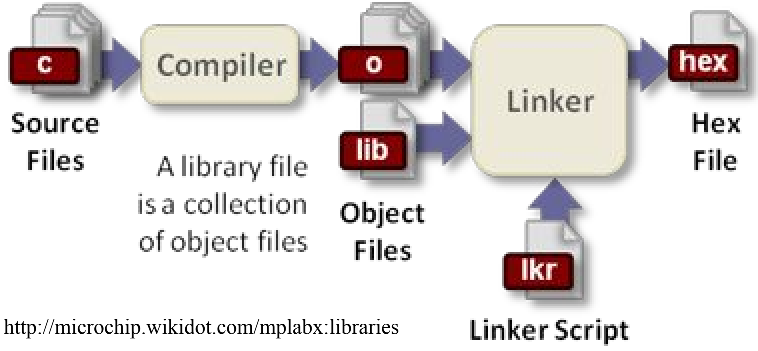
- Control preprocessing with conditional statements
- Syntax:
  - **`#if`**
    - evaluates a constant integer expression
    - if the expression is non-zero, all following lines until an #endif or #elif or #else are included
  - **`#else`** , **`#elif`**
    - provide alternative paths
  - **`#endif`**
    - marks the end of the conditional block

- Can be used to avoid repetitive and circular inclusions:
  - included file:

```
#if !defined(HDR)
#define HDR
/* contents of hdr.h go here */
#endif
```

Compiling Multiple Sources

- The compiler 1st stage is the preprocessor
  - deals with the # directives: define, include, conditional ...
- The compiler 2nd stage is translate .c files to .o files
  - each .c file will be translated to a single .o file
  - to invoke this stage only, use gcc option -c
- The compiler then links .o files together
  - along with library files



http://microchip.wikidot.com/mplabx:libraries

Makefile

- To compile multiple source files:
  `gcc -Wall -ansi -o` `<output> <file1.c> <file2.c> ...`
- Or use makefiles:
  - Special format file used to build and manage the program automatically
  - contains a collection of rules and commands
- Syntax:

```
<target> [<more targets>]  : [<dependent files>]
<tab> <commands>
```

- Example:

```
calc: main.c stack.c getch.c getop.c calc.h
    gcc -Wall -ansi -o calc main.c stack.c getch.c getop.c
```

  - How to use: `on the command line type: make calc` ⏎

Makefile

- Conventional macros:
  - `CC`   : Program for compiling C programs; default is `cc'
  - `CFLAGS`: Extra flags to give to the C compiler.

- Example:

```
CC=gcc
CFLAGS= -Wall -ansi

calc: main.c stack.c getch.c getop.c calc.h
    ${CC} ${CFLAGS} -o calc main.c stack.c getch.c getop.c
```

- Usage:

`make`

or

`make calc`

Makefile

- At object level:

```
CC=gcc
CFLAGS= -Wall -ansi

calc: main.o stack.o getch.o getop.o calc.h
    ${CC} ${CFLAGS} -o calc main.o stack.o getch.o getop.o

main.o: main.c calc.h
    ${CC} ${CFLAGS} -c main.c

stack.o: stack.c calc.h
    ${CC} ${CFLAGS} -c stack.c

getch.o: getch.c calc.h
    ${CC} ${CFLAGS} -c getch.c

getop.o: getop.c calc.h
    ${CC} ${CFLAGS} -c getop.c
```

- Can invoke any target by:
  `make <target>`
- If the dependency object file has not changed since last compile, it will linked as is. Otherwise, it is recompiled

Makefile

- With useful extra targets:

```
CC=gcc
CFLAGS= -Wall -ansi

calc: main.o stack.o getch.o getop.o
	${CC} ${CFLAGS} -o calc main.o stack.o getch.o getop.o

main.o: main.c calc.h
	${CC} ${CFLAGS} -c main.c

stack.o: stack.c calc.h
	${CC} ${CFLAGS} -c stack.c

getch.o: getch.c calc.h
	${CC} ${CFLAGS} -c getch.c

getop.o: getop.c calc.h
	${CC} ${CFLAGS} -c getop.c

clean:
	rm *.o calc
```

Pointers and Arrays

Outline

- Physical and virtual memory
- Pointers
  - Declaration, operators, casting
  - Passing as arguments and returning from functions
- Arrays
  - Declaration, initialization, accessing individual elements
  - Arrays as constant pointers
  - Multidimensional arrays
- Pointer Arithmetic
  - Assignment, addition and subtraction, increment and decrement, comparative operators
  - Unary operators precedency
- Cryptic C code

Pointers and Memory Addresses

- Physical memory: physical resources where data can be stored and accessed by your computer
  - Cache
  - RAM
  - hard disk
  - removable storage
- Physical memory considerations:
  - Different sizes and access speeds
  - Memory management – major function of OS
  - Optimization – to ensure your code makes the best use of physical memory available
  - OS moves around data in physical memory during execution
  - Embedded processors – may be very limited

Pointers and Memory Addresses

- Virtual memory:
  - abstraction by OS
  - addressable space accessible by your code
- How much physical memory do I have?
  Answer: 2 MB (cache) + 2 GB (RAM) + 100 GB (hard drive) + . . .
- How much virtual memory do I have?
  Answer: <4 GB (32-bit OS)
- Virtual memory maps to different parts of physical memory
- Usable parts of virtual memory: stack and heap
  - stack: where declared variables go
  - heap: where dynamic memory goes

Pointers and variables

- Every variable residing in memory has an address!
  - What doesn't have an address?
    - register variables
    - constants/literals/preprocessor defines
    - expressions (unless result is a variable)
- C provides two unary operators, `&` and `*`, for manipulating data using pointers
  - address operator `&`: when applied to a variable `x`, results in the address of `x`
  - dereferencing (indirection) operator `*`:
    when applied to a pointer, returns the value stored at the address specified by the pointer.
- All pointers are of the same size:
  - they hold the address (generally 4 bytes)
  - pointer to a variable of type T has type T*
  - a pointer of one type can be converted to a pointer of another type by using an explicit cast:
    `int *ip;     double *dp;      dp= (double *)ip;` OR `ip = (int*)dp;`

Examples

```
char a;         /* Allocates 1 memory byte */
char *ptr;      /* Allocates memory space to store memory address */
ptr = &a;       /* store the address of a in ptr. so, ptr points to a */
int x = 1, y = 2, z[10]={0, 1, 2, 3, 4, 5, 4, 3, 2, 1};
int *ip;        /* ip is a pointer to int */
ip = &x;        /* ip now points to x */
y = *ip;        /* y is now 1 */
*ip = 0;        /* x is now 0 */
ip = &z[0];     /* ip now points to z[0] */
printf("%d %d %d", x, y, *ip);
y = *ip + 1;
printf("%d %d %d", x, y, *ip);
*ip += 1;
printf("%d %d %d", x, y, *ip);
```

```
0 1 00 1 00 1 1
```

Dereferencing & Casting Pointers

- You can treat dereferenced pointer same as any other variable:
  - get value, assign, increment/decrement
- Dereferenced pointer has new type, regardless of real type of data
- null pointer, i.e. 0 (NULL): pointer that does not reference anything
- Can explicitly cast any pointer type to any other pointer type
  `int* pn; ppi = (double *)pn;`
- Implicit cast to/from `void *` also possible
- Possible to cause segmentation faults, other difficult-to-identify errors
  - What happens if we dereference ppi now?

Passing Pointers by Value

```
/* Does not work as expected*/
void swap(int a, int b){
  int temp = a;
  a = b;
  b = temp;
}

int main(){
  int a[] = {3, 5, 7, 9};
  swap(a[1], a[2]);
  printf("a[1]=%d, a[2]=%d\n", a[1], a[2]);
  return 0;
}
```

```
/* Works as expected*/
void swap(int *a, int *b){
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main(){
  int a = {3, 5, 7, 9};
  swap(&a[1], &b[2]);
  printf("a[1]=%d, a[2]=%d\n",a[1], a[2]);
  return 0;
}
```

Function Returning a Pointer

- Functions can return a pointer
  Example: `int * myFunction() { . . . }`

- But: never return a pointer to a local variable

```
#include <stdio.h>
char * get_message ( ) {
  char msg[] = "Hello";
  return msg;
}
int main ( void ){
  char * str = get_message() ;
  puts(str);
  return 0;
}
```

```
#include <stdio.h>
char * get_message ( ) {
  static char msg[] = "Hello";
  return msg;
}
int main ( void ){
  char * str = get_message() ;
  puts(str);
  return 0;
}
```

- unless it is defined as static

- Multiple returns? Use extra parameters and pass addresses as arguments.

Arrays

- Fixed-size sequential collection of elements of the same type
- Primitive arrays implemented as a pointer to block of contiguous memory locations
  - lowest address corresponds to the first element and highest address to the last element
- **Declaration:** `<element_type> <array_name> [<positive_int_array_size>];`
  Example: `int balance[8]; /* allocate 8 int elements*/`
- **Accessing individual elements:** <array_name>[<element_index>]
  Example `int a = balance[3]; /* gets the 4th element's value*/`
- **Array Initializer:** `<type> <name>[<optional_size>] = {<comma-sep elements>};`
  `<optional_size> must be >= # of elements`

Arrays

- Under the hood: the array is constant pointer to the first element

```
int *pa = arr;  ⇔  int *pa = &arr[0];
```

- Array variable is not modifiable/reassignable like a pointer

```
int a[5];
int b[] = {-1, 3, -5, 7, -9};
a = b;
error: assignment to expression with array type
```

- `arr[3]` is the same as `*(arr+3)`: to be explained in few minutes

- Iterating over an array:

```
int i;                                  int *pi;
for(i = 0; i < n; i++)        ⇔         for(pi = a; pi < a + n; pi++)
  arr[i]++;                               (*pi)++;
```

Strings

- There is no string type, we implement strings as arrays of chars

```
char str[10];  /* is an array of 10 chars or a string */
char *str;  /* points to 1st char of a string of unspecified length
*/
```

- There is a string.h library with numerous string functions
    - they all operate on arrays of chars and include:

      `strcpy(s1, s2)` : copies s2 into s1 (including ‘\0’ as last char)

      `strncpy(s1, s2, n)` : same but only copies up to n chars of s2

      `strcmp(s1, s2)` : returns a negative int if s1 < s2, 0 if s1 = = s2 and a positive int if s1 > s2

      `strncmp(s1, s2, n)` : same but only compares up to n chars

      `strcat(s1, s2)` : concatenates s2 onto s1 (this changes s1, but not s2)

      `strncat(s1, s2, n)` : same but only concatenates up to n chars

      `strlen(s1)` : returns the integer length of s1

      `strchr(s1, ch)` : returns a pointer to the 1st occurrence of ch in s1 (or NULL if not found)

Arrays

- Array length? no native function

```
#include <stdio.h>
int main() {
  char* pstr = "CSC215";
  printf("%s\t%d\n", pstr, sizeof(pstr));
  char astr[7] = "CSC215";
  printf("%s\t%d\n", astr, sizeof(astr));
  int aint[10];
  printf("%d\t%d\n", sizeof(aint[0]), sizeof(aint));
  int* pint = aint;
  printf("%d\t%d\n", sizeof(pint[0]), sizeof(pint));
  return 0;
}
```

```
CSC215    4
CSC215    7
4     40
4     4
```

- How about: `sizeof(arr)==0?0 : sizeof(arr)/sizeof(arr[0]);`

  can be defined as a macro:

```
#define arr_length(arr)(sizeof(arr)==0?0 : sizeof(arr)/sizeof((arr)[0]))
```

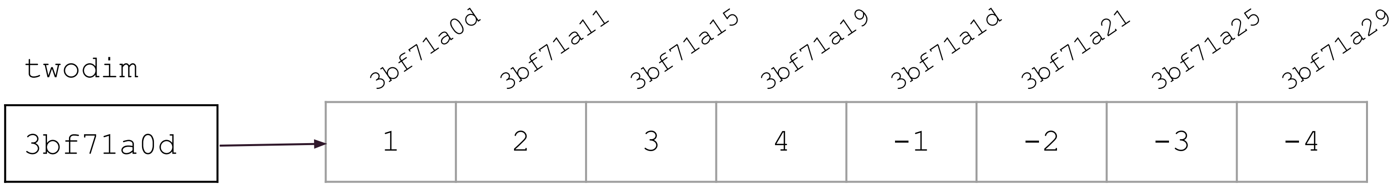
Multidimensional Arrays

- **Syntax**: <type> <name>**[**<size1>**][**<size2>**]...[**<sizeN>**]**;
  Example: int threedim[5][10][4];

- **Initializer**: = { { {..},{..},{..}}, {...}, {...}}
  Example: int twodim[2][4]={{1,2,3,4},{-1,-2,-3,-4}}; /* or simply: */
  int twodim[2][4]={1, 2, 3, 4, -1, -2, -3, -4};
  - You cannot omit any dimension size

- **Accessing individual elements**:
  <name>**[**<dim1index>**][**<dim2index>**]...[**<dimNindex>**]**
  Example: twodim[1][2]=5; printf("%d\n", twodim[0][3]);

- **Allocation**:



Multidimensional Arrays

- **Pointer style:** <type> ** <name>; /* add * for every extra dimension */
  a pointer to the 1st element of an array, each element of which is a pointer to the 1st element in an array

- More flexibility:
  Example:
  ```
  char b[4][7] = {"CSC111", "CSC113", "CSC212", "CSC215"};
  char *bb[] = {"CSC215", "This is a beautiful morning","M","I guess so"};
  ```

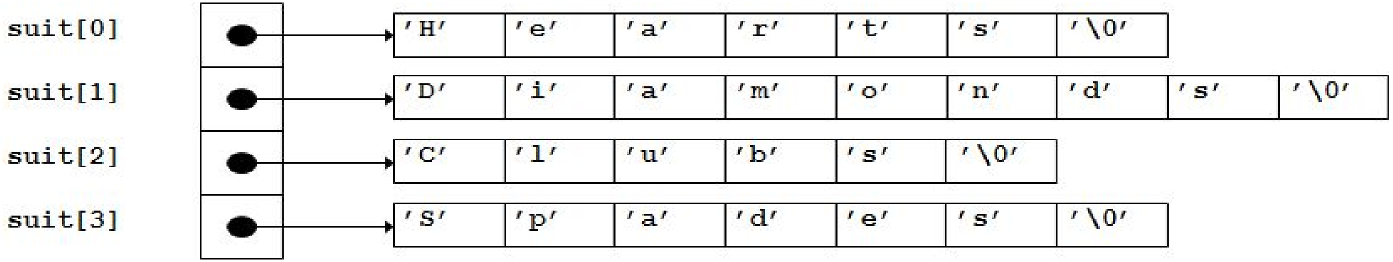
- Still have [ ]?
  - To define pure pointer 2D array:
    - Declare <type>** x variable
    - Allocate memory for N elements of type <type>* (1^{st} dimension)
    - For each of these elements, allocate memory for elements of type <type> () (2^{nd} dimension)
  - Ignore it for now, you learn first about memory managements in C.

- Arguments to main: int main(int argc, char** argv){ … }
  - Name of the executable is always the element at index 0
    ```
    for (i=0; i<argc; i++) printf("%s\n", argv[i]);
    ```

Arrays of Pointers

- Example is an array of strings:

  `char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };`
  - strings are pointers to the first character
  - `char *` each element of suit is a pointer to a char
  - strings are not actually stored in the array `suit`, only pointers to the strings are stored
  - `suit` array has a fixed size, but strings can be of any size



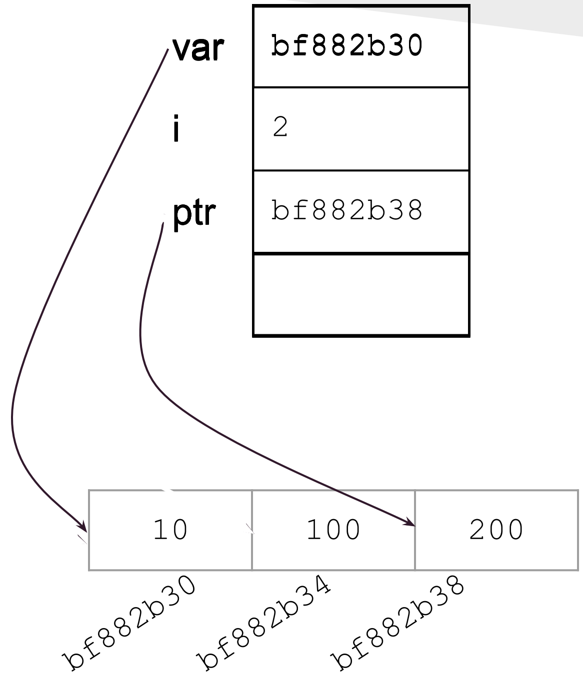
Pointer Arithmetic

- Assignment operator = : initialize or assign a value to a pointer
  - value such as 0 (`NULL`), or
  - expression involving the address of previously defined data of appropriate type, or
  - value of a pointer of the same type, or different type casted to the correct type
- Arithmetic operators + , −: scaling is applied
  - adds a pointer and an integer to get a pointer to an element of the same array
  - subtract an integer from a pointer to get a pointer to an element of the same array
  - Subtract a pointer from a pointer to get number of elements of the same array between them
- Increment/Decrement ++ , --: scaling is applied
  - result is undefined if the resulting pointer does not point to element within the same array
- Comparative operators:
  - == , != : can be used to compare a pointer to 0 (NULL)
  - == , != , > , >= , < , <= : can be used between two pointers to elements in the same array
- All other pointer arithmetic is illegal

Example: Increment/Decrement Operators

```
#include <stdio.h>
int main (){
  int var[] = {10, 100, 200};
  int i, *ptr;
  /* let us have array address in pointer */
  ptr = var;
  for ( i = 0; i < 3; i++){
    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );
    /* move to the next location */
    ptr++;
  }
  return 0;
}
```

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```



Example: Comparative operators

```
#include <stdio.h>
const int MAX = 3;
int main (){
  int var[] = {10, 100, 200};
  int i, *ptr;
  /* let us have address of the first element in pointer */
  ptr = var;
  i = 0;
  while ( ptr <= &var[MAX - 1] ){
    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );
    /* point to the next location */
    ptr++;
    i++;
  }
  return 0;
}
```

Precedence of Pointer Operators

- Unary operators `&` and `*` have same precedence as any other unary operator
  - with associativity from right to left.
- Examples:

```
c=*++cp       c=*(++cp)
c=*cp++       c=*(cp++)
c=++*cp       c=++(*cp)
???           c=(*cp)++
```

Cryptic vs. Short C Code

- Consider the following function that copies a string into another:

```
void strcpy(char *s, char *t){
  int i;
  i = 0;
  while ((*s = *t) != '\0') {
    S++;
    T++;
  }
}
```

  - Now, consider this

```
void strcpy(char *s, char *t){
  while ((*s++ = *t++) != '\0');
}
```

  - and this

```
void strcpy(char *s, char *t){
  while (*s++ = *t++);
}
```

- Obfuscation (software)
- The International Obfuscated C Code Contest *http://www.ioccc.org/*

Memory Management

Outline

- Static vs Dynamic Allocation
- Dynamic allocation functions
  `malloc, realloc, calloc, free`
- Implementation
- Common errors

Static Allocation

- Allocation of memory at compile-time
  - before the associated program is executed
- Let’s say we need a list of 1000 names:
  - We can create an array statically
    char names[1000][20]
  - allocates 20000 bytes at compile time
  - wastes space
  - restricts the size of the names

Dynamic allocation of memory

- Heap is a chunk of memory that users can use to dynamically allocated memory
  - Lasts until freed, or program exits.
- Allocate memory during runtime as needed

  `#include <stdlib.h>`
- Use sizeof number to return the number of bytes of a data type.
- To reserve a specified amount of free memory and returns a void pointer to it, use:
  - `malloc`
  - `calloc`
  - `Realloc`
- To release a previously allocated memory block, use:
  - `free`

Dynamic Allocation: `malloc`

- C library function allocates the requested memory and returns a pointer to it

  ```
  void *malloc(size_t size)
  ```
  - `size_t: unsigned integer type`
  - size: the size of the requested memory block, in bytes
  - return value: a pointer to the allocated memory, or NULL if the request fails
  - memory block is not cleared (undefined)

- Example:

  ```
  char *str = (char *) malloc(3*sizeof(char));
  *str = 'O';
  *(str+1) = 'K';
  *(str+2) = '\0';
  ```

Dynamic Allocation: `realloc`

- C library function attempts to resize the memory block pointed to by a pointer

  `void *realloc(void *ptr, size_t size)`

  - ptr: a previously allocated pointer (using malloc, calloc or realloc)
    - if `NULL`, a new block is allocated ⇔ `malloc`
  - size: the total size of the requested memory block, in bytes
    - if `0`, the memory pointed to by ptr is freed ⇔ `free`
  - return value: a pointer to the allocated memory, or NULL if the request fails
  - may move the memory block to a new location

- Example:

```
char *str = (char *) malloc( 3 * sizeof(char) );
*str = 'H';         *(str+1) = 'i';         *(str+2) = '\0';

str = (char *) realloc( str , 6 * sizeof(char) );
*(str+1) = 'e';   *(str+2) = 'l';         *(str+3) = 'l';
*(str+4) = 'o';   *(str+5) = '\0';
```

What is considered
a bad practice here?

Dynamic Allocation: `calloc`

- Dynamically allocating arrays:
  - allows the user to avoid fixing array size at declaration
  - use malloc to allocate memory for array when needed:

```
int *a = (int *)malloc(sizeof(int)*10);
a[0]=1;
```

- Alternatively, use:

```
void *calloc(size_t nitems, size_t size)
```

  - nittems: the number of elements to be allocated
  - size: the size of the requested memory block, in bytes
  - return value: a pointer to the allocated memory, or NULL if the request fails
  - sets allocated memory to 0s

- Example:

```
int size;    char *s;
printf("How many characters?\n"); scanf("%d", &size);
s = (char *)calloc(size+1, 1);
printf("type string\n");  gets(s);
```

Dynamic Deallocation: `free`

- C library function deallocates the memory previously allocated
  - by a call to calloc, malloc, or realloc

```
void free(void *ptr)
```

  - ptr : the pointer to a memory block previously allocated with malloc, calloc or realloc to be deallocated
  - If a null pointer is passed as argument, no action occurs.
- Can only be used on pointers that are dynamically allocated
- It is an error to free:
  - A pointer that has already been freed
  - Any memory address that has not been directly returned by a dynamic memory allocation routine
- Example:

```
char *str = (char *)malloc(3*sizeof(char));
/* use str */
free(str);
```

How It Is Done

- Best-fit method:
  an area with m bytes is selected, where m is the smallest available chunk of contiguous memory equal to or larger than n.
- First-fit method:
  returns the first chunk encountered containing n or more bytes.
- Prevention of fragmentation
  a memory manager may allocate chunks that are larger than the requested size if the space remaining is too small to be useful.
- When free is called:
  returns chunks to the available space list as soon as they become free and consolidate adjacent areas

Common Dynamic Allocation Errors

- Initialization errors
  do not assume memory returned by malloc and realloc to be filled with zeros
- Failing to check return values
  since memory is a limited resource, allocation is not always guaranteed to succeed
- Memory leak
  Forgetting to call free when the allocated memory is no more needed
- Writing to already freed memory
  if pointer is not set to NULL it is still possible to read/write from where it points to
- Freeing the same memory multiple times
  may corrupt data structure
- Improper use of allocation functions
  malloc(0): insure non-zero length

Example

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int input, n, count = 0;
  int *numbers = NULL, *more_numbers = NULL;
  do {
    printf ("Enter an integer (0 to end): ");     scanf("%d", &input);
    count++;
    more_numbers = (int*)realloc(numbers, count * sizeof(int));
    if (more_numbers!=NULL) {
      numbers = more_numbers;
      numbers[count-1]=input;
    else {
      free(numbers);
      puts("Error (re)allocating memory");
      return 1;
    }
  } while (input!=0);
  printf ("Numbers entered: ");
  for (n=0;n<count;n++) printf ("%d ",numbers[n]);
  free (numbers);
  return 0;
```

Example: mat.c

```
#include <stdio.h>
#include <stdlib.h>
#include "mat.h"

int** get_matrix(int rows, int cols){
  int  i, **matrix;
  if (matrix = (int**)malloc(rows*sizeof(int*)))
    if (matrix[0] = (int*)calloc(rows*cols,sizeof(int))){
      for (i=1; i<rows; i++)
        matrix[i] = matrix[0] + cols * i;
      return matrix;
    }
  return NULL;
}
```

Compare with:

```
if (matrix =
      (int**) malloc(rows*sizeof(int*)))
  for (i=0; i<rows; i++)
    if(!(matrix[i] =
      (int*) calloc(cols,sizeof(int))))
      return NULL;
   return matrix;
```

```
void free_matrix(int** m){
  free(m[0]);
  free(m);
}
```

Compare with:

```
void free_matrix(int*** m){
  free(*m[0]);
  free(*m);
  *m = NULL;
}
```

Example: mat.c

```
void fill_matrix(int** m, int rows, int cols){
  int i, j;
  for (i=0; i < rows; i++)
    for (j=0; j < cols; j++){
      printf("Enter element [%d, %d]:", i, j); scanf("%d", &m[i][j]);
    }
}
void print_matrix(int** m, int rows, int cols){
  int i, j;
  for (i=0; i < rows; i++){
    for (j=0; j < cols; j++) printf("%d\t", m[i][j]);
    printf("\n");
  }
}
int** transpose(int** m, int rows, int cols){
  int i, j, **t = get_matrix(cols, rows);
  for (i=0; i < rows; i++)
    for (j=0; j < cols; j++) t[j][i] = m[i][j];
  return t;
}
```

Example: mat.h

```
#if !defined MAT
#define MAT

int** get_matrix(int, int);

void free_matrix(int**);      /* OR */  void free_matrix(int***);

void fill_matrix(int**, int, int);

void print_matrix(int**, int, int);

int** transpose(int**, int, int);

#endif
```

Example: test.c

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

int main(){
  int r, c;
  printf("How many rows? "); scanf("%d", &r);
  printf("How many columns? "); scanf("%d", &c);

  int** mat = get_matrix(r, c);

  fill_matrix(mat, r, c);
  print_matrix(mat, r, c);

  int** tra = transpose(mat, r, c);
  print_matrix(tra, c, r);

  free_matrix(mat);      /* OR */
  free_matrix(tra);
  return 0;
}
```

```
free_matrix(&mat);
free_matrix(&tra);
```

User-Defined Data Types

Outline

- Enumerated
  - definition , declaration of variables
- Structures
  - definition , declaration of variables , members access , initialization
  - nested structures , size of structure
  - pointer to structure , array of structure
- Union
  - definition , declaration of variables , size of union
- Bitfield
- typedef keyword

Enumerated Constants

- An enumeration is a user-defined data type that consists of integral constants.
- Synax:

```
enum <type_name> {<id1>[=<val1], <id2>[=<val2>], ..., <idn>[=<valn>]};
```

- Example:

```
enum suit {
    club = 0,
    diamonds = 10,
    hearts = 20,
    spades = 3,
};
```

- Values can be omitted
  - Assigned automatically starting from 0, or from last assigned value, and increasing

```
                0        1          2           3             4          5         6
enum week {sunday,monday,tuesday,wednesday,thursday,friday,saturday };
```

Structure

- A Structure is a collection of related variables:
  - possibly of different types, unlike arrays
  - grouped together under a single name
- A structure type in C is called `struct`
- A Structure holds data that belongs together
- **Examples**:
  - Student record: student id, name, major, gender, ..
  - Bank account: account number, name, balance, ..
  - Date: year, month, day
  - Point: x, y
- `struct` defines a new datatype.

struct Definition

- Syntax:

```
struct <struct_tag>{  struct [<struct_tag>]{
    <type> <identifier_list>;
    <type> <identifier_list>;
    <type> <identifier_list>;
    <type> <identifier_list>;
  …
    …
    };
    }<variable_list>;
```

- **Examples**

```
struct point{
  int x ;
  int y ;
};
```

```
struct Student{
  int st_id;
  char fname[100];
  char lname[100];
  int age;
}
```

Declaration of `struct` Variable

- Declaration of a variable of struct type:

```
struct <struct_type> <identifier_list>;
```

- Example1

```
struct studentRec {
  int student_idno;
  char student_name[20];
  int age;
} s1, s2;
```

```
struct studentRec {
  int student_idno;
  char student_name[20];
  int age;
};
struct studentRec s1, s2;
```

- Example2

```
struct s1 { char c; int i; } u ;
struct s2 { char c; int i; } v ;
struct s3 { char c; int i; } x ;
struct s3 y ;
```

  - The types of u , v and x are all different, but the types of x and y are the same.

`struct` Members

- Individual components of a struct type are called members (or fields)
    - can be of different types (simple, array or struct).
- Complex data structures can be formed by defining arrays of structs.
- Members of a struct type variable are accessed with direct access operator (.)
- Syntax: `<struct-variable>.<member_name>;`
- Example:

```
strcpy(s1.student_name, "Mohamed Ali");
s1.studentid = 43321313;
s1.age = 20;
printf("The student name is %s", s1.student_name);
struct point ptA;
```

`struct` Variable Initialization

- Initialization is done by specifying values of every member.

```
struct point ptA={10,20};
```

- Assignment operator copies every member of the structure
  - be careful with pointers
  - Cannot use == to compare two structure variables
- A variable of a structure type can be also initialized by any the following methods:
- Example:

```
struct date {
  int day, month , year ;
} birth_date = {31 , 12 , 1988};
struct date newyear={1, 1};
```

Nested Structures

- Let’s consider the structures:

```
struct Client{
  char name[21];
  char gender;
  int age;
  char address[21];
};
struct BankAccount{
  char name[21];
  int accNum[20];
  double balance;
  struct Client aHolder;
} ba;
ba.aHolder.age = 35;
```

- We can define the Client inside the BankAccount

```
struct BankAccount{
  char name[21];
  int accNum[20];
  double balance;
  struct{
    char name[21];
    char gender;
    int age;
    char address[21];
  } aHolder;
} b1;
ba.aHolder.age = 35;
```

- Client is not visible outside the BankAccount which makes its name optional.

Pointer to Structure

- Created the same way we create a pointer to any simple data type.

  ```
  struct date *cDatePtr,  cDate;
  ```

- We can make cDatePtr point to cDate by:

  ```
  cDatePtr = &cDate
  ```

- The pointer variable `cDatePtr` can now be used to access the member variables of `date` using the dot operator as:

  ```
  (*cDatePtr).year
  (*cDatePtr).month
  (*cDatePtr).day
  ```

- The parentheses are necessary!
  - the precedence of the dot operator `.` is higher than that of the dereferencing operator `*`

Pointer to Structure Example

```
#include <stdio.h>
#include <math.h>
struct Point{
  int x;
  int y;
};

float distance(struct Point p1, struct Point p2){
  return sqrt((p1.x-p2.x)*(p1.x-p2.x)+
              (p1.y-p2.y)*(p1.y-p2.y));
}

int main(){
  struct Point pp = {3,7};
  struct Point ppp = {-5,2};
  printf("%.2f\n", distance(pp, ppp));
  return 0;
}
```

Pointer to Structure

- Pointers are so commonly used with structures.
- C provides a special operator -> called the structure pointer or arrow operator or the indirect access operator, for accessing members of a structure variable pointed by a pointer.
- Syntax:
  `<pointer-name> -> <member-name>`
- Examples:

```
cDatePtr-> year     ⇔     (*cDatePtr).year
cDatePtr-> month    ⇔     (*cDatePtr).month
cDatePtr-> day      ⇔     (*cDatePtr).day
```

- You cannot edclare a member `x` of type `struct T` inside `struct T`
- But you can declare a member `x` of type `structT*` inside `struct T`

Pointer to Structure Example

```
#include <stdio.h>
#include <math.h>
struct Point{
  int x;
  int y;
};

float distance(struct Point *p1, struct Point *p2){
  return sqrt((p1->x-p2->x)*(p1->x-p2->x)+
              (p1->y-p2->y)*(p1->y-p2->y));
}

int main(){
  struct Point pp = {3,7};
  struct Point ppp = {-5,2};
  printf("%.2f\n", distance(&pp, &ppp));
  return 0;
}
```

Size of structure

- size of a structure is greater than or equal to the sum of the sizes of its members.
- when computer reads/writes from/to memory address
  - it reads/writes a whole word
  - a word size is determined by platform: Ex. 4 bytes in 32-bit systems
  - Self-alignment speeds up memory access to fetch/write typed data
- Alignment

  On modern processors, basic C data types has some storage constraints:
  - Variables of 8-bit length can start at any address
  - Variables of 16-bit length must start on even address
  - Variables of 32-bit length must start on an address that is divisible by 4
  - Variables of 64-bit length must start on an address that is divisible by 8
- Padding:
  - Meaningless bytes were inserted between the end of a structure member and the next

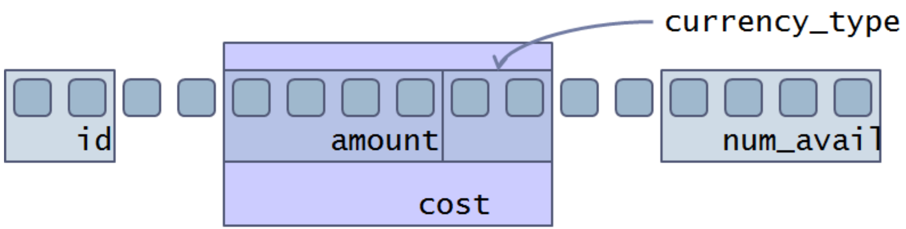
Size of structure

- `struct` alignment is based on its widest scalar member
- Address of `struct` is the same as its first member
- Padding bytes will be added between `struct` members as needed
- Trailing bytes will be added after `struct` variables as needed
- `struct` reordering is not guaranteed to shrink the size of the `struct`
- Compiler directive #pragma can be used to override the alignment:
  - Not a good idea since it slows down the execution
  - Needed when a format has to be followed
- There are too many other details and some are implementation dependent

Size of structure

- Alignment

```
struct COST {
  int amount;
  char currency_type[2]; };
struct PART {
        char id[2];
  struct COST cost;
        int num_avail; };
```

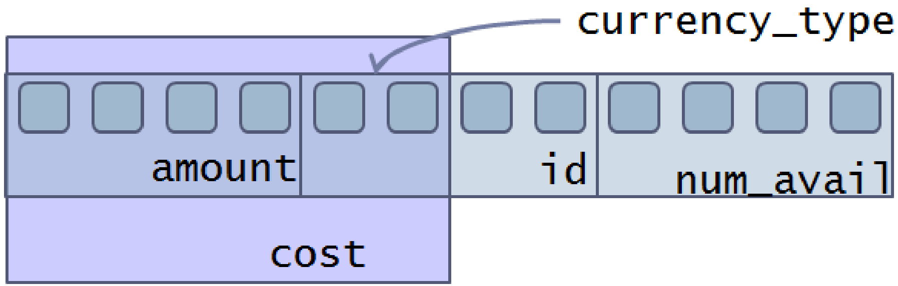


  - Size of struct Part: 16 bytes

- You can try to reduce the structure size by structure reordering or using `#pragma`

- Better:

```
struct COST {
  int amount;
  char currency_type[2]; }
struct PART {
  struct COST cost;
  char id[2];
  int num_avail; }
```



  - Size of struct Part: 12 bytes

Array of Structures

- Can create an array of structures
- Example:

```
struct studentRec {
  int student_idno;
  char *student_name;
  int age;
};
struct studentRec studentRecords[500];
```

  - `studentRecords` is an array containing 500 elements of the type `struct studentRec`.
  - Member variable inside `studentRecords` can be accessed using array subscript and dot operator: `studentRecords[0].student_name = "Mohammad";`

Example

```
#include <stdio.h>

struct Employee {/* declare a global structure type */
  int idNum; double payRate; double hours;
};
double calcNet(struct Employee *); /* function prototype */
int main() {
  struct Employee emp = {6787, 8.93, 40.5};
  double netPay;
  netPay = calcNet(&emp); /* pass an address*/
  printf("The net pay for employee %d is $%6.2f\n", emp.idNum, netPay);
  return 0;
}
/* pt is a pointer to a structure of Employee type */
double calcNet(struct Employee *pt) {
  return(pt->payRate * pt->hours);
}
```

Union

- A variable that may hold objects of different types/sizes in same memory location

```
union data{
    int idata;        d1.idata = 10;
    float fdata;      d2.fdata = 3.14F ;
    char* sdata;      d3.sdata = "hello world" ;
} d1, d2, d3;
```

- Size of union variable is equal to size of its largest element.
- Compiler does not test if the data is being read in the correct format.

```
union data d; d.idata=10; float f=d.fdata; /* will give junk */
```

- A common solution is to maintain a separate variable.

```
enum dtype {INT,FLOAT,CHAR};
struct variant {
    union data d;
    enum dtype t;
};
```

BitField

- A set of adjacent bits within a single ’word’.

  Example:

  ```
  struct flag{
    unsigned int is_color:1;
    unsigned int has_sound:1;
    unsigned int is_ntsc:1;
  } ;
  ```

- Number after the colons specifies the width in bits.
- Each variables should be declared as unsigned int Bit fields
- Portability is an issue

typedef Keyword

- Gives a type a new name

```
typedef unsigned char BYTE;
BYTE  b1, b2;
```

- Can be used to give a name to user defined data types as well

```
struct Books {
   char title[50];
   char author[50];
   char subject[100];
   int book_id;
};
typedef struct Books Book;
Book b1, b2;
```

```
typedef struct {
   char title[50];
   char author[50];
   char subject[100];
   int book_id;
} Book;

Book b1, b2;
```

Example1: Polygon (polygon.h)

```
#ifndef POLYGON
#define POLYGON
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
typedef struct {
  int x;
  int y;
} Point;
typedef struct{
  Point* points;
  int count;
} Polygon;

float distance(Point*, Point*);
Polygon* getPG();
int isParallelogram(Polygon*);

#endif
```

Example1: Polygon (polygon.c)

```
#include "polygon.h"

Polygon* getPG(){
  Polygon* pg;
  Point* p;
  int i=0;
  pg = (Polygon*)calloc(1, sizeof(Polygon));
  printf("Enter number of points:");
  scanf("%d", &(pg->count));
  p=pg->points=(Point*)calloc(pg->count, sizeof(Point));
  if (!p) return NULL;
  while (p < (pg->points)+(pg->count)){
    printf("Enter x for point p%d:", i+1);
    scanf("%d", &(p->x));
    printf("Enter y for point p%d:", i+++1);
    scanf("%d", &(p->y));
    p++;
  }
  return pg;
}
```

Example1: Polygon (polygon.c)

```
float distance(Point* p1, Point* p2){
  return sqrt((p1->x-p2->x)*(p1->x-p2->x) + (p1->y-p2->y)*(p1->y-p2->y));
}

int isParallelogram(Polygon* pg){
  if (pg->count == 4)
    if (distance(pg->points, pg->points+1) == distance(pg->points+2, pg->points+3) &&
        distance(pg->points+1, pg->points+2) == distance(pg->points+3, pg->points))
      return 1;                       /* not a good idea, why? */
  return 0;
}
```

Example1: Polygon (pgtest.c)

```
#include "polygon.h"

int main(){
  Polygon *pg1, *pg2;
  pg1 = getPG();
  printf("This polygon is %sa parallelogram.", isParallelogram(pg1)?"":"not ");
  pg2 = getPG();
  printf("This polygon is %sa parallelogram.", isParallelogram(pg2)?"":"not ");

  return 0;
}
```

data.txt

```
4          5 -3                 6 5                 1 4                 -4 0
4          -3 5                 5 6                 4 1                 -4 0
```

Example2: Matrix - revisited

```
#if !defined MAT
#define MAT

typedef struct{
  int** data;
  int rows;
  int cols;
} Matrix;

Matrix get_matrix(int, int);
void free_matrix(Matrix);     /* OR */  void free_matrix(Matrix*);
void fill_matrix(Matrix);
void print_matrix(Matrix);
Matrix transpose(Matrix);

#endif
```

Input and Output

Outline

- ❖ Introduction
- ❖ Standard files
- ❖ General files I/O
- ❖ Command-line parameters
- ❖ Error handling
- ❖ String I/O

Introduction

- C has no built-in statements for input or output
- Input and output functions are provided by the standard library `<stdio.h>`
- All input and output is performed with streams:
  - Stream: a sequence of bytes
    - text stream: consists of series of characters organized into lines ending with `'\n'`
      The standard library takes care of conversion from `"\r\n" to '\n'`
    - binary stream: consists of a series of raw bytes
  - The streams provided by standard library are buffered
- Streams are represented by the data type `FILE*`
  - FILE is a struct contains the internal state information about the connection to the file

Standard Files

- Standard input stream:
  - called `stdin`
  - normally connected to the keyboard
  - OS knows it by number 0
- Standard output stream:
  - Called `stdout`
  - normally connected to the display screen
  - OS knows it by number 1
- Standard error stream:
  - called `stderr`
  - also normally connected to the screen
  - OS knows it by number 2

Standard Files

- `int putchar(int char)`
  - Writes the character (an unsigned char) `char` to `stdout`
  - returns the character printed or `EOF` on error
- `int puts(const char *str)`
  - Writes the string `str` to `stdout` up to, but not including, the null character
  - A newline character is appended to the output
  - returns non-negative value, or `EOF` on error
- `int getchar(void)`
  - reads a character (an unsigned char) from `stdin`
  - returns `EOF` on error
- `char *gets(char *str)`
  - Reads a line from `stdin` and stores it into the string pointed to by `str`
  - It stops when either: the newline character is read or when the end-of-file is reached, whichever comes first
  - Prone to overflow problem

Standard Files

- `int scanf(const char *format, ...)`
  - Reads formatted input from `stdin`
  - Prone to overflow problem when used with strings
- `int printf(const char *format, ...)`
  - Sends formatted output to `stdout`
- `void perror(const char *str)`
  - prints a descriptive error message to `stderr`
  - string `str` is printed, followed by a colon then a space.
- What does the following code do?

```
int main ( ){
  char c ;
  while ((c=getchar())!= EOF){
    if ( c >= 'A' && c <= 'Z')
      c = c - 'A' + 'a';
      putchar(c) ;
  }
  return 0;
}
```

Standard Files

- Redirecting standard streams:
  - Provided by the operating system
  - Redirecting `stdout`: `prog > output.txt`
    and to append: `prog >> output.txt`
  - Redirecting `stderr`: `prog 2> error.txt`
    and to append: `prog 2>> error.txt`
  - Redirecting to `stdin`: `prog < input.txt`
  - Redirect the output of prog1 to the input of prog2: `prog1 | prog2`

General Stream I/O

- So far, we have read from the standard input and written to the standard output
- C allows us to read data from any text/binary files
- `FILE* fopen(char *filename,char *mode)`
    - opens file `filename` using the given `mode`
    - returns a pointer to the file stream
    - or NULL otherwise.
- `int fclose(FILE* fp)`
    - closes the stream (releases OS resources).
    - all buffers are flushed.
    - returns 0 if successful, and EOF otherwise.
    - automatically called on all open files when program terminates

`r`	For reading. File must exist
`w`	Creates empty file for writing. If file exists, it content is erased.
`a`	Appends to an existent file. Creates one if not exist.
`r+`	For reading & writing. File must exist
`w+`	Creates a file for reading & writing.
`a+`	For reading and appending

General Stream I/O

- `int getc(FILE* stream)`
  - reads a single character from the stream.
  - returns the character read or EOF on error/end of file.
  - We can implement it as follows: `#define getchar() getc(stdin)`
- `char* fgets(char *line, int maxlen, FILE* fp)`
  - reads a single line (upto maxlen characters) from the input stream (including linebreak)
  - stops when reading n-1 characters, reading \n or reaching end of file
  - returns a pointer to the character array that stores the line
  - returns NULL if end of stream.
- `int fscanf(FILE* fp, char *format, ...)`
  - similar to scanf,sscanf
  - reads items from input stream fp.
  - returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure

General Stream I/O

- `int ungetc(int ch, FILE *stream)`
  - pushes `ch` (unsigned char) onto the specified `stream` to be read again.
  - returns character that was pushed back if successful, otherwise EOF
- `int putc(int ch, FILE* fp)`
  - writes a single character `ch` to the output stream.
  - returns the character written or EOF on error.
  - we can implement it as follows: `#define putchar(c) putc(c,stdout)`
- `int fputs(char *line, FILE* stream)`
  - writes a single line to the output stream.
  - returns 0 on success, EOF otherwise.
- `int fprintf(FILE *stream, const char *format, ...)`
  - sends formatted output to a stream
  - returns total number of characters written, otherwise, a negative number is returned.

General Stream I/O

- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)`
  - reads data from the given `stream` into the array pointed to by `ptr`.
  - size: size in bytes of each element to be read
  - nmemb: number of elements, each one with a size of size bytes.
  - returns total number of elements successfully read.
    - if differs from `nmemb`, either an error has occurred or EOF was reached.
- `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)`
  - writes data from the array pointed to by `ptr` to the given `stream`
  - returns total number of elements successfully written
    - if differs from `nmemb`, it will show an error
- `void rewind(FILE *stream)`
  - sets file position to beginning of `stream`.
- `int fseek(FILE *stream, long int offset, int whence)`
  - sets file position of `stream` to `offset`
  - `offset` signifies number of bytes to seek from given `whence` position

`SEEK_SET`	Beginning of file
`SEEK_CUR`	Current position
`SEEK_END`	End of file

Example: std.h

```
typedef struct{
  int id;
  char name[25];
  float gpa;
} Student;

int save_students_data(char*, Student*, int);

Student* get_students_data(char*, int*);

Student enter_student_data();

void print_student_data(Student*);
```

Example: std.c

```
#include <stdio.h>
#include <stdlib.h>
#include "std.h"

int save_students_data(char* fn, Student* slist, int num){
  FILE* fp;
  int i;

  if ((fp = fopen(fn, "w"))){
    fwrite(&num, sizeof(int), 1, fp);
    for (i=0; i<num; i++)
      if (!fwrite(slist+i, sizeof(Student), 1, fp)) {
        perror("Problem writing to file");
        return -2;
      }
    fclose(fp);
    return 0;
  }
  perror("File could not be opened.");
  return -1;
}
```

```
if ((fp = fopen(fn, "w"))){
  fwrite(&num, sizeof(int), 1, fp);
  if (!fwrite(slist,
              sizeof(Student),
              Num,
              fp)) {
      perror("Problem writing to file");
      return -2;
  }
  fclose(fp);
  return 0;
}
```

Example: std.c (cont.)

```
Student* get_students_data(char* fn, int* num){
  FILE* fp;
  Student* result;
  int i;

  if ((fp = fopen(fn, "r"))){
    fread(num, sizeof(int), 1, fp);
    result = (Student*)calloc(*num, sizeof(Student));
    for (i=0; i<*num; i++)
      if (!fread(result+i, sizeof(Student), 1, fp)){
        perror("Problem reading from file");
        return NULL;
      }
    fclose(fp);
    return result;
  }
  perror("File could not be opened.");
  return NULL;
}
```

```
if ((fp = fopen(fn, "r"))){
  fread(&num, sizeof(int), 1, fp);
  result=(Student*)calloc(num,
                          sizeof(Student));
  if (!fread(result,
             sizeof(Student),
             num,
             fp)){
    perror("Problem reading from file");
    return NULL;
  }
  fclose(fp);
  return result;
}
```

Example: std.c (cont.)

```
Student enter_student_data(){
  Student s;
  printf("Enter student's id:");
  scanf("%d", &(s.id));
  printf("Enter student's name:");
  fgets(s.name, 24, stdin);
  printf("Enter student's GPA:");
  scanf("%f", &(s.gpa));
  return s;
}

void print_student_data(Student* s){
  printf("\n-----------------\n");
  printf("Student's id: %d\n", s->id);
  printf("Student's name: %s", s->name);
  printf("Student's GPA: %.2f\n", s->gpa);
  printf("-----------------\n");
}
```

Example: test-std.c

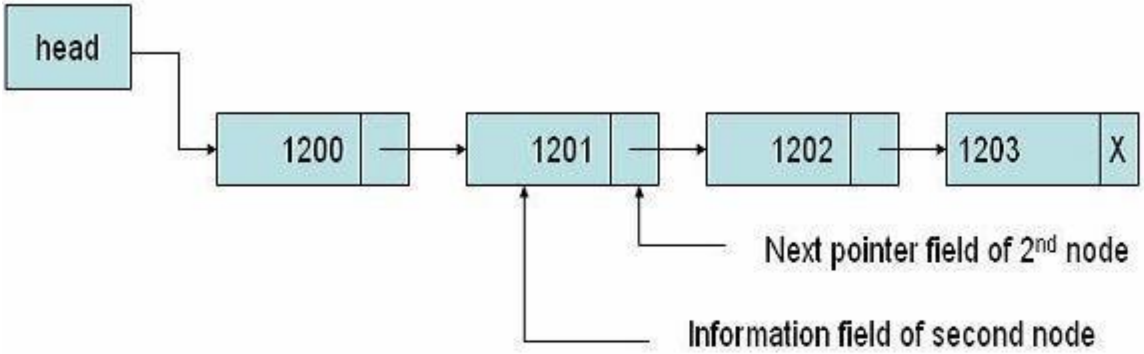
```
#include "std.h"

int main(){
  Student slist[3], *sff;
  int i, count;
  for (i=0; i<3; i++)
    slist[i] = enter_student_data();

  save_students_data("std.dat", slist, 3);

  sff = get_students_data("std.dat", &count);

  for (i=0; i<count; i++)
    print_student_data(sff+i);

  return 0;
}
```

Handling Files

- `int remove(const char *filename)`
  - deletes the given filename so that it is no longer accessible.
  - returns 0 on success and -1on failure and `errno` is set appropriately
- `int rename(const char *old_filename, const char *new_filename)`
  - causes filename referred to, by `old_filename` to be changed to `new_filename`.
  - returns 0 on success and -1on failure and `errno` is set appropriately
- How to get a file's size?
  - Use fseek with `long int ftell(FILE *stream)`
    - returns current file position of the given stream
  - ```
    FILE* f; long int size=0;
    if ((f = fopen("readme.txt"))){
      fseek(f, 0, SEEK_END);
      size = ftell(f);
      fclose(f);
    }
    ```
    ```

Command line Input

- In addition to taking input from standard input and files, you can also pass input while invoking the program.
  - so far, we have used int main() as to invoke the main function.
  - however, main function can take arguments that are populated when the program is invoked.
- `int main(int argc,char* argv[])`
  - `argc`: count of arguments.
  - `argv`: an array of pointers to each of the arguments
  - note: the arguments include the name of the program as well
  - Examples:

```
./cat a.txt b.txt
( argc = 3 , argv[0] = "cat" , argv[1] = "a.txt" and argv[2] = "b.txt" )
./cat
( argc = 1 , argv[0] = "cat" )
```

Error Handling

- No direct support for error handling
- `errno.h`
  - defines the global variable `errno`, set to zero at program startup
  - defines macros that indicate some error codes
- `char* strerror(int errnum)`
  - returns a string describing error errnum, must include `string.h`
- `stderr`
  - output stream for errors
  - assigned to a program just like `stdin` and `stdout`
  - appears on screen even if stdout is redirected
- `exit` function
  - terminates the program from any function, must include `stdlib.h`
  - argument is passed to the system
    - `EXIT_FAILURE` , `EXIT_SUCCESS`: defined in stdlib.h

Error Handling: Example

```
#include <stdio.h>
#include <errno.h>
#include <string.h>

extern int errno ;

int main () {
   FILE* pf;
   pf = fopen ("unexist.txt", "rb");
   if (pf == NULL) {

     int e = errno;
     fprintf(stderr, "Value of errno: %d\n", e);
     perror("Error printed by perror");
     fprintf(stderr, "Error opening file: %s\n", strerror(e));
   }
   else
      fclose (pf);
   return 0;
}
```

String I/O

- Instead of writing to the standard output, the formatted data can be written to or read from character arrays.
- `int sprintf(char *str, const char *format, ...)`
  - `format` specification is the same as `printf`.
  - output is written to `str` (does not check size).
  - returns number of character written or negative value on error.
- `int sscanf(const char *str, const char *format, ...)`
  - `format` specification is the same as `scanf`;
  - input is read from `str` variable.
  - returns number of items read or negative value on error.

Data Structures in C

Outline

- Linked Lists
- Binary Trees
- Stacks
- Queues
- Hash Tables

Linked Lists

- ❖ Linked List: A dynamic data structure that consists of a sequence of nodes
  - each element contains a link or more to the next node(s) in the sequence
  - Linked lists can be singly or doubly linked, linear or circular.
- ❖ Every node has a payload and a link to the next node in the list
- ❖ The start (head) of the list is maintained in a separate variable
- ❖ End of the list is indicated by NULL (sentinel).
- ❖ Example:

```
struct Node{
    void* data;
    struct Node* next;
};
struct LinkedList {
    struct Node* head;
};
```



Linked Lists: Operations

```
typedef struct Node Node;

Node* new_node(void*);

typedef struct LinkedList LinkedList;

LinkedList* new_linked_list();

void insert_at_front(LinkedList*,void*);

void insert_at_back(LinkedList*,void*);

void* remove_from_front(LinkedList*);

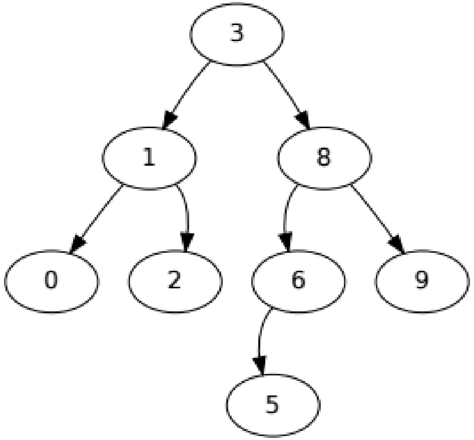
void* remove_from_back(LinkedList*);

int size(LinkedList*);

int is_empty(LinkedList*);
```

```
struct Node{
  void* data;   Node* next;
};

Node* new_node(void* data){
 Node* n=(Node*)
          calloc(1,sizeof(Node));
 n->data = data;
 return n;
}

struct LinkedList {
 Node* head;
};

LinkedList* new_linked_list(){
 LinkedList* ll=(LinkedList*)
    calloc(1,sizeof(LinkedList));
  return ll;
}
```

Linked Lists: Operations

- Iterating:
  - `for (p=head; p!=NULL; p=p->next) /* do something */`
  - `for (p=head; p->next !=NULL; p=p->next) /* do something */`
  - `for (p=head; p->next->next !=NULL; p=p->next) /* do something */`

```
int size(LinkedList* ll){
  int result = 0;
  Node* p = ll->head;
  while (p){
    p=p->next; result++;
  }
  return result;
}
```

```
int is_empty(LinkedList* ll) {
  return !ll->head;
}
```

Linked Lists: Operations - insert

```
void insert_at_front(LinkedList* ll, void* data){
  Node* n = new_node(data);
  if (!n) return;
  n->next = ll->head;
  ll->head = n;
}
```

```
void insert_at_back(LinkedList* ll, void* data){
  Node* n = new_node(data);
  if (!n) return;
  Node* p = ll->head;
  if (!p) ll->head = n;
  else {
    while (p->next) p=p->next;
    p->next = n;
  }
}
```

Linked Lists: Operations - insert

```
void insert_after_nth(LinkedList* ll, void* data, int n){
  Node* nn = new_node(data);
  if (!nn) return;
  int i=0;
  Node* p = ll->head;
  if (!p) ll->head = nn;
  else {
    while (p->next && i < n){
      p = p->next; i++;
    }
    nn->next = p->next;
    p->next = nn;
  }
}
```

Linked Lists: Operations - insert

```
void insert_in_order(LinkedList* ll, void* data, int(*comp)(void*,void*)){
  Node* n = new_node(data);
  if (!n) return;
  Node* p = ll->head;
  if (!p || comp(data, p->data)<0){
    n->next = p;
    ll->head = n;
  }
  else {
    while (p->next && comp(data, p->next->data)>0) p=p->next;
    n->next = p->next;
    p->next = n;
  }
}
```

Linked Lists: Operations - remove

```
void* remove_from_front(LinkedList*ll){
  void* result;
  Node* p = ll->head;
  if (!p) return NULL;
  result = p->data;
  ll->head = p->next;
  free(p);
  return result;
}
```

```
void* remove_from_back(LinkedList*ll){
  void* result;
  Node* p = ll->head;
  if (!p) return NULL;
  if (!(p->next)){
    result = p->data;
    ll->head = NULL;
    free(p);
  }
  else {
    while (p->next->next) p=p->next;
    result = p->next->data;
    free(p->next);
    p->next = NULL;
  }
  return result;
}
```

Linked List vs Arrays - operations

- Time complexity:
Operation	Linked List	Array
Indexing	O(n)	O(1)
Insert at front	O(1)	O(n)
Insert at back	O(n)	O(1)
Remove from front	O(1)	O(n)
Remove from back	O(n)	O(1)
- Other aspects:
Aspect	Linked List	Array
Extensibility	dynamic size	fixed size: expansion is costly
Shifting	not required	some operations (discuss)
Random access	inefficient	efficient
Sequential access	slow	fast (discuss)
Memory use	efficient	inefficient for large arrays and few data

Binary Trees

- Binary Tree: dynamic data structure where each node has at most two children
- A binary search tree is a binary tree with ordering among its children
  - all elements in the left subtree are assumed to be ”less” than the root element
  - and all elements in the right subtree are assumed to be "greater" than the root element
- Example:

```
struct tnode{
  void* data; /* payload */
  struct tnode* left;
  struct tnode* right;
};
struct tree{
  struct tnode root;
}
```



Binary Trees

- The operation on trees can be framed as recursive operations.
  - Traversal (printing, searching):
    - pre-order: root, left subtree, right subtree
    - inorder: left subtree, root, right subtree
    - post-order: right subtree, right subtree, root
- Add node:

```
struct tnode* addnode(struct tnode* root, int data){
  if (root==NULL){ /* termination condition */
    /* allocate node and return new root */
  }
  else if (data < root->data) /* recursive call */
         return addnode(root->left, data);
       else
         return addnode(root->right, data);
}
```

Stack

- A structure that stores data with restricted insertion and removal:
  - insertion occurs from the top exclusively: push
  - removal occurs from the top exclusively: pop
-
```
typedef struct Stack Stack;
Stack* new_stack(int size);
void* pop(Stack* q);
void push(Stack* q, void* data);
```
- may provide `void* top(void);` to read last (top) element without removing it

Stack as an Array

stack.h stack_ar.c test1.c

```
struct Stack{
  int capacity;
  void** buffer;
  int top;
};
```

- Stores in an array buffer (static or dynamic allocation)
- insert and remove done at end of array; need to track end
- ```
  Stack* new_stack(int size){
    Stack* result = (Stack*)calloc(1,sizeof(Stack));
    result->capacity = size;
    result->buffer = (void**)calloc(size, sizeof(void*));
    return result;
  }
  ```

```
void push(Stack* s, void* data){
  if (s->top < s->capacity)
    s->buffer[s->top++] = data;
}
```

```
void* pop(Stack* s){
  if (s->top > 0)
    return s->buffer[--(s->top)];
  else return NULL;
}
```
```

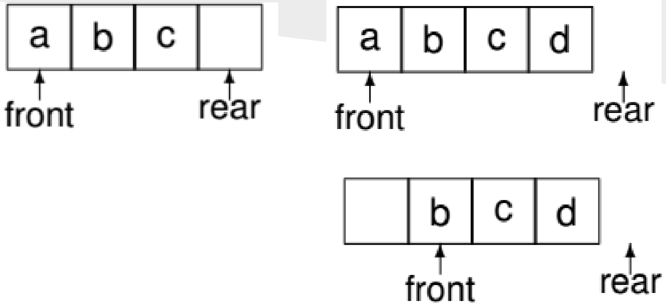
Stack as a Linked List

ll.h ll.c stack.h stack_ll.c

```
struct Stack{
  LinkedList* buffer;
};
```

- Stores in a linked list (dynamic allocation)
- "Top" is now at front of linked list (no need to track)
- ```
  Stack* new_stack(int size){ /* size is not needed */
    Stack* result = (Stack*)calloc(1,sizeof(Stack));
    result->buffer = new_linked_list();
    return result;
  }
  ```

```
void push(Stack* s, void* data){
  insert_at_front(s->buffer, data);
}
```

```
void* pop(Stack* s){
  return remove_from_front(s->buffer);
}
```
```

Queue

- Opposite of stack:
  - first in: enqueue
  - first out: dequeue
  - Read and write from opposite ends of list
- Important for:
  - UIs (event/message queues)
  - networking (Tx, Rx packet queues)
  - :
- Imposes an ordering on elements
-
```
typedef struct Queue Queue;
Queue* new_queue(int size);
void* dequeue(Queue* q);
void enqueue(Queue* q, void* data);
```

Queue as an Array

queue.h queue_ar.c test1.c

- Stores in an array buffer (static or dynamic allocation);
- Elements added to rear, removed from front
  - need to keep track of front and rear: `int front=0, rear=0;`
  - or, track the front and number of elements: `int front=0, count=0;`

```
struct Queue{
   int capacity;
   void** buffer;
   int front;
   int count;
};
```

```
Queue* new_queue(int size){
 Queue* result = (Queue*)calloc(1,sizeof(Queue));
 result->capacity = size;
 result->buffer = (void**)calloc(size,sizeof(void*));
 return result;
}
```

```
void enqueue(Queue* q, void* data){
 if (q->count < q->capacity){
  q->buffer[q->front+q->count] = data;
  q->count++;
 }
}
```

```
void* dequeue(Queue* q){
  if (q->count > 0) {
    q->count--;
    return q->buffer[q->front++];
  }
  else return NULL;
}
```

Queue as an Array

[queue.h](#) [queue_arr.c](#) [test1.c](#)

- Let us try a queue of capacity 4:
  - enqueue a, enqueue b, enqueue c, enqueue d
  - queue is now full.
  - dequeue , enqueue e: where should it go?



- Solution: use a circular (or ring) buffer
  - 'e' would go in the beginning of the array
- Need to modify enqueue and dequeue:

```
void enqueue(Queue* q, void* data){
 if (q->count < q->capacity){
  q->buffer[q->front+q->count %
                q->capacity] = data;
  q->count++;
 }
}
```

```
void* dequeue(Queue* q){
  void* result = NULL;
  if (q->count > 0) {
    q->count--;
    result=q->buffer[q->front++];
    if (q->front == q->capacity)
      q->front = 0;
  }
  return result;
}
```

Queue as a Linked List [ll.h] [ll.c] [queue.h] [queue_ll.c]

```
struct Queue{
  LinkedList* buffer;
};
```

- Stores in a linked list (dynamic allocation)
- ```
  Queue* new_queue(int size){
    /* size is not needed*/
    Queue* result = (Queue*)calloc(1,sizeof(Queue));
    result->buffer = new_linked_list();
    return result;
  }
  ```

```
void enqueue(Queue* q, void* data){
  insert_at_back(q->buffer, data);
}
```

```
void* dequeue(Queue* q){
  return remove_from_front(q->buffer);
}
```
```

Example: Postfix Evaluator

- Stacks and queues allow us to design a simple expression evaluator
- Prefix, infix, postfix notation:
  - operator before, between, and after operands, respectively
  - Infix
    - A + B
    - A * B - C
    - (A + B) * (C - D)
  - Prefix
    - + A B
    - - * A B C
    - * + A B - C D
  - Postfix
    - A B +
    - A B * C
    - A B + C D - *
  - Infix more natural to write, postfix easier to evaluate

Example: Postfix Evaluator

```
float pf_eval(char* exp){
  Stack* S = new_stack(0);
  while (*exp){
    if (isdigit(*exp) || *exp=='.'){
      float* num; num = (float*)malloc(sizeof(float));
      sscanf(exp, "%f", num);
      push(S, num);
      while(isdigit(*exp) || *exp=='.') exp++; exp--;
    }
    else if (*exp!= ' ') {
      float num1 = *(float*)pop(S), num2 = *(float*)pop(S);
      float* num; num = (float*)malloc(sizeof(float));
      switch (*exp){
        case '+': *num = num1+num2; break;
        case '-': *num = num1-num2; break;
        case '*': *num = num1*num2; break;
        case '/': *num = num1/num2; break;
      }
      push(S, num);
    }
    exp++;
  }
  return *(float*)pop(S);
}
```

Hash Table

- Hash tables (hashmaps) an efficient data structure for storing dynamic data.
- commonly implemented as an array of linked lists (hash tables with chaining).
- Each data item is associated with a key that determines its location.
  - Hash functions are used to generate an evenly distributed hash value.
  - A hash collision is said to occur when two items have the same hash value.
  - Items with the same hash keys are chained
  - Retrieving an item is O(1) operation.
- Hash function: map its input into a finite range: hash value, hash code.
  - The hash value should ideally have uniform distribution. why?
  - Other uses of hash functions: cryptography, caches (computers/internet), bloom filters etc.
  - Hash function types:
    - Division type
    - Multiplication type
    - Other ways to avoid collision: linear probing, double hashing.

Hash Table

- ```
  struct Pair{
    char* key;
    void* data;
  };
  ```
- ```
  struct HashTable{
    LinkedList* buckets;
    int capacity;
  };
  ```
- ```
  unsigned long int hash(char* key);
  HashTable* new_hashtable(int size);
  int is_empty_ht(HashTable* ht);
  int length(HashTable* ht);
  void insert(HashTable* ht, Pair* p);
  void* remove(char* key);
  void* retrieve(char* key);
  ```
  ```

Hash Table

```
HashTable* new_hashtable(int size){
  HashTable* result = (HashTable*)calloc(1, sizeof(HashTable));
  result->size = size;
  result->buckets = (LinkedList*)calloc(size, sizeof(LinkedList));
  return result;
}

int is_empty_ht(HashTable* ht){
  int i;
  for (i=0; i < ht->size; i++)
    if (!is_empty(&(ht->buckets[i])))
      return 0;
  return 1;
}

int length(HashTable* ht){
  int i, result=0;
  for (i=0; i < ht->size; i++)
    result += size(&(ht->buckets[i]));
  return result;
}
```

Hash Table

```
unsigned long int hash(char* key){
  /* any good hashing algorithm */
  const int MULTIPLIER = 31;
  unsigned long int hashval = 0;
  while (*key)
    hashval = hashval * multiplier + *key++;
  return hashval;
}

void insert(HashTable* ht, Pair* p){
  if (retrieve(ht, p->key)) return NULL;
  int index = hash(p->key) % ht->capacity;
  insert_at_front(&(ht->buckets[index]));
}

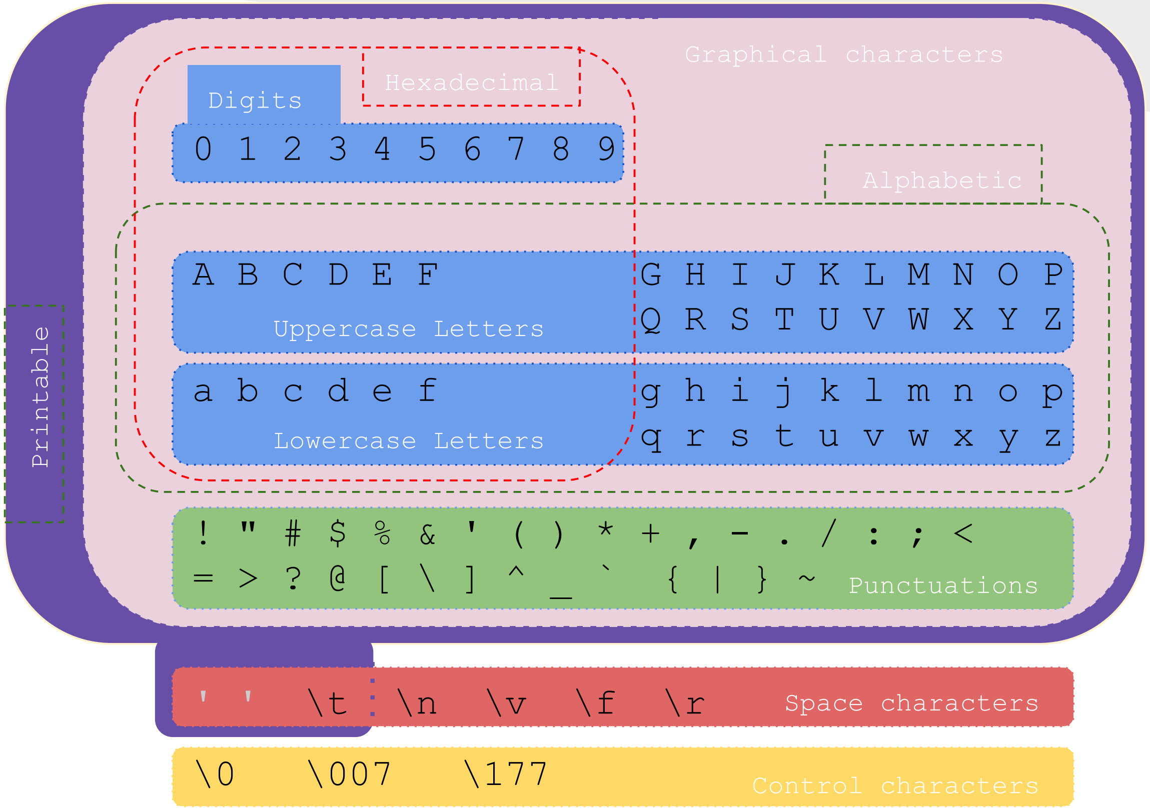
void* remove(char* key){
  /* since we do not have a ready supporting ll function we'll go low level */
}

void* retrieve(char* key){
  /* since we do not have a ready supporting ll function we'll go low level */
}
```

CSC215 Lecture

Standard Library in C

Outline

- Introduction
- stdio.h
- stdlib.h
- ctype.h
- stdarg.h
- math.h
- string.h
- assert.h
- errno.h
- time.h

Introduction

- Standard library:
  - type definitions
  - variable declarations
  - constant and macro definitions
  - functions

- Description and usage information can be obtained from `man` pages on unix-like OS or the web
  - section 3
  - on unix and unix-like OS: in the terminal type: man [<section>] <library_function_name>⏎
  - Many websites host copies of the man pages: Die, HE , MAN7 , …

- List of standard library header files:

```
assert.h    ctype.h   errno.h   float.h   limits.h   locale.h   math.h
    setjmp.h
signal.h    stdarg.h  stddef.h  stdio.h   stdlib.h   string.h   time.h
```

Library `stdio.h`

- Types
  - `size_t`
  - `FILE`

- Constants
  - `NULL`
  - `EOF`
  - `SEEK_CUR`
    `SEEK_END`
    `SEEK_SET`
  - `stderr`
    `stdin`
    `stdout`

- Functions
  - `FILE *fopen(const char *, const char *)`
  - `int fclose(FILE *)`
  - `int fflush(FILE *)`

  - `int getchar(void)`
  - `char *gets(char *)`
  - `int scanf(const char *, ...)`
  - `int putchar(int char)`
  - `int puts(const char *)`
  - `int printf(const char *, ...)`

  - `int fgetc(FILE *)`
  - `int ungetc(int char, FILE *stream)`
  - `char *fgets(char *, int , FILE *)`
  - `int fscanf(FILE *, const char *, ...)`
  - `int fputc(int, FILE *)`
  - `int fputs(const char *, FILE *)`
  - `int fprintf(FILE *, const char *, ...)`

Library `stdio.h`

- Functions (cont.)
  - `size_t fread(void *, size_t, size_t, FILE *)`
  - `size_t fwrite(const void *, size_t, size_t, FILE *)`
    `---------------------------------------------------`
  - `void rewind(FILE *)`
  - `int fseek(FILE *, long int, int)`
  - `int fgetpos(FILE *, fpos_t *)`
  - `int fsetpos(FILE *, const fpos_t *)`
  - `long int ftell(FILE *)`
  - `---------------------------------------------------`
  - `int remove(const char *)`
  - `int rename(const char *, const char *)`
    `---------------------------------------------------`

Library `stdlib.h`

- Types
  - `size_t`

- Constants
  - `NULL`
  - `EXIT_FAILURE`
    `EXIT_SUCCESS`
  - `RAND_MAX`

- Functions (cont.)
  - `void *malloc(size_t)`
  - `void *calloc(size_t, size_t)`
  - `void *realloc(void *, size_t)`
  - `void free(void *)`

  - `double atof(const char *)`
  - `int atoi(const char *)`
  - `long int atol(const char *)`
  - `double strtod(const char *, char **)`
  - `long int strtol(const char *, char **, int)`
  - `unsigned long int strtoul(const char *, char **, int)`

  - `void abort(void)`
  - `void exit(int)`
  - `int atexit(void (*func)(void))`
  - `int system(const char *string)`

  - `int abs(int x)`
  - `long int labs(long int x)`

Library `stdlib.h`

- Functions (cont.)
  - `int rand(void)`
  - `void srand(unsigned int seed)`

    `----------------------------------------------------`
  - ```
    void *bsearch(const void *, const void *, size_t, size_t,
                  int (*compar)(const void *, const void *))
    ```
  - `void qsort(void *, size_t, size_t, int (*compar)(const void *, const void*))`
    ```

Library `ctype.h`

- Functions (cont.)
  - `int isalnum(int c)`
  - `int isalpha(int c)`
  - `int iscntrl(int c)`
  - `int isdigit(int c)`
  - `int isgraph(int c)`
  - `int islower(int c)`
  - `int isprint(int c)`
  - `int ispunct(int c)`
  - `int isspace(int c)`
  - `int isupper(int c)`
  - `int isxdigit(int c)`

  - `int tolower(int c)`
  - `int toupper(int c)`



Library `stdarg.h`

- Types
  - `va_list`
- Macros
  - `void va_start(va_list, last_arg)`
  - `type va_arg(va_list, type)`
  - `void va_end(va_list)`
  - `void va_copy(va_list, va_list)`

Optional Parameters

- C permits functions to have optional parameters
- Syntax: <returntype> <name>(<paramslist>, …)
  - … indicates that further parameters can be passed, must be listed only after the required parameters
  - since you specify the parameters as …, you do not know their names!
- How to use these additional parameters when they are passed?
  - `stdarg.h` file contains the definition of `va_list` (variable argument list)
  - declare a variable of type `va_list`
  - use the macro `va_start` which initializes your variable to the first of the optional params
  - use the function `va_arg` which returns the next argument

Optional Parameters

- Example:

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

int sum(int, ...);

int main(){
  printf("Sum of 15 and 56 = %d\n",  sum(2, 15, 56) );
  return 0;
}

int sum(int num_args, ...){
  int val = 0;
  va_list ap;
  int i;
  va_start(ap, num_args);
  for(i = 0; i < num_args; i++)
    val += va_arg(ap, int);
  va_end(ap);
  return val;
}
```

Library `math.h`

- Arithmetic functions
  - `double fabs(double )` $|x|$
  - `double ceil(double )` $\lceil x \rceil$
  - `double floor(double )` $\lfloor x \rfloor$
  - `double fmod(double , double )` $x \% y$
  - `double modf(double , double *)` $x-\lfloor x \rfloor$, $\lfloor x \rfloor$
- Exponential functions
  - `double pow(double , double )` x^y
  - `double sqrt(double )` $\sqrt{x}$
  - `double exp(double )` e^x
  - `double ldexp(double , int )` $x.2^y$
  - `double log(double )` $\log_e x$
  - `double log10(double )` $\log_{10} x$
- Trigonometric functions
  - `double sin(double )` $\sin(x)$
  - `double cos(double )` $\cos(x)$
  - `double asin(double )` $\sin^{-1}(x)$
  - `double acos(double )` $\cos^{-1}(x)$
  - `double atan(double )` $\tan^{-1}(x)$

- All functions take and yields double precision floating point values.
- Trigonometric functions deals with input and output angles in radians.

Example

```c
#include <stdio.h>
#include <math.h>
const double PI = acos(-1);
const double E = exp(1);
int main(){
  double buf;
  printf("pi = %f\n", PI);
  printf("e = %f\n", E);

  printf("Absolute: |%f| = %f \n", -1.3, fabs(-1.3));
  printf("Floor: %f >= %f\n", -1.3, floor(-1.3));
  printf("Ceiling: %f <= %f\n", -1.3, ceil(-1.3));
  printf("F Mod: %f mod %f = %f\n", 18.9, 9.2, fmod(19.9, 9.2));
  printf("Split: %f into %f and ", 427.049, modf(427.049, &buf));
  printf("%f\n", buf);
  printf("Floor: %f >= %f = \n", -1.3, floor(1.0/3+1.0/3+1.0/3));

  printf("Fifth root of : %f is %f\n", 1.3, pow(1.3, 1.0/5));
  printf("Square root of : %f is %f\n", 112.7, sqrt(112.7));
  printf("%fx2^%d = %f\n", 5.2, 7, ldexp(5.2, 7));
  printf("Loge %f\t= Loge 10\tx Log10 %f\n", 5.2, 5.2);
  printf("%f\t= %f\tx %f\n", log(5.2), log(10), log10(5.2));

  printf("Sin(%d deg) = Sin(%dxPI/180 rad) = %f\n", 45, 45, sin(45*PI/180));
  return 0;
}
```

```
pi = 3.141593
e = 2.718282

Absolute: |-1.300000| = 1.300000

Floor: -1.300000 >= -2.000000
Ceiling: -1.300000 <= -1.000000

F Mod: 18.900000 mod 9.200000 = 1.500000
Split: 427.049000 into 0.049000 and 427.000000
Floor: -1.300000 >= 1.000000 =

Fifth root of : 1.300000 is 1.053874
Square root of : 112.700000 is 10.616026

5.200000x2^7 = 665.600000
Loge 5.200000    = Loge 10       x Log10 5.200000
1.648659         = 2.302585      x 0.716003

Sin(45 deg) = Sin(45xPI/180) = 0.707107
```

Library `string.h`

- Memory functions
  - `int memcmp(const void *, const void *, size_t)`
  - `void *memchr(const void *, int, size_t)`
  - `void *memcpy(void *, const void *, size_t)`
  - `void *memset(void *, int, size_t )`
- String functions
  - `size_t strlen(const char *)`
  - `char *strcat(char *, const char *)`
  - `char *strncat(char *, const char *, size_t )`
  - `char *strcpy(char *, const char *)`
  - `char *strncpy(char *, const char *, size_t )`
  - `int strcmp(const char *, const char *)`
  - `int strncmp(const char *, const char *, size_t )`
  - `char *strchr(const char *, int)`
  - `char *strrchr(const char *, int)`
  - `char *strstr(const char *, const char *)`
  - `char *strpbrk(const char *, const char *)`
  - `size_t strspn(const char *, const char *)`
  - `size_t strcspn(const char *, const char *)`
  - `char *strtok(char *, const char *)`

- In coping functions, the first parameter is the destination and the second is the source.
- In search functions, first parameter is the haystack (text) and the second is the needle (pattern).

Example

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

int main(){
  int* pArr = (int*)malloc(10*sizeof(int));
  char sentence[255], *word;
  memset(pArr, -100, 10*sizeof(int));
  printf("pArr[8]=%d\n", (char)pArr[8]);
  int iArr[] = {-3, 5, 0, 12, -8, 27};
  memcpy(pArr, iArr, 6*sizeof(int));
  printf("The 2 arrays are%s equal\n",memcmp(pArr,iArr,6*sizeof(int))?" not":"");
  int* ind = (int*)memchr(pArr, 12, 6*sizeof(int));
  printf("%d exist at index %d\n", 12, (int)(ind-pArr));

  char* name = "Adam";
  printf("Length of string %s is %d\n", name, (int)strlen(name));
  sprintf(sentence, "Length of string %s is %d\n", name, (int)strlen(name));
  word = strtok(sentence, ", ");
  do {
    printf("%s\n", word);
  } while (word = strtok(NULL, ", "));

  return 0;
}
```

```
pArr[8]=-100
The 2 arrays are equal
12 exist at index 3
Length of string Adam is 4
Length
of
string
Adam
is
4
```

Libraries: `assert.h`, `errno.h` and `time.h`

- Macro of assert.h
  - `void assert(int expression)`
- Macro of errno.h
  - `extern int errno`
- time.h
  - `clock_t`
  - `time_t`
  - `struct tm`
- Functions of time.h
  - `clock_t clock()`
  - `time_t time(time_t*)`
  - `double difftime(time_t, time_t)`
  - `time_t mktime(struct tm*)`
  - `char* asctime(const struct tm*)`
  - `char* ctime(const time_t)`
  - `struct tm* gmtime(const time_t)`
  - `struct tm* localtime(const time_t )`
  - `size_t strftime(char* , size_t , const char* , const struct tm* )`

```
struct tm {
  int tm_sec;     /* Seconds (0-60) */
  int tm_min;     /* Minutes (0-59) */
  int tm_hour;    /* Hours (0-23) */
  int tm_mday;    /* Day of the month (1-31) */
  int tm_mon;     /* Month (0-11) */
  int tm_year;    /* Year - 1900 */
  int tm_wday;    /* Day of week (0-6, Sunday=0) */
  int tm_yday;    /* Day in year (0-365,1 Jan=0) */
  int tm_isdst;   /* Daylight saving time */
};
```

```
Unix time epoch:
1970, Jan, 1 00:00:00 UTC
```

Example

```c
#include <stdio.h>
#include <time.h>
int main(){
  time_t rawtime;
  struct tm * timeinfo;
  char buffer [80];

  time(&rawtime);
  printf("%s\n", ctime(&rawtime));

  timeinfo = localtime(&rawtime);
  printf("%s\n", asctime(timeinfo));

  strftime(buffer,80,"Now it's %y/%m/%d.",timeinfo);
  puts(buffer);
  strftime(buffer,80,"Now it's %Y/%m/%d.",timeinfo);
  puts(buffer);
  return 0;
}
```

```
Tue Apr 18 04:55:50 2017
Tue Apr 18 04:55:50 2017
Now it's 17/04/18.
Now it's 2017/04/18.
```

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

int main(){
  clock_t start_t, end_t;
  float total_t;
  int i;
  start_t = clock();
  printf("Starting @ start_t = %ld\n", start_t);
  printf("Run a big loop\n", start_t);
  for(i=0; i< 10000000; i++) { }
  end_t = clock();
  printf("Ending @ end_t = %ld\n", end_t);
  total_t=1000*(float)(end_t-start_t)/CLOCKS_PER_SEC;
  printf("Total CPU time: %f ms\n",total_t );
  return(0);
}
```

```
Starting @ start_t = 7865
Run a big loop
Ending @ end_t = 7915
Total CPU time: 0.050000 ms
```

Advanced Pointers

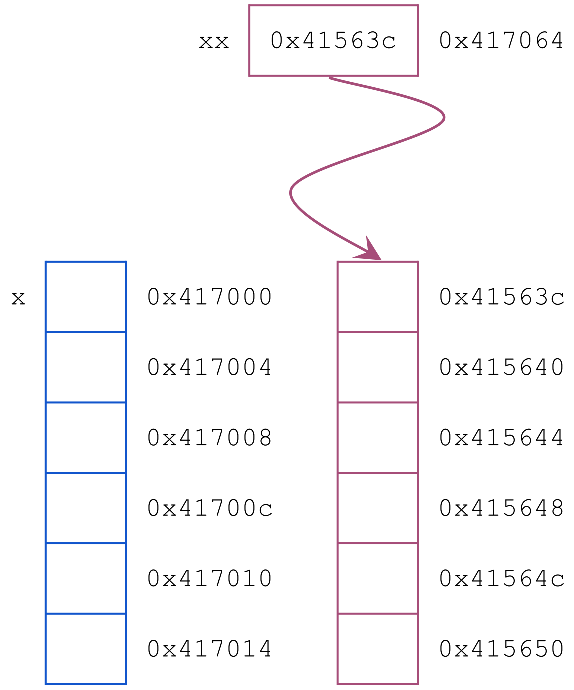
Outline

- Pointer to Pointer
  - Pointer Array
  - Strings Array
  - Multidimensional Array
- void Pointers
- Incomplete Types
- Pointer to Function

Array vs. Pointer

```
#include <stdio.h>
int main() {
  int x[5];
  int* xx = (int*)malloc(5*sizeof(int));

  printf("%p\n", x);
  printf("%p\n", x+1);
  printf("%p\n", &x);
  printf("%p\n", &x+1);
  printf("%d\n", (int)sizeof(x));
  printf("==========\n");
  printf("%p\n", xx);
  printf("%p\n", xx+1);
  printf("%p\n", &xx);
  printf("%p\n", &xx+1);
  printf("%d\n", (int)sizeof(xx));
  return 0;
}
```

```
0x417000
0x417004
0x417000
0x417014
20
==========
0x41563c
0x415640
0x417064
0x417068
4
```



Pointer to Pointer

- Pointer represents address to variable in memory
- Address stores pointer to a variable is also a data in memory and has an address
- The address of the pointer can be stored in another pointer
- Example:

```
int n = 3;
int *pn = &n; /* pointer to n */
int **ppn = &pn; /* pointer to address of n */
```

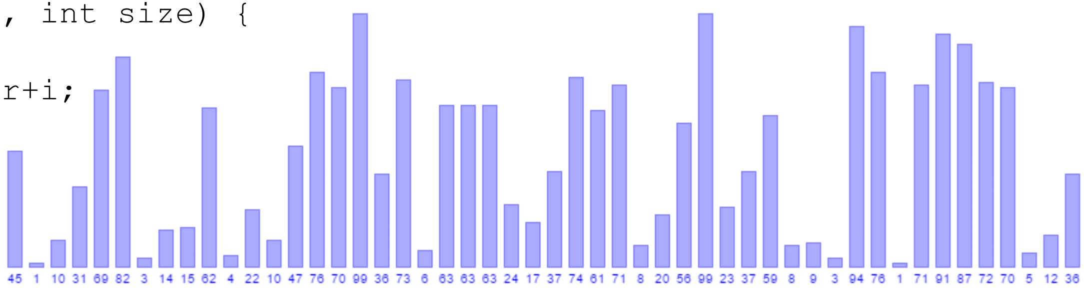
- Many uses in C: pointer arrays, string arrays, multidimensional arrays

Pointer Arrays Example

- Assume we have an array `int arr [20]` that contains some numbers
  `int arr[20]= {73,59,8,82,48,82,84,94,54,5,28,90,83,55,2,67,16,79,6,52};`
- Want to have a sorted version of the array, but not modify arr
- Declare a pointer array: `int* sarr[20]` containing pointers to elements of arr and sort the pointers instead of the numbers themselves
- Good approach for sorting arrays whose elements are very large (like strings)
- Example: insert sort
  - `void shift _element(int* sarr[], int i)`
  - `void insert_sort(int arr[], int* sarr[], int size)`

Pointer Arrays Example

```
#include <stdio.h>
void shift_element (int* sarr[], int i) {
  int* p2i;
  for (p2i = sarr[i]; i > 0 && *sarr[i-1] > *p2i; i--)
    sarr[i] = sarr[i-1];
  sarr[i] = p2i;
}
void insert_sort(int arr[], int* sarr[], int size) {
  int i;
  for (i=0; i < size; i++) sarr[i] = arr+i;
  for (i=1; i < size; i++)
    if (*sarr[i] < *sarr[i-1])
      shift_element(sarr, i);
}
int main(){
  int i, arr[20]={73,59,8,82,48,82,84,94,54,5,28,90,83,55,2,67,16,79,6,52}, *sarr[20];
  insert_sort(arr, sarr, 20);
  for (i = 0; i < 20; i++) printf("%d\t", *(sarr[i]));
  return 0;
}
```



String Array Example

- An array of strings, each stored as a pointer to an array of chars
  - each string may be of different length

```
char word1[] = "hello";    /* length = 6 */
char word2[] = "goodbye";  /* length = 8 */
char word3[] = "welcome!"; /* length = 9 */
char* str_arr[] = {word1, word2, word3} ;
```

- Note that str_arr contains only pointers, not the characters themselves!

Multidimensional Arrays

- C permits multidimensional arrays specified using [ ] brackets notation:

```
int world[20][30]; /* a 20x30 2-D array of integers */
```

- Higher dimensions are also possible:

```
char big_matrix[15][7][35][4]; /* what are the dimensions of this?
                                /* what is the size of big_matrix? */
```

- Multidimensional arrays are rectangular, while pointer arrays can be of any shape
- See: Lecture 05, Lab 05, Lecture 07

void Pointers

- C does not allow declaring or using void variables.
- void can be used only as return type or parameter of a function
- C allows void pointers
  - What are some scenarios where you want to pass void pointers?
- void pointers can be used to point to any data type

```
int x; void* px = &x; /* points to int */
float f; void* pf = &f; /* points to float */
```

- void pointers cannot be dereferenced
  - The pointers should always be cast before dereferencing

```
int x=5; void* px=&x;
printf("%d",*px); /* warning: dereferencing 'void *' pointer
                     error: invalid use of void expression */
printf("%d",*(int*)px); /* valid */
```

Incomplete types

- Types are partitioned into:
  - object types (types that fully describe objects)
    Example:
    - `float x;`
    - `char word[21];`
    - `struct Point (int x, int y};`
  - function types (types that describe functions)
    - characterized by the function's return type and the number and types of its parameters
  - incomplete types (types that describe objects but lack information needed to determine their sizes)
    - A struct with unspecified members: Ex. `struct Pixel;`
    - A union with unspecified members: Ex. `union Identifier;`
    - An array with unspecified length: Ex. `float[]`

- A pointer type may be derived from:
  - an object type
  - a function type, or
  - an incomplete type

Pointer to Incomplete Types

- Members of a struct must be of a complete type
- What if struct member is needed to be of the same struct type?

```
struct Person{
  char* name;
  int age;
  struct Person parent; /* error, struct Person is not complete yet */
};
```

- Pointers may point to incomplete types

```
struct Person{
  char* name;
  int age;
  struct Person* parent; /* valid */
}
```

- Good news for linked lists!

Function Pointers

- Functions of running program are stored in a certain space in the main-memory
- In some programming languages, functions are first class variables (can be passed to functions, returned from functions etc.).
- In C, function itself is not a variable
  - but it is possible to declare pointer to functions.
- Function pointer is a pointer which stores the address of a function
  - What are some scenarios where you want to pass pointers to functions?
- Declaration examples:

```
int (*fp1)(int)
int (*fp2)(void* ,void*)
int (*fp3)(float, char, char) = NULL;
```

- Function pointers can be assigned, passed to/from functions, placed in arrays etc.

Function Pointers

- Typedef Syntax:

```
typedef <func_return_type> (*<type_name>)(<list_of_param_types>);
```

- Declaration Syntax:

```
<func_return_type> (*<func_ptr_name>)(<list_of_param_types>); /* or */
<type_name> <func_ptr_name>;
```

- Assignment Syntax:

```
<func_ptr_name> = &<func_name>; /* or */
<func_ptr_name> = <func_name>;  /* allowed as well */
```

- Calling Syntax:

```
(*<func_ptr_name>)(<list_of_arguments>); /* or */
<func_ptr_name>(<list_of_arguments>);    /* allowed as well */
```

- Example:

```
void print_sqrt(int x){
       printf("%.2f\", sqrt(x));
     }
```

```
/* use */ void (*func)(int);
func = &print_sqrt;

(*func)(25);
```

Function Pointers Examples

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

int f1(float a){
  return (int)ceil(a);
}

int f2(float a){
  return (int)a;
}

int main(){
  int (*func)(float);
  float  f;
  scanf("%f", &f);
  func = (f - (int)f >= 0.5)? &f1:&f2; /* or f1:f2 */
  printf("Rounding of %f is %d\n", f, *func(f) /* or func(f) */);
  return 0;
}
```

```
> test
3.7
Rounding of 3.70 is 4
> test
3.3
Rounding of 3.30 is 3
```

```c
typedef int(*Fun)(float);
Fun func;
```

Function Pointers: Callbacks

- Definition: Callback is a piece of executable code passed to functions.
- In C, callbacks are implemented by passing function pointers.
- Example:

```
void qsort(void* arr, int num,int size,int (*fp)(void* pa, void* pb))
```

  - `qsort()` function from the standard library can be used to sort an array of any datatype.
  - How does it do that? Callbacks.
    - `qsort()` calls a function whenever a comparison needs to be done.
    - the function takes two arguments and returns (<0 , 0 , >0) depending on the relative order of the two items.

```
    int a rr [ ] ={ 1 0 , 9 , 8 , 1 , 2 , 3 , 5 };
    int asc ( void* pa , void* pb ){
      return ( *(int*)pa - *(int*)pb ) ;
}
int desc ( void* pa , void* pb ){
  return ( *(int*)pb - *(int*)pa ) ;
}
qsort(arr, sizeof(arr)/sizeof(int), sizeof(int), asc);
     qsort(arr, sizeof(arr)/sizeof(int), sizeof(int), desc);
```

Misselaineous Topics

Outline

- Endianness
- const Keyword
- Comma operator
- Static Functions
- NAN and Infinity
- Threads and Processes
- Interprocessor Communications

Endianness

- The order in which bytes of a multi-byte variable are arranged when stored in memory
  - can be extended to transmission
- Common formats:
  - Big-endian:
    most significant byte (contains most significant bit) is stored first (at lowest address)
  - Little-endian:
    least significant byte (contains least significant bit) is stored first (at lowest address)
- Bits Endianness:
  - Order of bits within a byte. Affects bit fields but not bitwise operations
- Can be checked in several ways

const Keyword

- Pointer to a const: const <type> * <ptr>
  - defines a pointer ptr that points to types of type
  - const modifier means that the values stored in the pointee cannot be changed

```
int i = 100;    const int* pi = &i;
     /* *pi = 200; <- won't compile */                pi++;
```

- const pointer
  - you can change the value of the pointee
  - but you can't make the pointer points to a different variable or memory location

```
int i = 100;     int* const pi = &i;
     *pi = 200;                                                     /*
pi++; <- won't compile */
```

- const Pointer to a const
  - a constant pointer to a constant variable
  - you can NOT change neither where the pointer points nor the value of the pointee

```
int i = 100;    const int* const pi = &i;
/* *pi = 200; <- won't compile */           /* pi++; <- won't compile */
```

const Keyword

- What is the meaning of:
  - ```
    /* pointer to const function -- has no meaning */
    int (const *func)(int);
    ```
  - ```
    /* const pointer to function. Allowed, must be initialized.*/
    int (*const func)(int) = ff;
    ```
  - ```
    /* pointer to function returning pointer to const */
    void const *(*func)(int);
    ```
  - ```
    /* const pointer to function returning pointer to const. */
    void const *(*const func)(int) = ff.
    ```

The comma Operator

- The comma operator:
  - combines the two expressions either side of it into one
  - evaluating them both in left-to-right order
  - the value of the right-hand side is returned as the value of the whole expression
  - (expr1, expr2) is like { expr1; expr2; } but you can use the result of expr2
  - Not recommended, and it is easy to abuse

Example:

```
#include <stdio.h>
int main(){
  int x, y;
  x = 1, 2;
  y = (3,4);
  printf( "%d %d\n", x, y );
}
```

Common in for statements:

```
for (low = 0, high = 100; low < high; low++, high--) {
  /* do something with low and high and put new values in newlow and newhigh */
```

Static Functions

- When a function’s definition prefixed with static keyword it is called a static function
- Have no effect if the program consists of one source file
- A static function is not visible outside of its translation unit:
  - the object file it is compiled into
  - making a function static limits its scope
  - think of a static function as being "private" to its *.c file (and its included files)
- Useful scenario:
  - a program of several files that you include in your main file
  - two of them have a function that is only used internally for convenience called add(int a, b)
  - the compiler would easily create object files for those two modules
  - but the linker will throw an error, because it finds two functions with the same name and it does not know which one it should use (even if there's nothing to link, because they aren't used somewhere else but in it's own file)

NAN and INFINITY

- IEEE 754 FP numbers can represent +∞, -∞, and NaN (not a number).
- Not supported in C89
- In C99:

```
int main(){
  double x = 0/0.;
  double y = 1/0.;
  double z = -1./0.;
  printf("%lf\n", x);
  printf("%lf\n", y);
  printf("%lf\n", z);
  printf("It is %sNAN\n", x!=x?"":"not ");
  printf("It is %sPositive Infinity\n", y==1/0.?"":"not ");
  printf("It is %sNegative Infinity\n", z==-1/0.?"":"not ");
  return 0;
}
```

- Some implementations of math.h defines constants INFINITY and NAN

Threads and processes

- Processes
  - Multiple simultaneous programs
  - Independent memory space
  - Independent open file-descriptors

  In linux/unix process can be forked to sub processes each is a clone of the original the continues execution either from the forking point or from the beginning of the program.

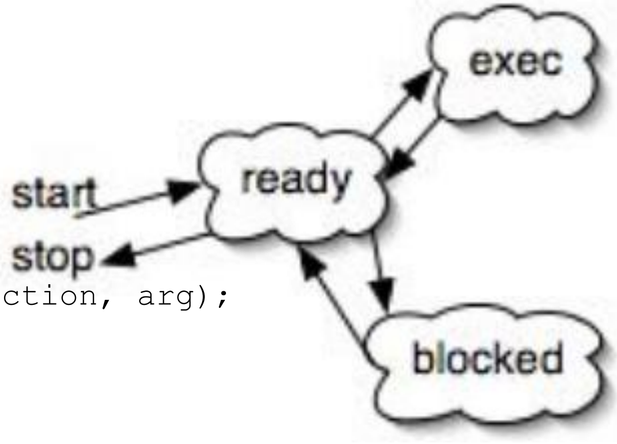
- Threads
  - Multiple simultaneous functions
  - Shared memory space
  - Shared open file-descriptors

  Examples:

  - Web browser tabs (not in google chrome)
  - GUI

Threading

- Shared memory:
  - One copy of the heap
  - One copy of the code
  - Multiple stacks
- Life cycle:
  - #include <pthread.h>
  - Define a worker function:
    - void *foo(void *args){ }
  - Initialize pthread attr t
    - pthread_attr t attr;
    - pthread_attr init(attr);
  - Create a thread
    - pthread_t thread;
    - pthread create(&thread, &attr, worker function, arg);
  - Exit current thread
    - pthread_exit(status)



Threading: Example

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 5
void *print_hello(void *threadid){
  long tid = (long)threadid;
  printf("Hello World! It’s me, thread #%ld!\n", tid);
  pthread_exit(NULL);
}
int main(int argc, char *argv[]){
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  for(t = 0; t < NUM_THREADS; t++) {
    printf("In main: creating thread %ld\n", t);
    rc = pthread_create(threads + t, NULL, print_hello, (void *)t);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }
  return 0;
}
```

Threading: Example

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 5
void* print_hello(void* threadid){
  long tid = (long)threadid;
  printf("Hello World! It’s me, thread #%ld!\n", tid);
  pthread_exit(NULL);
}
int main(int argc, char *argv[]){
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  for(t = 0; t < NUM_THREADS; t++) {
    printf("In main: creating thread %ld\n", t);
    rc = pthread_create(threads + t, NULL, print_hello, (void *)t);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }
  /* wait for all threads to complete */
  for (t = 0; t < NUM_THREADS; t++)
    pthread_join(threads[t], NULL);
  return 0;
}
```

Interprocess Communication

- Each process has its own address space
  - individual processes cannot communicate through program memory unlike threads
- Interprocess communication:
  - Linux/Unix and Windows provide several ways to allow communications:
    - Signals
    - Pipes
    - Sockets
    - RPC
    - clipboard
    - shared memory (linux) and File mapping (windows)
  - Linux/Unix provides
    - FIFO queues
    - semaphores
  - Windows provides:
    - DDE, COM and DCOM
    - Data copy
    - Mailslot