Joanna Klukowska
joannakl@cs.nyu.edu

Lecture 2: Introduction to C Programming Language

Notes include some materials provided by Andrew Case, Jinyang Li, Mohamed Zahran, and the textbooks.

Reading materials Chapters 1-6 in The C Programming Language, by B.W. Kernighan and Dennis M. Ritchie
Section 1.2 and Aside on page 4 in Computer Systems, A Programmer's Perspective by R.E. Bryant and D.R. O'Hallaron

Contents

1 Intro to C and Unix/Linux **3**
1.1 Why C? . . . 3
1.2 C vs. Java . . . 3
1.3 Software Development Process (Not Only in C) . . . 4
1.4 Basic Unix Commands . . . 4

2 First C Program and Stages of Compilation **6**
2.1 Writing and running hello world in C . . . 6
2.2 Hello world line by line . . . 7
2.3 What really happens when we compile a program? . . . 8

3 Basics of C **9**
3.1 Data types (primitive types) . . . 9
3.2 Using `printf` to print different data types . . . 9
3.3 Control flow . . . 10
3.4 Functions . . . 11
3.5 Variable scope . . . 12
3.6 Header files . . . 14

4 Pointers **15**
4.1 Pointer Basics . . . 15
4.2 Pointers and functions . . . 16
4.3 Pointers and static arrays . . . 18
4.4 Casting pointers . . . 19
4.5 Pointers, functions and arrays . . . 20

5 Strings in C **22**
5.1 Length of a string . . . 23
5.2 Copying a string . . . 23
5.3 Concatenating/Appending strings . . . 24
5.4 `string.h` . . . 24

Joanna Klukowska
joannakl@cs.nyu.edu

6 Multidimensional Arrays **24**

7 Structures **26**

7.1 `struct` . . . 26
7.2 Pointers to structures . . . 27
7.3 Structures and functions . . . 27
7.4 Creating simple data structures . . . 28
7.5 `typedef` - not really a structure . . . 29

8 Memory Management **29**

8.1 `malloc` . . . 30
8.2 `free` . . . 30
8.3 Revised linked list . . . 30

9 Reading Input Data **32**

10 Summary **32**

Joanna Klukowska
joannakl@cs.nyu.edu

1 Intro to C and Unix/Linux

Brian W. Kernighan

Dennis M. Ritchie

In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of The C Programming Language by Kernighan & Ritchie caused a revolution in the computing world.

1.1 Why C?

Mainly because it produces code that runs nearly as fast as code written in assembly language, but is a high level programming language. Some examples of the use of C:

- Operating Systems
- Language Compilers
- Assemblers
- Text Editors
- Print Spoolers
- Network Drivers
- Language Interpreters
- Utilities

1.2 C vs. Java

C

- procedure oriented
- compiled to machine code: runs directly on a machine hardware

Java

- object oriented
- compiled to byte code: runs within another piece of software (JVM = Java Virtual Machine)

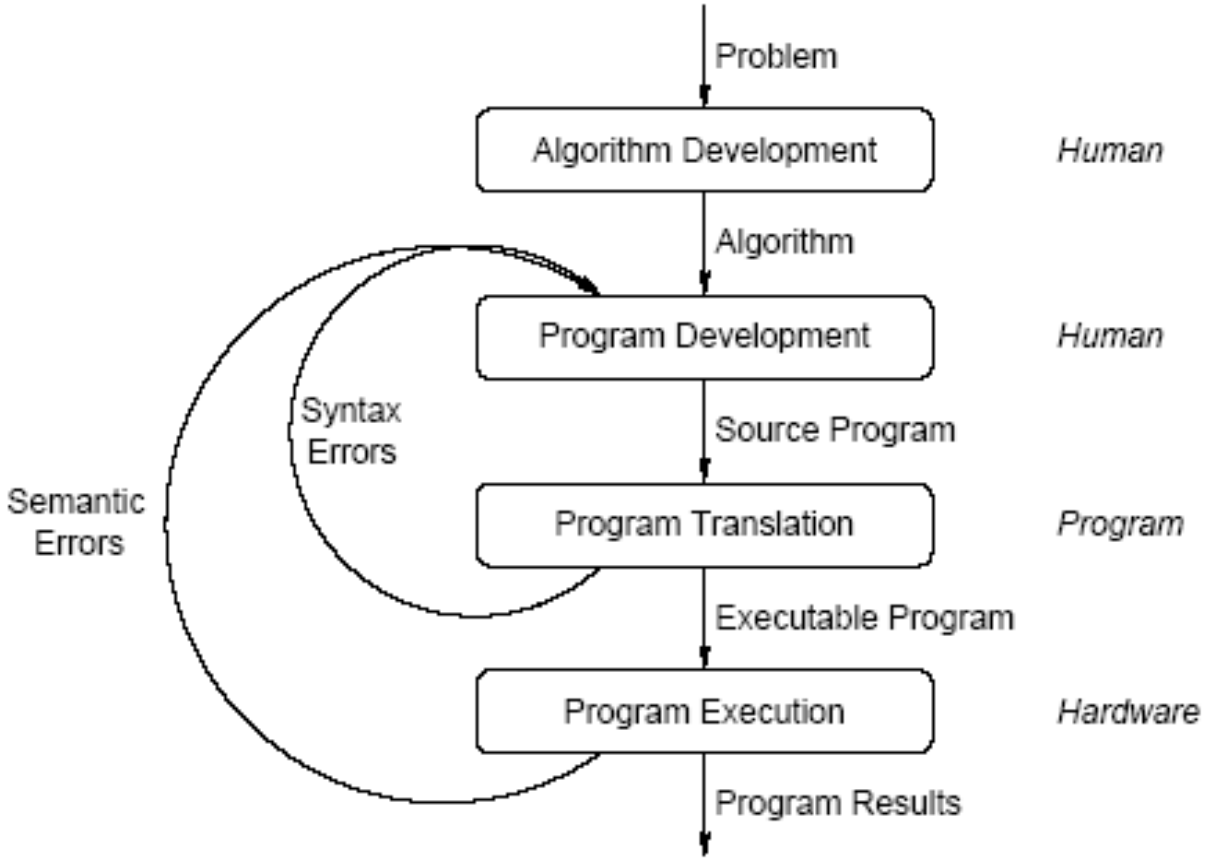
About C

- Hardware independent

Joanna Klukowska
joannakl@cs.nyu.edu

- Programs portable to most computers (in source code format, not executable format)
- Case-sensitive
- Four stages of software development:
  - Editing: Writing the source code by using some IDE or editor
  - Preprocessing or libraries: Already available routines
  - Compiling: translates or converts source to object code for a specific platform source code -¿ object code
  - Linking: resolves external references and produces the executable module

1.3 Software Development Process (Not Only in C)



1.4 Basic Unix Commands

In this class you will be working a lot in a command line or terminal environment.

Joanna Klukowska
joannakl@cs.nyu.edu

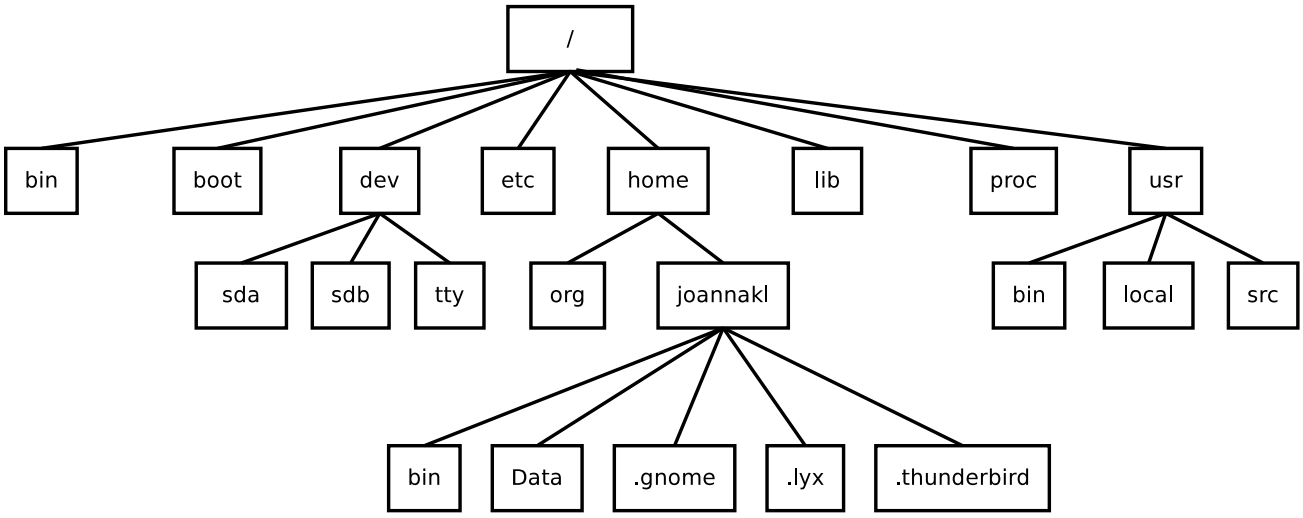
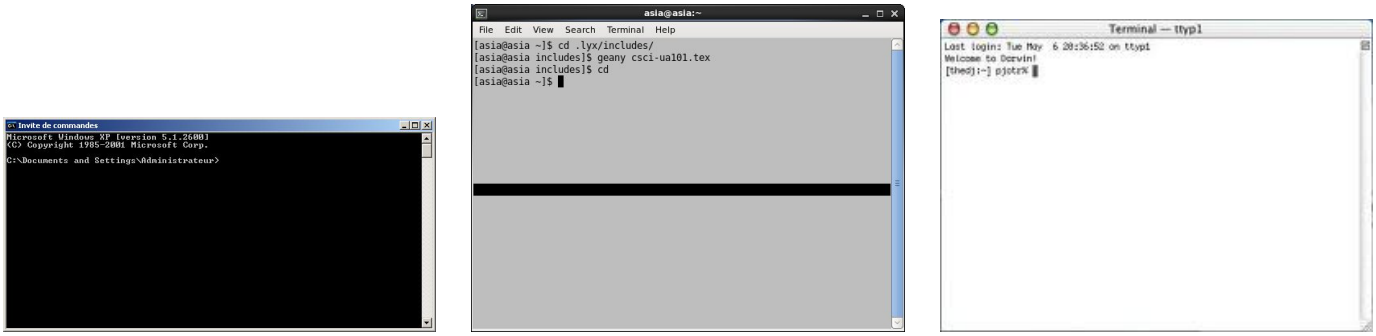


Figure 1: Partial directory structure on a typical Linux/Unix system.



When you log in to a computer system remotely, very often this is the only way of interacting with the computer.

Finding your way around Unix/Linux

Whenever you are logged into a Unix/Linux system, you have a unique, special directory called your **current** or **present working directory** (PWD for short). The present working directory is where you "are" in the file system at the moment, i.e., the directory that you feel like you are currently "in". This is more intuitive when you are working with the command line, but it carries over to the graphical user interface (GUI) as well.

Many commands operate on the PWD if you do not give them an argument. We say that the PWD is their default argument. (Defaults are fall-backs – what happens when you don't do something.) For example, when you enter the "`ls`" command (list files) without an argument, it displays the contents of your PWD. The dot "`.`" is the name of the PWD:

```
ls .
```

and

```
ls
```

both display the files in the PWD, first one because the name of the directory is provided, second one because it is the default behavior for `ls`.

When you first login, your present working directory is set to your **home directory**. Your home directory is a directory created for you by the system administrator. It is the top level of the part of the file system that belongs to you. You can create files and directories within your home directory, but usually nowhere else. Usually your home directory's name is the same as your username.

In Unix/Linux, files are referred to by **pathnames**. A pathname is like a generalization of the file's name. A pathname specifies the location of the file in the file system by its position in the hierarchy. In Unix/Linux, a forward slash "/" separates successive directory

Joanna Klukowska
joannakl@cs.nyu.edu

and file names in the pathname, as in:
`joannakl/Data/NYU_Teaching/website/csci201/daily.php`

There are two kinds of pathnames: absolute pathnames, and relative pathnames. The absolute pathname is the unique name for the file that starts at the root of the file system. It always starts with a forward slash:
`/home/joannakl/Data/NYU_Teaching/website/csci201/daily.php`

A relative pathname is a pathname to a file relative to the PWD. It never starts with a slash: if the PWD is `/home/joannakl/Data/NYU_Teach` then the relative path to the file `daily.php` is `website/csci201/daily.php`.

Dot-dot or **..** is a shorthand name for the parent directory. The parent directory is the one in which the directory is contained. It is the one that is one level up in the file hierarchy. If my PWD is currently `/home/joannakl/Data/NYU_Teaching`, then `..` is the directory `/home/joannakl/Data` and `../../` is the directory `/home/joannakl`.

Unix/Linux commands to get you started

`man command` display a manual page (or simply help) for the `command` (this is the easiest way to learn about options to the commands that you know and about new commands)

`pwd` print the name of the present working directory

`ls` list content of the current working directory

`ls dir_name` list content of the directory named `dir_name`

`cd dir_name` `cd` stands for change directory, changes the current working directory to `dir_name`

`cd ..` move one directory up in the directory tree

`cd` change the current working directory to your home directory

`cp file1 file2` copy `file1` into `file2`, where `file1` and `file2` can be either relative or complete path names

`mv file1 file2` move `file1` into `file2`, where `file1` and `file2` can be either relative or complete path names

`rm file` remove a file (there is no undoing it, so be very careful!)

`mkdir path` make a directory at the specified `path`

`rmdir path` remove the directory specified by the `path` (there is no undoing it, so be very careful!)

`file file_name` determine the type of a file

`less file_name` view the file in the terminal

`more file_name` view the file in the terminal

`cat file_name(s)` concatenate files and print them to standard output

2 First C Program and Stages of Compilation

2.1 Writing and running hello world in C

The usual first program in any language is the "Hello World" program. Here it is in C:

```
1  #include <stdio.h>
2
3  int main()
4  {
5    printf("hello world\n");
6  }
```

Joanna Klukowska
joannakl@cs.nyu.edu

How do we actually write, compile and run the program?

1. Write the code in your favorite text editor (make sure it is a **text** editor, not rich text or any other document format that allows text formatting). For example: Emacs, gedit, geany, vi, sublime, ... Save and name your file. The C source code files usually have .c extension. Their name can be anything (should be something that is related to what the program does).
2. Run the "compiler" to convert the program to executable code (or binary):

   ```
   gcc -Wall -g -o programName programSource.c
   ```

   gcc is the name of the compiler.

   -Wall is an option of `gcc` that tells it to generate and print all warnings (it's always good to know what they are).

   -g option tells `gcc` to generate debugging information (we will need that fairly soon).

   -o programName option tells `gcc` to save the executable code in the file named `programName`. If you do not provide this option, `gcc` will save your executable code in the file named `a.out` - this file gets overwritten without warning when the same program or a different program is compiled in the same directory.

   programSource.c is the name of the text file that contains the source code.
3. If any errors occur during step 2, go back to step 1, fix the errors and recompile. If any warnings occur during step 2, read through them and decide if they need fixing.
4. Run the program: this is done using the following syntax

   ```
   ./programName
   ```

   (Given your environment setup, you may be able to run the program by typing only the name of the program without `./`)
5. If errors occur during running of your program, go back to step 1 and fix them.

2.2 Hello world line by line

Let's look at the hello world program line by line. Here it is again (in a slightly different version):

```
1  #include <stdio.h>
2  /* A simple C program */
3  int main( int argc, char ** argv )
4  {
5      printf("hello world\n");
6      return 0;
7  }
```

Line 1 `#include` inserts another file.
The `.h` files are called **header** files. They contain stuff needed to interface to libraries and code in other `.c` files. They contain lists of functions declarations (not definitions) - like a table of contents for an actual source code file that does not give away the details of the implementation.

Line 2 this is a comment

Line 3 The `main()` function is always where your program starts running. The parameters are optional. They are a way for the program to read the options/arguments provided on the command line.

Lines 4, 7 The body of any function (`main()` is a function) is enclosed in curly braces.

Line 5 `printf()` function prints the formatted text to the screen.

Line 6 This returns 0 from this function. This like could be skipped, but proper C requires that `main()` returns an integer indicating termination status (0 for success, any other value for failure).

Joanna Klukowska
joannakl@cs.nyu.edu

2.3 What really happens when we compile a program?

The compilation occurs in three steps: preprocessing, actual compilation (convert to assembly and then binary) and then linking.

During **preprocessing**, source code is "expanded" into a larger form that is simpler for the compiler to understand. Any line that starts with '#' is a line that is interpreted by the preprocessor. Such lines are called **preprocessor directives**.

- Include files are "pasted in" (`#include`).
- Macros are "expanded" (`#define`)
- Comments are stripped out (`/* */` , `//`)
- Continued lines are joined (`\`)

We can run only this stage by providing ing the `-E` option to `gcc`. Assuming that our program above is saved as `hello.c`, running

```
gcc -E hello.c > hello.i
```

produces another text file called `hello.i`. This file contains over 800 lines. That's because the entire file `stdio.h` is "pasted" at the beginning. If we scroll all the way down, we find our own program at the bottom. It does not include the comments though.

During the **compilation**, the compiler reads the code and makes sure that it is syntactically correct. The compilation itself can be viewed as two steps: 1) create the assembly code and then 2) create the executable code. We can tell `gcc` to stop right after the first stage and look at the assembly code by using the following options:

```
gcc -S hello.c
```

By default, the output is written to a file with `.s` extension. `hello.s` file is fairly short. It contains the assembly instructions. It may look like this:

```
  .file  "hello.c"
  .section  .rodata
.LC0:
  .string  "hello world"
  .text
.globl main
  .type  main, @function
main:
.LFB0:
  .cfi_startproc
  pushq  %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq  %rsp, %rbp
  .cfi_def_cfa_register 6
  subq  $16, %rsp
  movl  %edi, -4(%rbp)
  movq  %rsi, -16(%rbp)
  movl  $.LC0, %edi
  call  puts
  movl  $0, %eax
  leave
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
.LFE0:
  .size  main, .-main
  .ident  "GCC: (GNU) 4.4.7 20120313 (Red Hat 4.4.7-3)"
  .section  .note.GNU-stack,"",@progbits
```

(NOTE: In just a few weeks you will be able to read and understand the above code.)

Joanna Klukowska
joannakl@cs.nyu.edu

The second part of compilation converts the above assembly code into the binary code. To tell `gcc` to stop after that stage, we can use `-c` option. Running

```
gcc -c hello.c
```

produces `hello.o` file (this is called an object file). It is no longer a text file, so trying to look at it will produce gibberish (and many editors will refuse to display it). It contains the binary version of our `hello.c` program.

The final step is **linking**. At this point the binary code for our `hello.c` program exists, but it needs to be linked together with the code from `C` libraries (this is where `printf` lives, for example). Our original `gcc` command runs all of the steps above (including the linking) and produces a runnable/executable binary file.

3 Basics of C

If you know any other programming language you should be able to read simple C programs and understand what they do. In this section we will quickly review the basics and look at a few code examples.

3.1 Data types (primitive types)

The following table lists the data types in C together with their minimum size (in bytes) on a 32-bit and 64-bi systems.

type	size (32bit)	size (64bit)	example
`char`	1	1	`char c = 'a';`
`short`	2	2	`short s = 175;`
`int`	4	4	`int i = 2147483647;`
long	4	8	long int l = 2147483647;
long long	8	8	long long int ll = 9223372036854775807;
float	4	4	float f = 1.0;
double	8	8	double d = 1.0;
pointer	4	8	int *x = NULL;

Notice that there is no type to represent **Boolean data**. Instead, C uses int or char to represent true/false. The value of zero is always interpreted as false, any other value is interpreted as true. For example:

```
1  int i = 0;
2  if (i)    /* same as i!=0 */
3    printf("true");
4  else
5    printf("false");
```

3.2 Using `printf` to print different data types

The `printf` function provides way of printing values of variables of different data types. We need to know the format specifier that `printf` expects for each given type. The very basic `printf` calls using all of the types mentioned in the previous section are listed below:

Joanna Klukowska
joannakl@cs.nyu.edu

```
    char c = 'a';
    short s = 32767;
    int i = 2147483647;
    long int l = 2147483647;
    long long int ll = 9223372036854775806;
    float f = 1.0;
    double d = 1.0;
    int *iptr = NULL;
    char* string = "test string";

    printf("c: %c\n", c);
    printf("s: %i\n", s);
    printf("i: %i\n", i);
    printf("l: %li\n", l);
    printf("ll: %lli\n", ll);
    printf("f: %f\n", f);
    printf("d: %f\n", d);
    printf("iptr: %p\n", iptr);
    printf("string: %s\n", string);
```

See **types.c**.

3.3 Control flow

Selection statements

The syntax of the `if`, `if ... else ...` and `switch` statements should be familiar from other programming languages. In C the expression in the `switch` statement has to have an integral value (`int`, `char`, or anything else that evaluates to an integer).

```
   if (...) {
       ...
   }
------------------------
   if (...)
       ...
   else if
       ...
   else
       ...
------------------------
   switch (c) {
       case 'a' :
           x = 'A';
           break;
       case 'b' :
           x = 'B';
           break;
       ...
       default:
           break;
  }
```

Joanna Klukowska
joannakl@cs.nyu.edu

Repetition/loops

C also has three different loops: `for` loop, `while` loop, and `do ... while` loop. One thing to remember about the for loop is that the control variable has to be declared before the loop.

```
  int i;
  for (i = 0; i < 10; i++) {
      print("i=%d\n",i);
  }
------------------------

  int i = 10;
  while (i > 0) {
      print("i=%d\n",i);
      i--;
  }
------------------------

  int i = 10
  do {
      print("i=%d\n",i);
      i--;
  } while (i > 0)
```

Jump statements

C provides break and continue statements. Both of them can be used with loops. break can also be used in a switch statement.

`break` immediately jump to the next statement after the loop or after the switch statement

`continue` immediately jump to the next iteration of the loop

C also has another way of "jumping" in the code. It is the goto statment. One can specify labels in the code as in the example below and the goto statement immediately moves to the line following the label.

```
  for (...) {
      for (...) {
          ...
          if (disater)
              goto LABEL_1;
      }
  }
LABEL_1:
  ...
```

Do not use `goto` statements in your code. They are considered to be bad programming style at this point and result in code that is hard to debug and trace.

3.4 Functions

The file that contains `main` function can also contain other functions.

In C functions cannot be defined within other functions. A function has to be declared before it is used.

```
1 #include <stdio.h>
2 int modulo(int x, int divisor);    // Function declaration
3
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
4 int main()
5 {
6     int x = 10;
7     int divisor = 2;
8     printf("%d mod %d = %d\n", x, divisor,
9               modulo(x,divisor, 0 ));
10    divisor = 3;
11    printf("%d mod %d = %d\n", x, divisor,
12              modulo(x,divisor, 1));
13    return 0;
14 }
15
16 /*computes the remainder from dividing x by divisor */
17 int modulo(int x, int divisor, int rec)
18 {
19   if ( !rec ) {
20     /* iterative version */
21     while (x >= divisor) {
22         x -= divisor;
23     }
24     return x;
25   }
26   else {
27     /* recursive version */
28     if (x < divisor)
29         return x;
30     else
31         return modulo(x - divisor, divisor, rec);
32   }
33 }
```

We can also create functions in multiple files. If **functionA** uses **functionB** which is defined in another file, then one of the following has to be true:

- **functionB** is declared in a file that defines **functionA** (before **functionB** is used)
- the file that contains **functionA** includes a header file that declares **functionB** (we will look at header files soon)

3.5 Variable scope

Variable Scope: global/external variables

Scope rules:

- The scope of a variable/function name is the part of the program within which the name can be used
- A global (external) variable or function's scope lasts from where it is declared to end of the file
- A local (automatic) variable's scope is within the function or block

```
int x;

void foo(int y) {
  y++;
  x++;    /* x is accessible because it is in global scope*/
}
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
void bar() {
  y = 1; /* wrong - there is no y in this scope*/
  x = 1; /* x is accessible because it is in global scope*/
}
```

Demo (`scope.c`): What if we change all `y`'s to `x`'s?

```
void foo(int y) {
  if (y > 10) {
    int i;
    for (i = 0; i < y; i++) {
    ...
    }
  }
  else {
    y++;
    // 'i' is out of scope here
  }
}
```

Global/Local Variable Initialization In the absence of explicit initialization

- global and static variables are guaranteed to be initialized to zero
- local variables have undefined initial value **!!!**

Definition vs. declaration of a global (external) variable

- Declaration specifies the type of a variable.
- Definition also set aside storage for the variable and initializes its value.

The following example uses an external/global variable and a function both defined in a file different than `main` function.

```
add.c
--------------------
int counter = 0;

void add_one() {
  counter++;
}
```

```
program.c
--------------------
#include <stdio.h>

extern int counter;
void add_one();

int main() {
  printf("counter is %d\n", counter);
  add_one();
  add_one();
  printf("counter is %d\n", counter);
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
    return 0;
}
```

The two file program can be compiled using

```
gcc add.c program.c  -o program
```

or using separate compilation (each file compiled individually)

```
gcc -c -Wall -g add.c

gcc -c -Wall -g program.c

gcc add.o program.o -o program
```

WARNING: Avoid global/external variables (constants are fine)! They are considered bad programming style. Use of globals results in programs that are hard to debug and trace: just imagine how many different functions can possibly be modifying the value of a global variable.

`static` keyword The keyword `static` limits a scope of a global variable to within the rest of the source file in which it is declared. If we add the keyword `static` to the declaration of `counter` in `add.c` above, the `counter` variable will no longer be accessible from `program.c`.

The other use of keyword static allows the variable (global or local) to preserve its value between function calls. If the variable is declared with keyword static the initialization is performed only once.

```
add_2.c
--------------------
#include <stdio.h>

void add_one() {
    static int counter = 0;
    counter++;
    printf("counter is %d\n", counter);
}
```

```
program_2.c
--------------------
void add_one();

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

3.6 Header files

Header files contain declarations of functions and globals that can be included in other source code files. The header file's name usually matches the name of the file that provides the source code (the definitions) and ends with `.h` extension.

We could rewrite the example from the previous section by adding a header file as follows:

```
add.c
--------------------
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
int counter = 0;

void add_one() {
        counter++;
}
```

```
add.h
--------------------
extern int counter;
void add_one();
```

```
program.c
--------------------
#include <stdio.h>
#include "add.h"

int main() {
    printf("counter is %d\n", counter);
    add_one();
    add_one();
    printf("counter is %d\n", counter);
    return 0;
}
```

Notice that the file `program.c` no longer needs to repeat all the declarations, it simply includes the file `add.h`. The name of that file is included in **quotes** - this tells gcc to look for that file in a local directory, rather than in the locations of standard header files on your machine.

4 Pointers

The program's memory can be viewed as an array of bytes:

- an array has indexes, memory has addresses
- each variable takes up 1, 2, 4, or 8 bytes according to its size

We can use pointers to store memory addresses and access this memory array.

4.1 Pointer Basics

A **pointer** is a variable that stores a memory address and can be used to access that memory address (and more). Pointers in C have the type information associated with them: a pointer storying an address of an integer is different than a pointer that stores an address of a float.

To **declare a pointer** variable we add a star to the name of the variable:

```
type1 *varName1;

type2 * varName2;

type3 *varName3, * varName4;
```

Note that the spacing around the star does not matter, but there has to be one star per name, if more than one pointer variable is declared in the same statement.

Joanna Klukowska
joannakl@cs.nyu.edu

The **value of of a pointer variable** has to be a memory address of another variable (could be a pointer) or `null` to indicate that the pointer does not point to anything. To obtain a memory address of a variable we use the & operator: given a variable `x`, `&x` is the memory address of `x`.

Example:

```
int x = 5;     //regular variable
int y = 17;    //regular variable

int *p, *q;    //two pointers that can store memory addresses
               //of int variables

p = &x;       //p now stores the address of x
q = null;      //q does not store any address
```

We can use a pointer variable to access the value stored at the address that the pointer itself stores (I know, this sounds complicated). To do so we need to use a **dereference/indirection operator**, which is the * again.

Example continued:

```
*p = 7;        //changes the value stored in x to 7;

printf("x  = %d", x);
printf("*p = %d", *p); //both printf statements print the value
                       //of x variable

*q = 3;        //ERROR!, cannot dereference a null pointer!
```

Notice the **two new operators** that we just introduced:

& address of operator gives the address of a variable that follows it

***** has two uses: 1) in pointer declarations, 2) as a dereference operator to access the value of the variable that the pointer points to

4.2 Pointers and functions

In C, function arguments are passed by value (like in Java) - what does that entail?

Problem: How would you write a simple **`swap()`** function that exchanges the values of its two arguments (in Java or in C)?

Attempt #1: This code seems like it should work, but it does not. Why?

```
1
2  #include <stdio.h>
3
4  void swap(int x, int y);
5
6  int main()
7  {
8    int x = 1;
9    int y = 2;
10   printf("x=%d, y=%d\n", x, y);
11   swap(x,y);
12   printf("x=%d, y=%d\n", x, y);
13   return 0;
14 }
15
16 //swaps values of x and y
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
17  void swap(int x, int y)
18  {
19    int tmp;
20    tmp = x;
21    x = y;
22    y = tmp;
23  }
24
```

Attempt #2: Use pointers as parameters to the function. This way main() and swap() deal with the same memory locations, even though they have their own copies of the pointers.

```
1
2 #include <stdio.h>
3
4 void swap(int *x, int *y);
5
6 int main()
7 {
8   int x = 1;
9   int y = 2;
10  printf("x=%d, y=%d\n", x, y);
11  swap(&x, &y);
12  printf("x=%d, y=%d\n", x, y);
13  return 0;
14 }
15
16 //swaps values of x and y
17 void swap(int *x, int *y)
18 {
19  int tmp;
20  tmp = *x;
21  *x = *y;
22  *y = tmp;
23 }
24
```

This version of function `swap()` takes as parameters memory addresses of `x` and `y` that are defined in `main()`. This way it can manipulate the `x` and `y` variables even from within the function: there is only one copy of `x` and one copy of `y` in the program. On line 10 of the program, when the function is called, we pass it the `&x` (address of `x`) and `&y` (address of `y`).

Something to Think About:
DNHI: Would this alternative version of the `swap` function work? Why? or Why not? Draw what happens in memory when this `swap` function is used.

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

Note: you can run the code to see its output, but you should also be able to figure this out without running the code, just by analyzing what happens in memory.

Joanna Klukowska
joannakl@cs.nyu.edu

4.3 Pointers and static arrays

A static array in C has a fixed size that cannot be changed after it is declared. The memory is allocated automatically and we do not need to do anything. Writing

```
const int size = 10;
int a[size];
```

creates an array a of 10 integer elements. (We will talk about an alternative way of creating arrays later in the course.)

To populate the array with values, we can use a loop that assigns a value to each location:

```
for (i = 0; i < size; i++) {
  a[i] = i;
}
```

We can then print array using its name and subscript notation as follows

```
for (i = 0; i < size; i++) {
  printf("%d ", a[i]);
}
```

An alternative approach to accessing array locations is to set a pointer to point to the first element in the array

```
int * pa = &a[0];
```

and then use that pointer to print array using pointer to the array and subscript notation (**!!! you can use a pointer with the subscript operator**)

```
for (i = 0; i < size; i++) {
  printf("%d ", pa[i]);
}
```

The array name itself is really a constant pointer to the first element in the array (constant pointer means that its value, i.e. the memory address it stores, cannot be changed). So we can set a pointer `pa1` to be equal to `a` instead.

```
int * pa1 = a;
for (i = 0; i < size; i++) {
  printf("%d ", pa1[i]);
}
```

Yet another way of traversing the array is to advance a pointer through the array by incrementing what it points to (**!!! you can add/subtract values to/from pointers**)

```
int * pa2 = a;
for (i = 0; i < size; i++) {
  printf("%d ", *pa2);
  pa2 = pa2+1; //this adds enough bytes to pa2 to advance it to the next array location
}
```

And if we wish to traverse an array backwards, we set the pointer to the last location in the array and keep on moving it towards the front

```
int * pa3 = a + size-1;
for (i = 0; i < size; i++) {
  printf("%d ", *pa3);
  pa3 = pa3-1; //this subtracts enough bytes from pa3 to move it to the
               //previous array location
}
```

Joanna Klukowska
joannakl@cs.nyu.edu

See `array.c` for all of the examples mentioned above.

Something to Think About:
DNHI:
1) Write a loop that traverses the array backwards, but uses a pointer that is initialized to the first element in the array.
2) Write a loop that traverses the array and uses a pointer that is initialized to the first element in the array, but does not modify where that pointer points to.

Summary of pointer arithmetic and arrays

Assume that we have an array of integers and a pointer to an array as follows:

```
int a[10]; //initialized with values somewhere
int *pa;
```

- Array name is a constant pointer to the first element in the array. The following two statements are equivalent:

  ```
  pa = & a[0];
  pa = a;
  ```

- An array name is a **constant** pointer so it cannot be modified. The following statements are all illegal:

  ```
  a++;
  a = a+5;
  a = pa;
  ```

- We can perform arithmetic on pointers. Adding a value K to a pointer advaces the memory location that is stored in the pointer by K times the number of bytes used for storing variables of the pointer's type. For example, if type `int` uses 4 bytes, then

  ```
  pa=pa+3;
  ```

  advances `pa` by 12 bytes.

  Subtracting a value from a pointer works in the same way.

- The following expressions are all equivalent (assuming `pa` was assigned value of `a` and `i` is an integer between 0 and a declared size of array `a`) and point to (contain the address of) the `i`'th element of array `a`:

  ```
  & a[i]
  a + i
  & pa[i]
  pa + i
  ```

- The following expressions are all equivalent (assuming `pa` was assigned value of `a` and `i` is an integer between 0 and a declared size of array `a`) and are values of the `i`'th element of array `a`:

  ```
  a[i]
  *(a+i)
  pa[i]
  *(pa+i)
  ```

4.4 Casting pointers

Pointer (as other types) can be cast to another type.

Try to figure out what the following program does, then run it and see if your prediction was correct. Make sure you understand why the print statement produces what it does.

Joanna Klukowska
joannakl@cs.nyu.edu

```
1  casting.c:
2  -------------------------
3  #include <stdio.h>
4
5  int main() {
6    int x[] = {65, 66, 67, 68, 69};
7    char *p = (char * ) x;
8
9    int i;
10   for (i = 0; i < 5; i ++ ) {
11     printf ( "%c  ", *(p+i) );
12   }
13   printf("\n");
14
15   return 0;
16 }
```

Replace the printf() on line 11 with

```
printf ( "%d  ", *( (int*)(p) + i));
```

Can you figure out what the code will print right now?

4.5 Pointers, functions and arrays

When passing an array to a function, we are really passing the pointer to the first element of the array. The following two function declarations are equivalent (well, almost):

```
int sumElements( int a[], int size );

int sumElements( int *a, int size );
```

The function can be used to compute the sum of all of the elements of the array, or any subsequence of the elements within the array.

```
1  #include <stdio.h>
2
3  int sumElements (int *a, int size );
4
5  int main () {
6    const int size = 10;
7    int a[size];
8    int i;
9    for (i = 0; i < size; i++) {
10     a[i] = i;
11   }
12   printf( "sum of all elements in a is %d\n",
13                   sumElements ( a, size ) );
14   printf( "sum of first 5 elements in a is %d\n",
15                   sumElements ( a, 5 ) );
16   printf( "sum of last 5 elements in a is %d\n",
17                   sumElements ( a+5, 5 ) );
18   return 0;
19 }
20
21 //adds size many elements of array a
22 int sumElements ( int *a, int size ){
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
23    int sum = 0;
24    int i;
25    for (i = 0; i < size; i++ ) {
26      sum = sum + a[i];
27    }
28    return sum;
29  }
```

See `sum.c` and `sum_2.c` for this code and its version that passes int a[] as a parameter to the function.

Something to Think About:
DNHI: Modify the code above to check your answers to the following questions:
1) What happens if `sumElements()` function modifies the content of array `a`? Do changes affect the array declared in `main()`.
2) What happens if the `size` paramter passed to the `sumElements()` function exceeds the size of the declared array `a`? Try it by exceeding the size by small numbers (1, 2, 5, 10) and large numbers (1000).
Make sure you understand what you observed.

Generic swap

We looked at a `swap()` function that exchanged the values of two integer variables.

How about swapping values of variables of any type - one function that can swap values of type int, long, char, double, float, etc. Some of you may think of generics, but C does not have generics (or even templates). But we still can write a generic `swap()` function.

The following function uses casts the pointers passed to it to `(void *)`. `void *` is a pointer to anything. It then swaps item1 and item2 one byte at a time completely ignoring what those bytes represent.

```
swap_generic.c (partial):
-------------------------
/**
* swaps values of item1 and item2
* item1 - pointer to a variable / data structure
* item2 - pointer to a variable / another data structure
*         of the same type as item1
* size - number of bytes used for storying item1/item2
*
* preconditions: the item1 and item2 use the same number
* of bytes, otherwise the results are unspecified
*/
void swap(void *item1, void *item2, int size) {

  // cast to char* which is 1 byte in size
  char *x = (char *)item1;
  char *y = (char *)item2;
  char tmp;  // 1 byte of storage

  int i;
  for (i = 0; i < size; i++) {
    // for every byte: swap the byte
    tmp = x[i];
    x[i] = y[i];
    y[i] = tmp;
  }
}
```

See `swap_generic.c` for an example of use of this function.

Joanna Klukowska
joannakl@cs.nyu.edu

5 Strings in C

Two strings walk into a bar and sit down. The bartender says, "So what'll it be?"
The first string says, "I think I'll have a beer quag fulk boorg jdkˆCjfdLk jk3s d#f67howe˜owmc63ˆDz x.xvcu"
"Please excuse my friend," the second string says, "He isn't null-terminated."

A string in C is an array of characters terminated by a null character (\0 **).** Every function that performs operations on strings uses that null character. A missing null character in a string is also the source of most errors (and jokes) related to strings in C.

Whenever we have a double quoted string in a program it is stored as a string constant terminated by a null character. For example:

```
printf("Hello World!\n");
```

Writing

```
char h[] = "hello";
```

creates and array of 6 characters (yes, 5 letters + a null character). It is equivalent to writing

```
char h[6]= "hello";
```

Note that the following lines will compile as well and sometimes even run without obvious problems (at first)

```
char h[5] = "hello";

char h[2] = "hello"; //this will generate a warning, but not an error
```

In both cases we are writing past the end of the array!

Strings in C are governed by all the same rules as arrays discussed in the previous section.

On the other hand, writing

```
char *h = "hello";
```

does not create a new array that `h` points to. The string constant "hello" is stored in the same area in memory where globals are stored and h is just a pointer to that string constant. Attempting to modify such string constant produces undefined results. (Undefined result implies that the result is unpredictable and depends on the compiler and system used.) For example,

```
char *s = "hello";

s[0] = 'H';
```

does not produce a compiler error, but results in a segmentation fault error during runtime (at least on our system).

The following code has two different strings: `h1` is an array of characters, `h2` is a constant string. It also has a global, `y`, and local, `x`, variables of type `int`. The memory locations of all four of them are printed out.

```
1  #include <stdio.h>
2
3  const int y = 5;
4
5  int main () {
6    int x = 17;
7    char h1[] = "hello";
8    char *h2 = "hello";
9
10   printf("h1 is at %p\n", h1);
11   printf("h2 is at %p\n", h2);
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
12    printf("\n");
13    printf(" x is at %p\n", &x);
14    printf(" y is at %p\n", &y);
15
16    return 0;
17  }
```

One possible output is:

```
h1 is at 0x7fffeb307260

h2 is at 0x4006ac

x is at 0x7fffeb30725c

y is at 0x4006a8
```

This clearly shows that `h2` and `y` are in a different area of memory than `h1` and `x`.

5.1 Length of a string

As mentioned before, most (if not all) functions written for strings make use of the null character. They assume that there is null character that terminates the string - this is the only way of knowing where the string ends.

The following function can be used to compute the length of the string:

```
int length(char *str)
{
  int length = 0;
  while ((*(str + length)) != '\0') {
    length++;
  }
  return length;

  // array version: change *(str + length) to str[length]
}
```

This function looks at every character in string `str` until it encounters null character.

5.2 Copying a string

How about making a duplicate of a string? Would something like this work?

```
char *h = "hello";

char *w = h;
```

Well, we can access the string "hello" using both *h and *w. But there is only one string in memory. In order to create a true copy/duplicate, we need to do some more work.

```
void copy(char *src, char *dst) {

  while ((*src) != '\0') {
    *dst = *src;
    dst++;
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
    src++;
  }
  *dst = '\0';
}
```

This code assumes that the string passed as destination, `dst`, has enough memory associated with it to store all the characters stored in `src`. To use this copy function we have to create an array in memory for the second string:

```
char *h = "hello";

char w[100];

copy(h, w);
```

Note that this would not work!!!!

```
char *h = "hello";

char *w;

copy(h, w);
```

5.3 Concatenating/Appending strings

Something to Think About:
DNHI: Write the following functions that operate on null terminated strings (your functions do not need to perform correctly if they are given strings that are not null terminated).
1) a function that appends `src` string to the (possibly empty) `dst` string

```
void append (char * src, char * dst ) ;
```

2) a function that appends at most `n` characters or until null is encountered from the `src` string to `dst` string

```
void appendN (char * src, char * dst, int n ) ;
```

5.4 `string.h`

The `string.h` header file contains declarations of many useful functions for working with null terminated string (all of the one's mentioned above, among others). You can learn about their names by using the man pages:

```
man string.h
```

and then read about specific functions by using man pages for those specific functions, for example:

```
man strlen
```

6 Multidimensional Arrays

In C multidimensional arrays are stored in consecutive memory locations.

We can create a 2D array using the following syntax:

Joanna Klukowska
joannakl@cs.nyu.edu

```
#define ROWS 3

#define COLS 2

int matrix[ROWS][COLS] = { { 1,2}, {3,4}, {5,6} };
```

We can now traverse this matrix using a traditional for loop. The following code adds all the entries in the above 2D matrix:

```
for (i = 0; i < ROWS; i ++) {
  for (j = 0; j < COLS; j++) {
    sum += matrix[i][j];
  }
}
```

But we can also use a pointer to int and treat this matrix as a one dimensional array that is traversed using a single for loop

or a doubly nested for loop

```
for (i = 0; i < ROWS; i ++) {
  for (j = 0; j < COLS; j++) {
    sum += p[j*ROWS+i];
  }
}
```

Something to Think About:
DNHI: Follow the above idea to create a 3D array of integers and use a pointer to the first element to traverse the array using a single loop, a doubly nested loop and a triply nested loop.

Pointers to pointers as matrices

Creating a variable that stores a memory address of another variable that stores a memory address does not seem significant in itself. But in view of the relationship between pointers, arrays and strings the pointers to pointers are important. They are a way of creating a different type of multidimensional arrays.

The following array consists of three pointers to strings and is populated by three strings:

```
char* names[3] = { "alice", "bob", "clark" };
```

Each element of this one dimensional array of pointers contains a separate string. The following definition may seem to be equivalent to the one above, but it is not!

```
char** names = { "alice", "bob", "clark" }; //INCORRECT
```

Use of `[]` operator guarantees that the memory is allocated properly. The above definition does not allocate memory for all the strings.

But we can use a double pointer to char to access the elements of the array of characters:

```
char ** p_names = names;
for (i = 0; i < 3; i++ ) {
  printf ("%s \t %s \n", names[i], p_names[i] );
}
```

An example of another multidimensional array is the `argv` array that can be passed to `main()` function:

```
int main(int argc, char *argv[])
   //alternatiely, char **argv
{
  int i;
  //prints the number of command line arguments
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
  printf("argc = %d\n", argc);

  //prints the values of all the command line arguments
  for (i = 0; i < argc; i++) {
    printf("%s ", argv[i]);
  }
  printf("\n");
}
```

7 Structures

7.1 `struct`

Java and C++ have classes and objects. C has structures - we can think of **structures** as classes without methods, just a collection of data items grouped into a single thing. (Warning: the structures in C++ are not exactly the same as structures in C.)

Example of a structure definition (notice the keyword `struct`):

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

This gives us a definition for a new type called `point` with two members x and y. In order to declare a variable of type point we write:

```
struct point p1;
```

The keyword `struct` is needed in the declaration of the variable. It should not be omitted.

To access individual coordinates of the point `p1` we use the **dot operator:**

```
p1.x = 3.5;
p1.y = 8.9;

printf("p1 = (%f,%f)", p1.x, p1.y);
```

Alternatively, we can write

```
p1 = {3.5, 8.9};
```

This is allowed as long as the structure is simple and there is no ambiguity in which value is supposed to be assigned to which structure member.

Structure definitions can contain any number of members (variables) of any type. This includes pointers. For example:

```
struct student {
  char* id;
  char* name;
  float gpa;
  int num_of_credits;
}
```

And, of course, there can be a structure whose members are other types of structures. For example a rectangle defined by its two diagonally opposite corners:

```
struct rectangle {
  struct point c1;
  struct point c2;
}
```

Joanna Klukowska
joannakl@cs.nyu.edu

7.2 Pointers to structures

We can create a variable of the type that is a structure and that variable has its own memory address, hence we can use pointers to such variables as well.

Assuming the point structure from the previous section

```
struct point p2 = {1.5, -3.1};
struct point *ppoint;

ppoint = & p2;
```

Using the standard pointer dereference operator we can access the values of the members of p2 using ppoint

```
(*ppoint).x
(*ppoint).y
```

The parenthesis are needed since the `.` operator has higher precedence than the `*` operator.

But since pointers to structures are very frequently used, there is a shorthand notation for the above (well, you be the judge how shorthand it is):

```
ppoint->x
ppoint->y
```

The arrow operator `->`, is a dash followed by the greater than sign.

For example: assume the rectangle structure defined in the previous section. To access the coordinates of the two points using a pointer to a rectangle, we may write:

```
struct rectangle r = { {0,0}, {3,4} };
struct rectangle *pr = &r
printf(  t w o  corners are (%f,%f) and (%f,%f)    ,
            (pr->c1).x, (pr->c1).y, (pr->c2).x, (pr->c2).y );
```

We first use the pointer `pr` to access a specific point/corner and then use that to access its `x` and `y` coordinates.

When you use multiple operators, make sure that it is clear what happens in what order. C has its precedence rules and has no trouble parsing things like

```
*p->str++
```

but most humans reading this will have a hard time deciding what is happening. (`p` is a pointer to a structure, so `->` accesses its member. `str` is a member of `p` and is a pointer itself and `*` dereferences that pointer. Finally, `++` is applied to the value after dereferencing `str`.) It would have been much clearer if the above was written as

```
(*(p->str))++
```

7.3 Structures and functions

Structures can be passed to functions either by value or by pointer (just like any other variable). The following function attempts to add two points by adding the corresponding coordinates (this is vector addition) and returning sum.

```
struct point addpoints( struct point p1, struct point p2 ) {
  p1.x += p2.x;
  p1.y += p2.y;
  return p1;
}
```

Joanna Klukowska
joannakl@cs.nyu.edu

Something to Think About:
DNHI: Did the coordinates of the point that was passed to it as p1 change? Why or why not? Write a program that demonstrates what happens.

Another version of this function uses pointers

```
struct point addpoints( struct point *p1, struct point *p2 ) {
  struct point p = { p1->x + p2->x, p1->y + p2->y };
  return p;
}
```

and a slightly modified function adds the two points and stores the result in p1 instead of returning it:

```
void addpoints( struct point *p1, struct point *p2 ) {
  p1->x += p2->x;
  p1->y += p2->y;
}
```

7.4 Creating simple data structures

With structures we can start creating things like linked lists, trees, graphs, etc. The C `struct` describes the building block (or node) of such data structure, and separate functions provide their functionalities. Unlike in an object oriented language, the functions and structures are not combined together into a single thing.

Here is a definition of a node for a linked list

```
struct node {
  char *word;
  struct node *next;
}
```

The node can store a word/string and a pointer to the next node in the list. We want to make sure that the `next` pointer is initialized to `null`, unless it is set to a specific node.

A function that adds a node to an existing list needs to take as a parameter a pointer to the node and a pointer to the head of the list.

```
void addFront(struct node * n, struct node ** head) {
  if (head == NULL)
    return;
  n->next = (*head);
  (*head) = n;
}
```

```
1  #include <stdio.h>
2  struct node {
3      char * word;
4      struct node * next;
5  };
6
7  void addFront(struct node * n, struct node ** head);
8
9  int main() {
10     struct node * head = 0; //head pointer for the list
11
12     struct node n1 = { "hello", 0};
13     struct node n2 = { "cso201", 0};
14     struct node n3 = { "students", 0} ;
15
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
16      addFront ( &n1, &head);
17      addFront ( &n2, &head);
18      addFront ( &n3, &head);
19
20      struct node * current = head;
21
22      while (current != 0 ) {
23          printf ("%s     ", current->word );
24          current = current->next;
25      }
26      printf("\n");
27
28      return 0;
29  }
30  //... tbe above definition of addFront
```

Something to Think About:
DNHI: Use the ideas of this section to create a binary tree structure: define a node that can be used for a binary tree, write a function that adds nodes to the tree and one that performs an inorder traversal of the tree. Write a sample program that tests your sturcture and methods.

7.5 `typedef` - not really a structure

Typedefs are a way of creating more convenient or shorter names for existing types . For example,

```
typedef int * int_pointer;
```

creates a new type called `int_pointer` that is equivalent to `int *` (useful for those who cannot or do not like to keep track of the stars).

The `typedef` tends to be often used with the structures.

Rewriting the node structure definition as follows:

```
typedef struct {
  char * word;
  struct node * next;
} node;
```

allows us to use `node` as the type name, rather than `struct node`:

```
node n;
```

rather than

```
struct node n;
```

Something to Think About:
DNHI: Rewrite the `list.c` file using the definition of the node with `typedef` as above.

8 Memory Management

So far we have been using **static** memory. This limits the programs that we can write. For example, we needed to decide up front (at the time of writing the program) how many nodes there are going to be in our linked list. We can write much more interesting programs once we learn how to manage the memory: request it when we need it and release it when we no longer need it

(the second part is a crucial one in a language like C).

Joanna Klukowska
joannakl@cs.nyu.edu

8.1 malloc

The memory allocated **dynamically** (while the program is running) is in the area of memory reffered to as **heap**. In Java/C++ we use the operator `new` to allocate memory dynamically. In C we use a function called **`malloc`** (and its other flavors). The function takes as a parameter an integer that indicates how many bytes of memory we want and returns a pointer to a newly allocated block of memory of that size. Occasionally (for one reason or another) `malloc` is not able to give us what we want. In that case, the call to `malloc` returns 0 or `NULL`.

Here are some examples of using `malloc`:

```
//allocate memory big enough for an int
int * num = (int *) malloc (sizeof(int) );

//allocate memory big enough for an array of 1000 characters
char * words = (char *) malloc (1000*sizeof(char));

//allocate memory big enough for a node structure
node * n = (node *) malloc (sizeof(node) );
```

Before using any of the above allocated memory, we should always check if the memory was actually allocated:

```
if (num == 0){ //or (num == NULL)
  //ERROR: the memory was not allocated
}
```

8.2 free

C does not provide garbage collection. This means that if we allocat a chunk of memory dynamically (i.e., using `malloc`), we need to release it when it is no longer needed. If releasing of memory is not done diligently, the programs have **memory leaks** which decreases their own performance and, eventually, decreases the performance of the entire machine on which the program runs.

Here are some examples of using `free`:

```
free(num);

free(words);

free(n);
```

In all of the above cases, the pointer passed to **`free`** must point to a memory block that was previously allocated with **`malloc`**. If that is not the case, or if that memory block has been previously freed, the behavior of free is undefined. You may get a segmentation fault, but this is not guaranteed.

8.3 Revised linked list

The program for a linked list that we wrote previously can be rewritten to use dynamic memory allocation as below. This time the `addFront()` function takes care of the creation of the node and we added a function for removing the first node.
Notice: another include statement in this program - this is where `malloc` and `free` are declared

```
1 list2.c:
2 ----------------------------
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 struct node {
7     char * word;
8     struct node * next;
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
};

void addFront(char * word, struct node ** head);
char* removeFront(struct node ** head);


int main() {
    struct node * head = 0; //head pointer for the list

    char * w1 = "hello";
    char * w2 = "cso201";
    char * w3 = "students";

    addFront ( w1, &head);
    addFront ( w2, &head);
    addFront ( w3, &head);

    struct node * current = head;

    while (current != 0 ) {
        printf ("%s     ", current->word );
        current = current->next;
    }
    printf("\n");

    return 0;
}

void addFront(char * word, struct node ** head) {
    if (head == NULL)
      return;
    //allocate memory for storying the node
    struct node *n = (struct node *)malloc(sizeof(struct node));
    //make sure that the memory was allocated,
    //if not, just quit
    if (n == NULL)
      return;
    n->word = word;
    n->next = (*head);
    (*head) = n;
}

char* removeFront(struct node ** head) {
    //check if the list is empty
    if (head == NULL || *head == NULL )
        return NULL;
    //get the pointer to the first element
    struct node *n = *head;
    char *w = n->word;
    //advance head
    (*head) = n->next;
    //free memory allocated for n
    free(n);
    return w;
}
```

Joanna Klukowska
joannakl@cs.nyu.edu

Something to Think About:
DNHI: Use the ideas of this section to revise your binary tree structure: define a node that can be used for a binary tree, write a function that adds nodes to the tree, one that performs an inorder traversal of the tree and one that removes nodes from the tree. Write a sample program that tests your sturcture and methods.

9 Reading Input Data

A C program can use the function `getchar()` to read unformatted input. It returns a single character.

```
int c;

while ((c = getchar()) != EOF) {
  printf("%c", c);
}
```

The above code will read the input one character at a time and print it back to standard output. The loop ends when the end of file (EOF) is encountered. One can trigger the end of file character by pressing `Ctrl+D` in the terminal.

When the input is formatted, `scanf()` function is much more useful because it can perform a lot of parsing and converting to appropriate types.

```
double sum,v;

while (scanf("%lf",&v) == 1)
  printf("%.2f\n", sum += v);
```

The above loop keeps reading in floating point numbers. It accumulates their sum and displays it to standard output.

```
int day, year;

char month[100];

scanf("%s %d, %d", month, &day,  &year);
```

The above code reads in a data from the user/input file.

10 Summary

This concludes our discussion about C specifically as a programming language. We will use it for the rest of the semester and you will learn other features of the language. The Unix/Linux manual pages are a great resource for documentation. Running, for example, `man malloc` provides the documentation for the malloc f unction that is the most specific to the system on which the program is developed.