Lecture 02: Linux and C Programming Language

Concurrent and Multicore Programming
CSE 436/536

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan

What is parallel computing?

- A form of computation*:
  - Large problems divided into smaller ones
  - Smaller ones are carried out and solved simultaneously

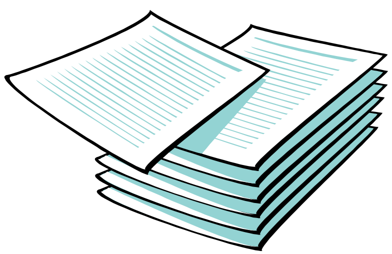
- Parallel programming is about
  - Distributing data and work
  - Coordinate individual tasks (synchronization and communication)
  - Efficiently use resources (memory hierarchy)

*http://en.wikipedia.org/wiki/Parallel_computing

An example: grading

15 questions

300 exams



Summary

- Data: 300 copies of exam
- Task: grade total 300*15 questions
- Data parallelism
  - **Distributed 300 copies to three TAs**
  - **They work independently**
- Task Parallelism
  - **Distributed 300 copies to three TAs**
  - **Each grades 5 questions of 100 copies**
  - **Exchange copies**
  - **Grade 5 questions again**
  - **Exchange copies**
  - **Grade 5 questions**

	Data Parallelism	Task Parallelism
Data	Different	Same
Task	Same	Different

- The three TAs can do in parallel, we can achieve 3 time speedup theoretically

Course Objectives

- Learn fundamentals of concurrent and parallel computing
  - …..
- Develop skills writing and analyzing parallel programs
  - Write parallel program using OpenMP, Cilk/Cilkplus, CUDA, and MPI programming models.
  - Perform analysis of parallel program problem.

We will alternate fundamental and practice classes as much as we can!

Contents

- Remote Login using SSH
- Linux
- C Programming
- Compiling and Linking

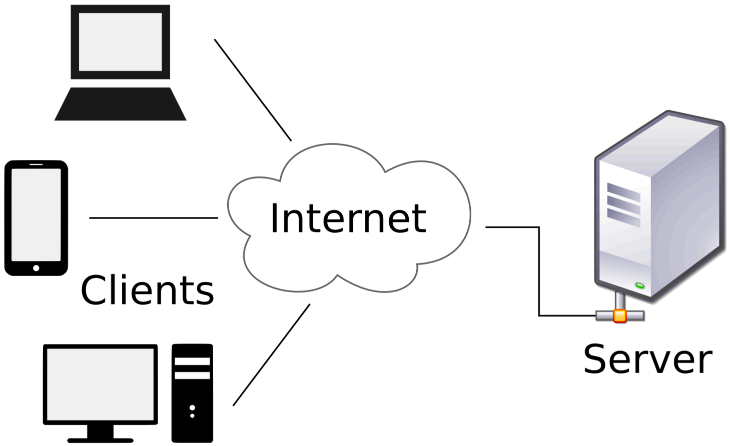
- Assignment 1

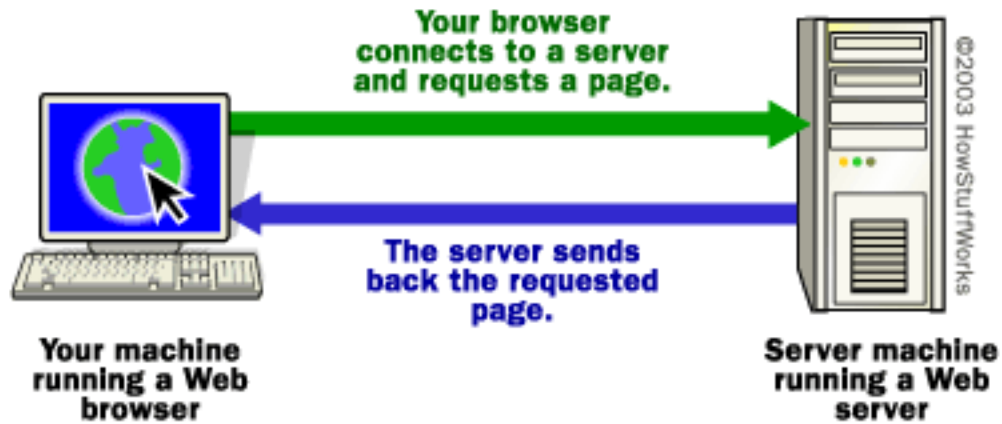
Computation Server

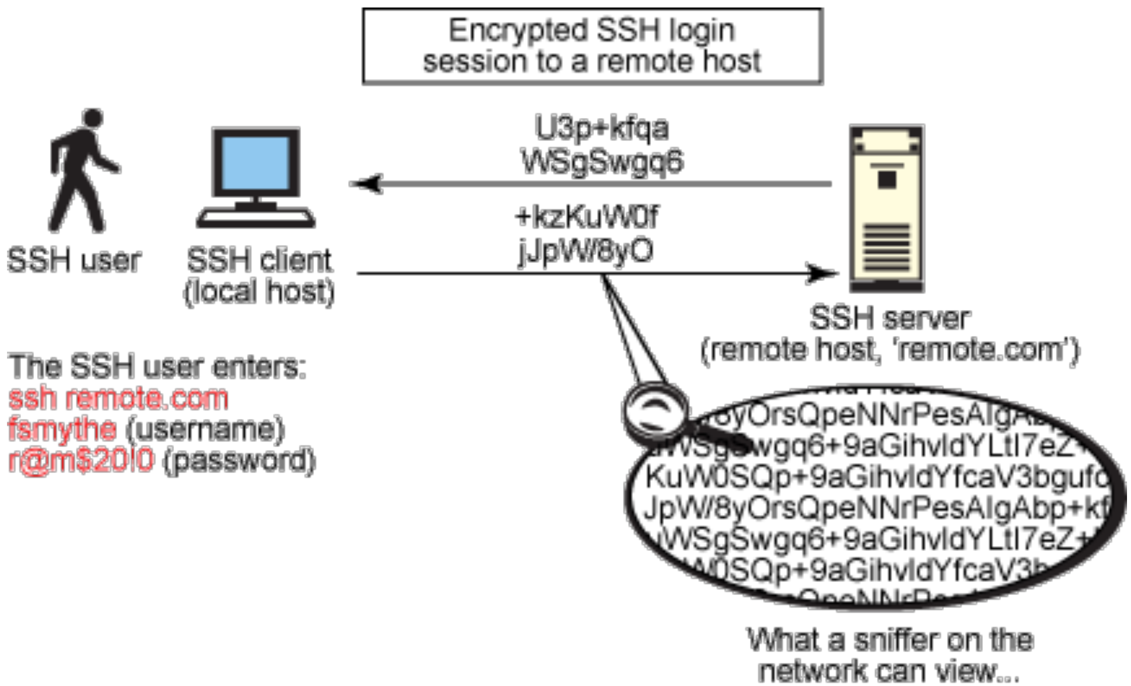
In the cold and dark server room!

Run Linux/Unix Operating System

Client/Server and SSH (Secure Shell)



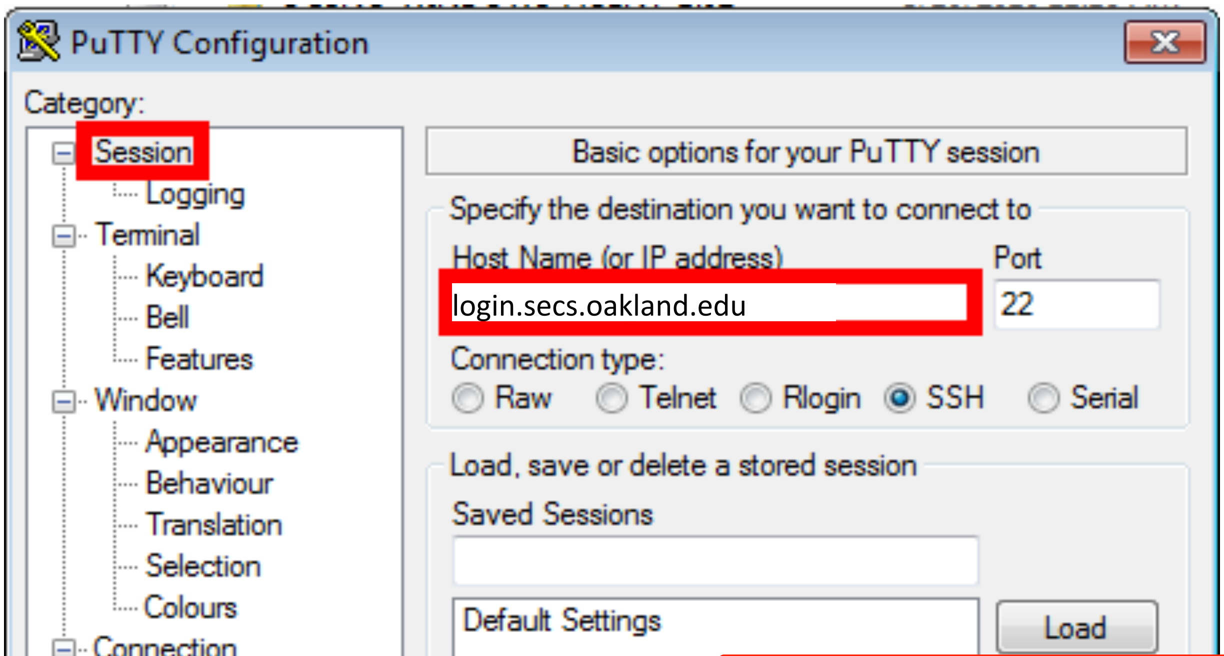


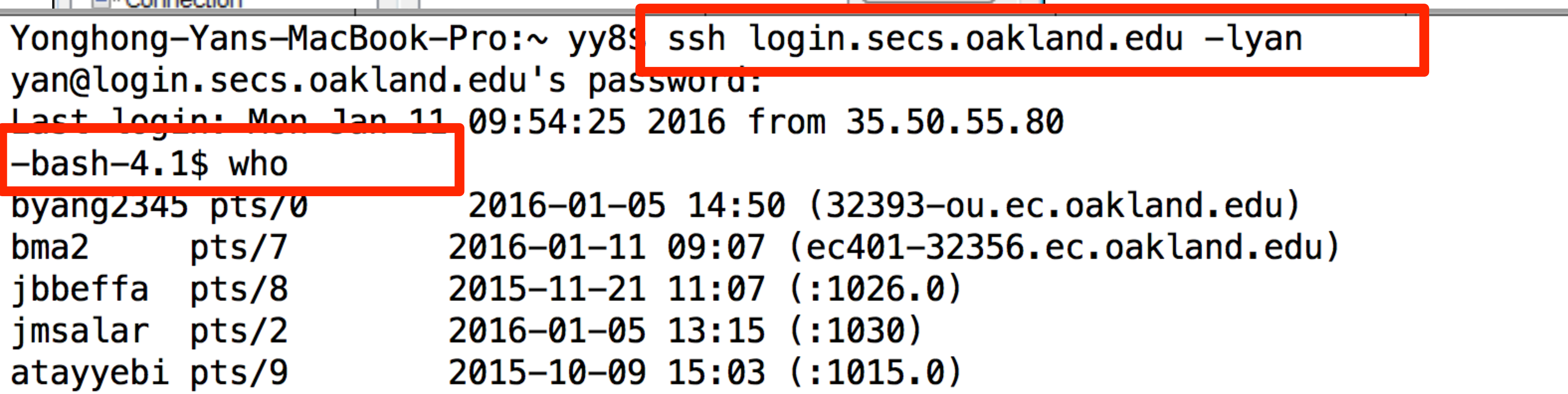


Machines

- http://cto.secs.oakland.edu/docs/pdf/linuxServers.pdf
- To connect to any available server
  - login.secs.oakland.edu
- Or you can connect directly with
  - ringo.secs.oakland.edu
  - harrison.secs.oakland.edu
  - paul.secs.oakland.edu
  - gpu.secs.oakland.edu
- SSH or putty
  - ssh <one of above machine name> -l<netid>
- Copy files or using H drive
  - scp or winscp
  - http://www.secs.oakland.edu/docs/pdf/accessNetworkDrive.pdf
- Need VPN if from home
  - http://secs.oakland.edu/docs/pdf/vpn.pdf

Putty SSH connection





Linux Basic Commands

It is all about dealing with files and folders
Linux folder: /home/yan/...

- ls (list files in the current folder)
  - $ ls -l
  - $ ls -a
  - $ ls -la
  - $ ls -l --sort=time
  - $ ls -l --sort=size –r
- cd (change directory to)
  - $ cd /usr/bin
- pwd (show current folder name)
  - $ pwd
- ~ (home folder)
  - $ cd ~
- *~user (home folder of a user)*
  - $ cd ~weesan
- What will "cd ~/weesan" do?

- rm (remove a filer/folder)
  - $ rm foo
  - $ rm -rf foo
  - $ rm -i foo
  - $ rm -- -foo
- cat (print the file contents to terminal)
  - $ cat /etc/motd
  - $ cat /proc/cpuinfo
- cp (create a copy of a file/folder)
  - $ cp foo bar
  - $ cp -a foo bar
- mv (move a file/folder to another location. Used also for renaming)
  - $ mv foo bar
- mkdir (create a folder)
  - $ mkdir foo

Basic Commands (cont)

- df (Disk usage)
  - $ df -h /
  - $ du -sxh ~/
- man (manual)
  - $ man ls
  - $ man 2 mkdir
  - $ man man
  - $ man -k mkdir
- Manpage sections
  - 1 User-level cmds and apps
    - /bin/mkdir
  - 2 System calls
    - int mkdir(const char *, …);
  - 3 Library calls
    - int printf(const char *, …);

Search a command or a file

- which
  - $ which ls
- whereis
  - $ whereis ls
- locate
  - $ locate stdio.h
  - $ locate iostream
- find
  - $ find / | grep stdio.h
  - $ find /usr/include | grep stdio.h

Smarty

1. **[Tab] key:** auto-complete the command sequence
2. ↑ **key:** to find previous command
3. **[Ctl]+r key**: to search previous command

Editing a File: Vi

- 2 modes
  - Input mode
    - ESC to back to cmd mode
  - Command mode
    - Cursor movement
      - h (left), j (down), k (up), l (right)
      - ^f (page down)
      - ^b (page up)
      - ^ (first char.)
      - $ (last char.)
      - G (bottom page)
      - :1 (goto first line)
    - Swtch to input mode
      - a (append)
      - i (insert)
      - o (insert line after
      - O (insert line before)
- Delete
  - dd (delete a line)
  - d10d (delete 10 lines)
  - d$ (delete till end of line)
  - dG (delete till end of file)
  - x (current char.)
- Paste
  - p (paste after)
  - P (paste before)
- Undo
  - u
- Search
  - /
- Save/Quit
  - :w (write)
  - :q (quit)
  - :wq (write and quit)
  - :q! (give up changes)

C Hello World

- vi hello.c
- Switch to editing mode: i or a
- Switching to control mode: ESC
- **Save a file: in control mode, :w**
- **To quit, in control mode, :q**
- To quit without saving, :q!
- Copy/paste a line: yy and then p, both from the current cursor
  - 5 line: 5yy and then p
- To delete a whole line, in control mode, : dd

- vi hello.c
- ls hello.c
- **gcc hello.c –o hello**
- **ls**
- ./hello

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv) {
  printf(“Hello World\n”);
  return 0;
}
```

C Syntax and Hello World

#include inserts another file. ".h" files are called "header" files. They contain declarations/definitions needed to interface to libraries and code in other ".c" files.

What do the < > mean?

A comment, ignored by the compiler

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
  printf("Hello World!\n");
  return 0;
}
```

The main() function is always where your program starts running.

Blocks of code ("lexical scopes") are marked by { … }

Return '0' from this function

Compilation Process in C

- Compilation process: gcc hello.c –o hello
  - Constructing an executable image for an application
  - FOUR stages
  - Command:
    gcc <options> <source_file.c>

- Compiler Tool
  - gcc (GNU Compiler)
    - man gcc (on Linux m/c)
  - icc (Intel C compiler)

4 Stages of Compilation Process

1. Preprocessing (Those with # …)
   - Expansion of Header files (#include …)
   - Substitute macros and inline functions (#define …)
2. Compilation
   - Generates assembly language
   - Verification of functions usage using prototypes
   - Header files: Prototypes declaration
3. Assembling
   - Generates re-locatable object file (contains m/c instructions)
   - nm app.o
     0000000000000000 T main
     U puts
   - nm or objdump tool used to view object files

4 Stages of Compilation Process (contd..)

4. Linking
   - Generates executable file (nm tool used to view exe file)
   - Binds appropriate libraries
     - Static Linking
     - Dynamic Linking (default)

- Loading and Execution (of an executable file)
  - Evaluate size of code and data segment
  - Allocates address space in the user mode and transfers them into memory
  - Load dependent libraries needed by program and links them
  - Invokes Process Manager → Program registration

4 Stages of Compilation Process

View the output of each stage using vi editor: e.g. vim hello.i

Preprocessing
gcc –E hello.c –o hello.i
hello.c → hello.i

Compilation (after preprocessing)

gcc –S hello.i –o hello.s

Assembling (after compilation)

gcc –c hello.s –o hello.o

Linking object files

gcc hello.o –o hello

Output → Executable (a.out)
Run → ./hello (Loader)

Compiling a C Program

- *gcc <options> program_name.c*

Four stages into one

- Options:

  -Wall: Shows all warnings

  -o output_file_name: By default a.out executable file is created when we compile our program with gcc. Instead, we can specify the output file name using "**-o**" option.

  -g: Include debugging information in the binary.

- man gcc

Linking Multiple files to make executable file

- Two programs, prog1.c and prog2.c for one single task
  - To make single executable file using following instructions

    First, compile these two files with option "**-c**"
    gcc -c prog1.c
    gcc -c prog2.c

    -c: Tells gcc to compile and assemble the code, but not link.

    We get two files as output, prog1.o and prog2.o
    Then, we can link these object files into single executable file using below instruction.

    gcc -o prog prog1.o prog2.o

    Now, the output is prog executable file.
    We can run our program using
    ./prog

Linking with other libraries

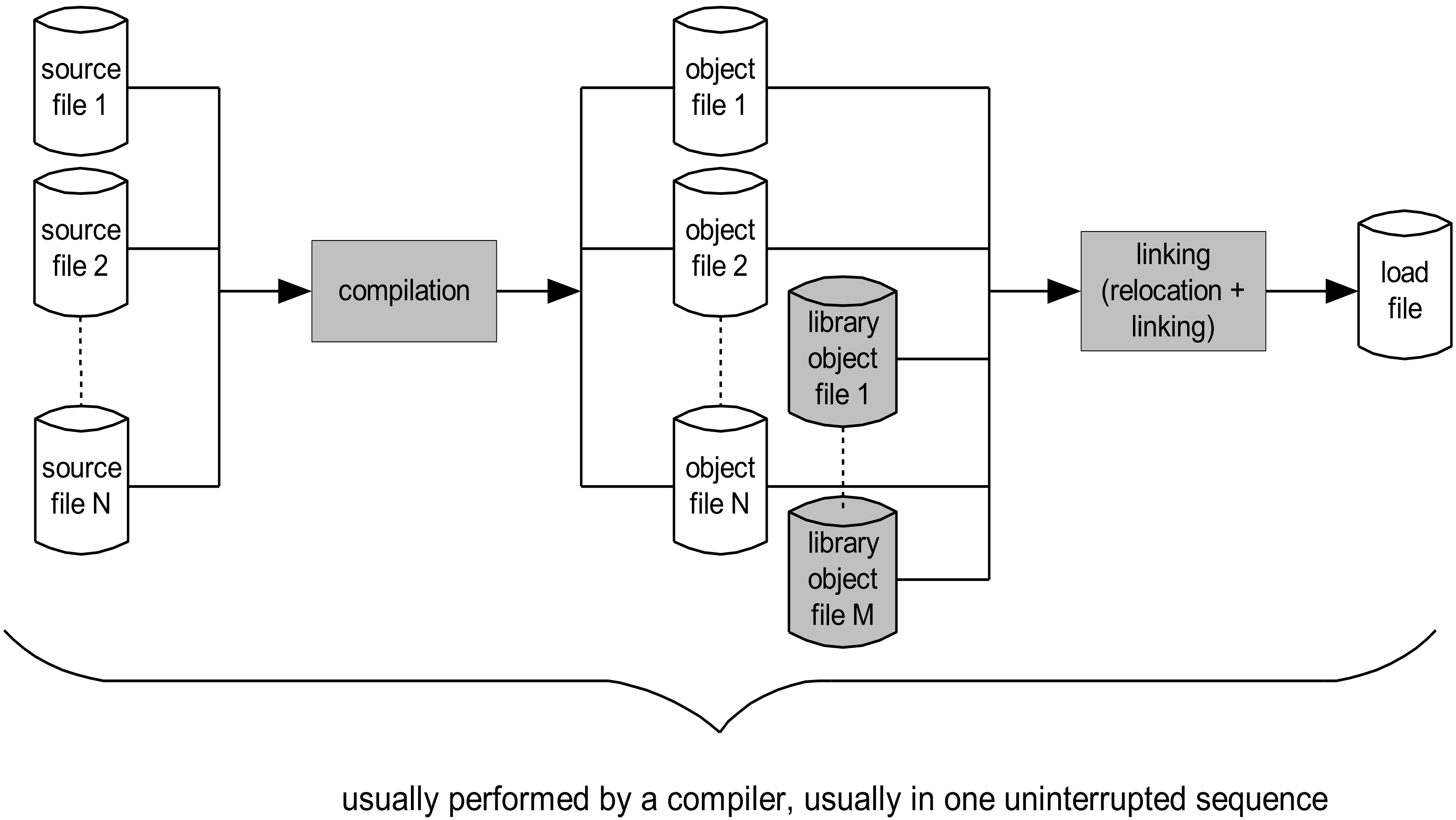
- Normally, compiler will read/link libraries from /usr/lib directory to our program during compilation process.
  - Library are precompiled object files

- To link our programs with libraries like pthreads and realtime libraries (rt library).
  - gcc <options> program_name.c -**lpthread -lrt**

    -lpthread: Link with pthread library → **libpthread.so** file
    -lrt: Link with rt library → **librt.so** file
    Option here is **"-l<library>"**

    Another option **"-L<dir>"** used to tell gcc compiler search for library file in given <dir> directory.

Compilation, Linking, Execution of C/C++ Programs



http://www.tenouk.com/ModuleW.html

Two useful commands

- ldd: e.g. “ldd a.out”, or “ldd hello”
  - List the name and the path of the dynamic library needed by a program
  - man ldd
- nm: e.g. “nm a.out”, “nm libc.so”
  - list **symbols** from object files
    - Symbols: function name, global variables that are exposed or reference by an object file.
  - man nm

sum.c and Assignment 1

- cp ~yan/sum.c ~ (copy sum.c file from my home folder to your home folder)
- gcc sum.c –o sum
- ./sum 102400
- vi sum.c

Or step by step

```
gcc -E sum.c -o sum.i
gcc -S sum.i -o sum.s
gcc -c sum.c -o sum.o
gcc sum.o -o sum
```

- Assignment 1
  - cp -r ~yan/Assignment_1 ~ : copy all the files to your home folder
  - View them from H drive
- Other system commands:
  - cat /proc/cpuinfo to show the CPU and #cores
  - top command to show system usage and memory

More on C Programming

Lexical Scoping

Every Variable is Defined within some scope. A Variable cannot be referenced by name (a.k.a. Symbol) from outside of that scope.

Lexical scopes are defined with curly braces { }.

- The scope of Function Arguments is the complete body of that function.
- The scope of Variables defined inside a function starts at the definition and ends at the closing brace of the containing block
- The scope of Variables defined outside a function starts at the definition and ends at the end of the file. Called "Global" Vars.

```
void p(char x)
{
              /* p,x */
  char y;
              /* p,x,y */
  char z;
              /* p,x,y,z */
}
              /* p */
char z;
              /* p,z */

void q(char a)
{
  char b;
              /* p,z,q,a,b */

  {
    char c;
              /* p,z,q,a,b,c */
  }

  char d;
  /* p,z,q,a,b,d (not c) */
}

/* p,z,q */
```

char b?

legal?

Comparison and Mathematical Operators

```
== equal to
<  less than
<= less than or equal
>  greater than
>= greater than or equal
!= not equal
&& logical and
|| logical or
!  logical not
```

```
+  plus
-  minus
*  mult
/  divide
%  modulo
```

```
&  bitwise and
|  bitwise or
^  bitwise xor
~  bitwise not
<< shift left
>> shift right
```

Beware division:

- 17/5=3, 17%5=2
- 5 / 10 = 0 *whereas* 5 / 10.0 = 0.5
- Division by 0 will cause a FPE(Float-point exception)

Don't confuse & and &&..
1 & 2 = 0 *whereas* 1 && 2 = <true>

The rules of precedence are clearly defined but often difficult to remember or non-intuitive. When in doubt, add parentheses to make it explicit.

Assignment Operators

```
x = y    assign y to x
x++      post-increment x
++x      pre-increment x
x--      post-decrement x
--x      pre-decrement x
```

```
x += y   assign (x+y) to x
x -= y   assign (x-y) to x
x *= y   assign (x*y) to x
x /= y   assign (x/y) to x
x %= y   assign (x%y) to x
```

Note the difference between ++x and x++ (high vs low priority (precedence)):

```
int x=5;
int y;
y = ++x;
/* x == 6, y == 6 */
```

```
int x=5;
int y;
y = x++;
/* x == 6, y == 5 */
```

Don’t confuse “=“ and “==“!

```
int x=5;
if (x==6)   /* false */
{
  /* ... */
}
/* x is still 5 */
```

```
int x=5;
if (x=6)   /* always true */
{
  /* x is now 6 */
}
/* ... */
```

A Quick Digression About the Compiler

```c
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

Preprocess

```c
__extension__ typedef  unsigned long long int
__dev_t;
__extension__ typedef  unsigned int   __uid_t;
__extension__ typedef  unsigned int   __gid_t;
__extension__ typedef  unsigned long int
__ino_t;
__extension__ typedef  unsigned long long int
__ino64_t;
__extension__ typedef  unsigned int
__nlink_t;
__extension__ typedef  long int   __off_t;
__extension__ typedef  long long int
__off64_t;
extern void flockfile (FILE *__stream)  ;
extern int ftrylockfile (FILE *__stream)  ;
extern void funlockfile (FILE *__stream)  ;
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

Compile

my_program

Compilation occurs in two steps: "Preprocessing" and "Compiling"

In 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.

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

The compiler then converts the resulting text (called translation unit) into binary code the CPU can execute.

C Memory Pointers

- To discuss memory pointers, we need to talk a bit about the concept of memory

- We'll conclude by touching on a couple of other C elements:
  - Arrays, typedef, and structs

The "memory"

Memory: similar to a big table of numbered slots where bytes of data are stored.

The number of a slot is its Address.
One byte Value can be stored in each slot.

Some data values span more than one slot, like the character string "Hello\n"

A Type provides a logical meaning to a span of memory. Some simple types are:

`char`	a single character (1 slot)
`char [10]`	an array of 10 characters
`int`	signed 4 byte integer
`float`	4 byte floating point
`int64_t`	signed 8 byte integer

Addr	Value
0	
1	
2	
3	
4	'H' (72)
5	'e' (101)
6	'l' (108)
7	'l' (108)
8	'o' (111)
9	'\n' (10)
10	'\0' (0)
11	
12	

What is a Variable?

symbol table?

A Variable names a place in memory where you store a Value of a certain Type.

You first Declare a variable by giving it a name and specifying its type and optionally an initial value

declare vs. define

```
char x;
char y='e';
```

Variable x declared but undefined

Initial value

Name

What names are legal?

Type is single character (char)

extern? static? const?

The compiler puts x and y somewhere in memory.

Symbol	Addr	Value
	0	
	1	
	2	
	3	
x	4	Some garbage
y	5	'e' (101)
	6	
	7	
	8	
	9	
	10	
	11	
	12	

Multi-byte Variables

Different types require different amounts of memory. Most architectures store data on "**word boundaries**", or even multiples of the size of a primitive data type (int, char)

```
char x;
char y='e';
int z = 0x01020304;
```

0x means the constant is written in hex

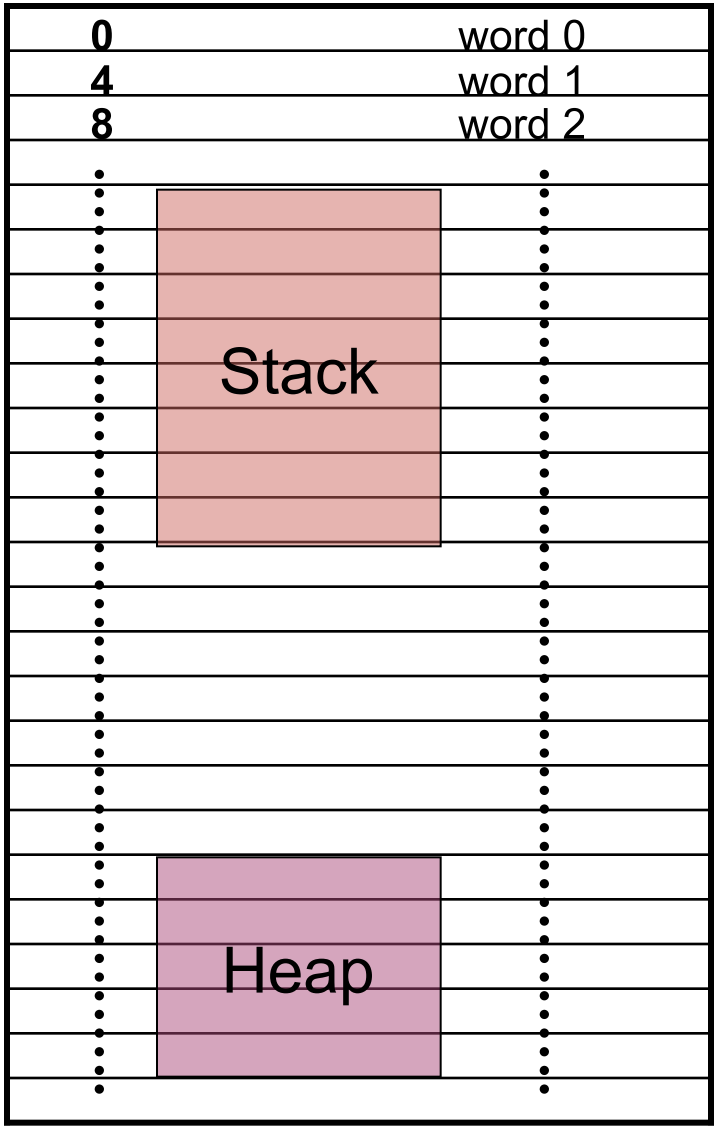
padding

An int requires 4 bytes

Symbol	Addr	Value
	0	
	1	
	2	
	3	
x	4	Some garbage
y	5	'e' (101)
	6	
	7	
z	8	4
	9	3
	10	2
	11	1
	12	

Memory, a more detailed view….

- A sequential list of words, starting from 0.
- On 32bit architectures (e.g. Win32): each word is 4 bytes.
- Local variables are stored in the stack
- Dynamically allocated memory is set aside on the heap (more on this later…)
- For multiple-byte variables, the address is that of the smallest byte (little endian).



Example

```
#include <iostream>

int main() {
   char c[10];
   int d[10];
   int* darr;

   darr = (int *)(malloc(10*sizeof(int)));
   size_t sizeC = sizeof(c);
   size_t sizeD = sizeof(d);
   size_t sizeDarr = sizeof(darr);

   free(darr);
   return 0;
}
```

What is the value of:
- sizeC
- sizeD
- sizeDarr

<u>NOTE</u>: ***sizeof*** is a compile-time operator that returns the size, **in multiples of the size of *char***, of the variable or parenthesized type-specifier that it precedes.

Can a C function modify its arguments?

What if we wanted to implement a function pow_assign() that *modified* its argument, so that these are equivalent:

```
float p = 2.0;
/* p is 2.0 here */
p = pow(p, 5);
/* p is 32.0 here */
```

?

```
float p = 2.0;
/* p is 2.0 here */
pow_assign(p, 5);
/* Is p is 32.0 here ? */
```

Native function, to use you need #include <math.h>

Would this work?

```
void pow_assign(float x, uint exp)
{
  float result=1.0;
  int i;
  for (i=0; (i < exp); i++) {
    result = result * x;
  }
  x = result;
}
```

In C you can't change the value of any variable passed as an argument in a function call…

Pass by value

```
void pow_assign(float x, uint exp)
{
  float result=1.0;
  int i;
  for (i=0; (i < exp); i++) {
    result = result * x;
  }
  x = result;
}

// a code snippet that uses above
// function
{
  float p=2.0;
  pow_assign(p, 5);
  // the value of p is 2 here…
}
```

In C, all arguments are passed by value

Keep in mind: pass by value requires the variable to be copied. That copy is then passed to the function. Sometime generating a copy can be expensive…

But, what if the argument is the *address* of a variable?

C Pointers

- What is a pointer?
  - A variable that contains the memory address of another variable or of a function
- In general, it is safe to assume that on 32 bit architectures pointers occupy one word
  - Pointers to int, char, float, void, etc. ("int*", "char*", "*float", "void*"), they all occupy 4 bytes (one word).
- Pointers: *very* many bugs in C programs are traced back to mishandling of pointers…

Pointers (cont.)

- The need for pointers
  - Needed when you want to modify a variable (its value) inside a function
    - The pointer is passed to that function as an argument
  - Passing large objects to functions without the overhead of copying them first
  - Accessing memory allocated on the heap
  - Referring to functions, i.e. function pointers

Pointer Validity

A Valid pointer is one that points to memory that your program controls.
Using invalid pointers will cause non-deterministic behavior
- Very often the code will crash with a SEGV, that is, Segment Violation, or Segmentation Fault.

There are two general causes for these errors:
- Coding errors that end up setting the pointer to a strange number
- Use of a pointer that was at one time valid, but later became invalid

Good practice:
- Initialize pointers to 0 (or NULL). NULL is never a valid pointer value, but it is known to be invalid and means "no pointer set".

```
char * get_pointer()
{
  char x=0;
  return &x;
}

{
  char * ptr = get_pointer();
  *ptr = 12;  /* valid? */
}
```

Will *ptr* be valid or invalid?

Answer: No, it’s invalid…

A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

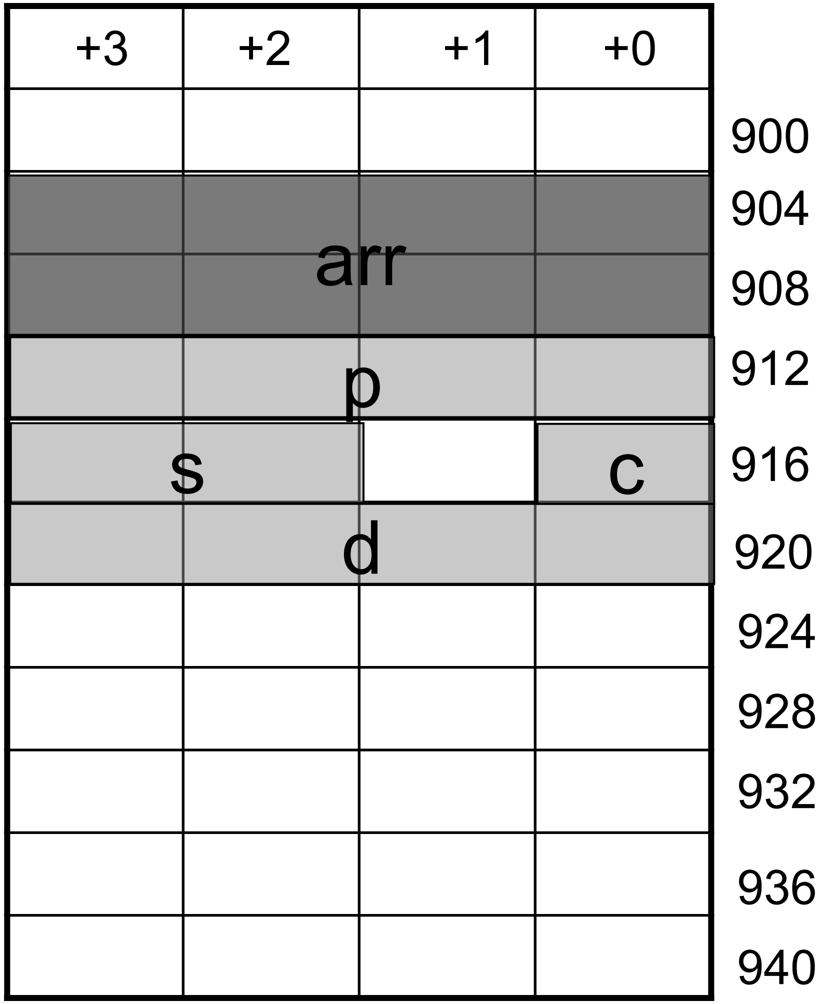
```
char * get_pointer()
{
  char x=0;
  return &x;
}

int main()
{
  char * ptr = get_pointer();
  *ptr = 12;  /* valid? */
  other_function();
  return 0;
}
```

But now, `ptr` points to a location that’s no longer in use, and will be reused the next time a function is called!

Here is what I get in DevStudio when compiling:
main.cpp(6) : warning C4172: returning address of local variable or temporary

Example: What gets printed out?

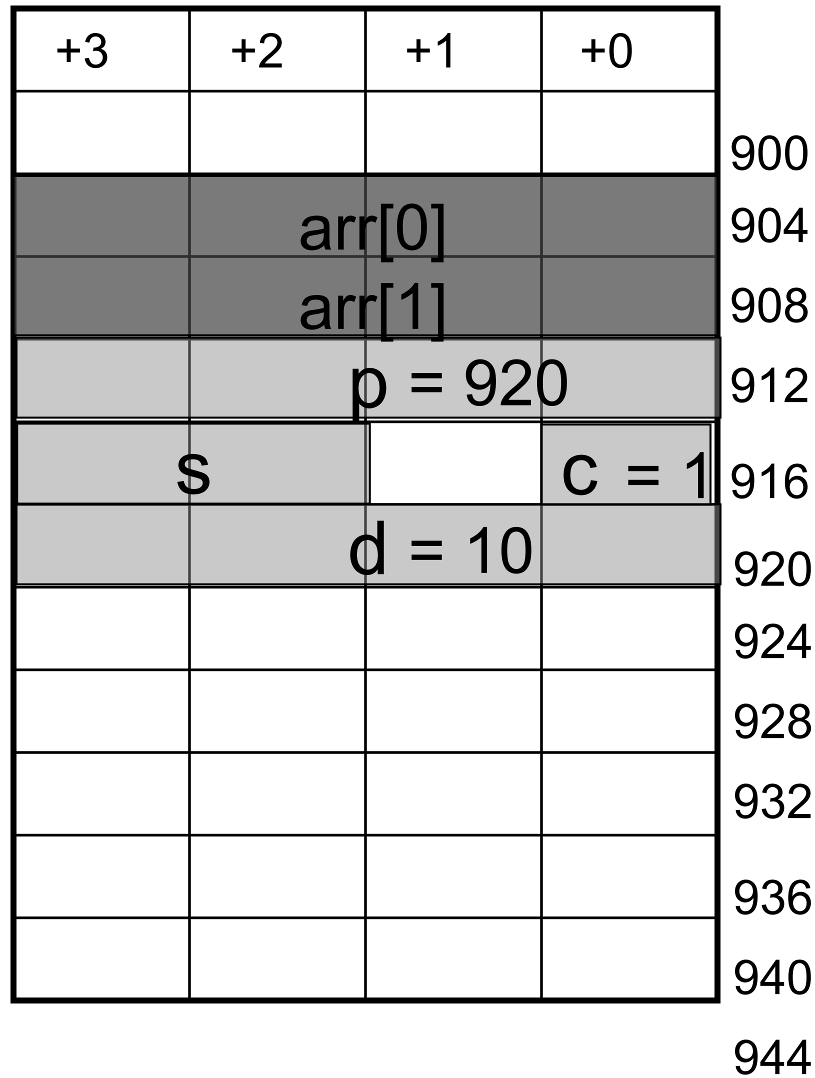
```
int main() {
    int d;
    char c;
    short s;
    int* p;
    int arr[2];
    printf( "%p, %p, %p, %p, %p\n",&d, &c, &s, &p, arr );
    return 0;
}
```



- NOTE: Here &d = 920 (in practice a 4-byte hex number such as 0x22FC3A08)

Example:
Usage of Pointers & Pointer Arithmetic

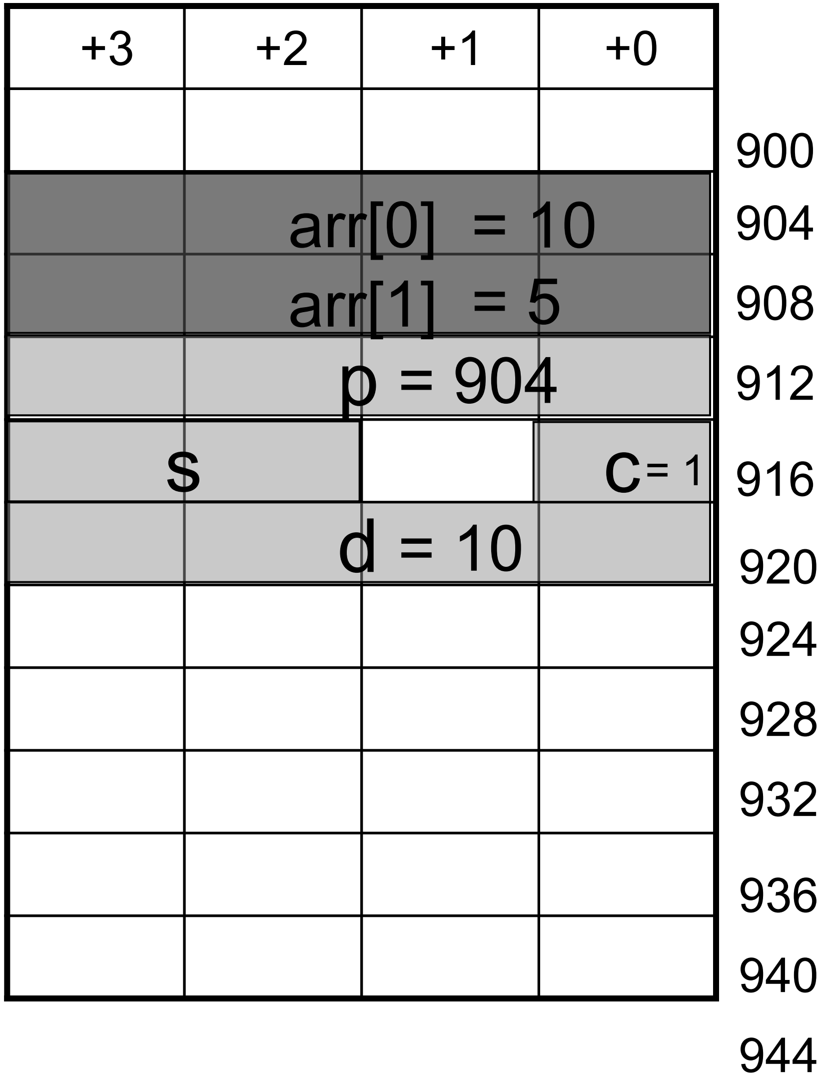
```
int main() {
    int d;
    char c;
    short s;
    int* p;
    int arr[2];

    p = &d;
    *p = 10;
    c = (char)1;

    p = arr;
    *(p+1) = 5;
    p[0] = d;

    *( (char*)p + 1 ) = c;

    return 0;
}
```



Q: What are the values stored in arr? [assume little endian architecture]

Example [Cntd.]

```
p = &d;
*p = 10;
c = (char)1;

p = arr;
*(p+1) = 5;  // int* p;
p[0] = d;

*( (char*)p + 1 ) = c;
```



Question: arr[0] = ?

Use of pointers, another example…

- Pass pointer parameters into function

```
void swap(int *px, int *py)
{
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}
int a = 5;
int b = 6;
swap(&a, &b);
```

- What will happen here?

```
int * a;
int * b;
swap(a, b);
```

Dynamic Memory Allocation (on the Heap)

- Allows the program to determine how much memory it needs *at run time* and to allocate exactly the right amount of storage.
  - It is your responsibility to clean up after you (free the dynamic memory you allocated)

- The region of memory where dynamic allocation and deallocation of memory can take place is called the heap.

Recall Discussion on Dynamic Memory Allocation

Recall that variables are allocated statically by having declared with a given size. This allocates them in the stack.

Allocating memory at run-time requires dynamic allocation. This allocates them on the heap.

sizeof() reports the size of a type in bytes

```
int * alloc_ints(size_t requested_count)
{
  int * big_array;
  big_array = (int *)calloc(requested_count, sizeof(int));
  if (big_array == NULL) {
    printf("can't allocate %d ints: %m\n", requested_count);
    return NULL;
  }

  /* big_array[0] through big_array[requested_count-1] are
   * valid and zeroed. */
  return big_array;
}
```

calloc() allocates memory for N elements of size k

Returns NULL if can't alloc

It's OK to return this pointer. It will remain valid until it is freed with free(). However, it's a bad practice to return it (if you need is somewhere else, declare and define it there…)

Caveats with Dynamic Memory

Dynamic memory is useful. But it has several caveats:

Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and free()'d when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory causes "memory leak".

Whereas the compiler enforces that reclaimed stack space can no longer be reached, it is easy to accidentally keep a pointer to dynamic memory that was freed. Whenever you free memory you must be certain that you will not try to use it again.

Because dynamic memory always uses pointers, there is generally no way for the compiler to statically verify usage of dynamic memory. This means that errors that are detectable with static allocation are not with dynamic

Data Structures

- A data structure is a collection of one or more variables, possibly of different types.

- An example of student record

```
struct StudRecord {
        char name[50];
        int id;
        int age;
        int major;
};
```

Data Structures (cont.)

- A data structure is also a data type

```
struct StudRecord my_record;
struct StudRecord * pointer;
pointer = & my_record;
```

- Accessing a field inside a data structure

```
my_record.id = 10;
        // or
pointer->id = 10;
```

Data Structures (cont.)

- Allocating a data structure instance

This is a new type now

```
struct StudRecord* pStudentRecord;
pStudentRecord = (StudRecord*)malloc(sizeof(struct StudRecord));
pStudentRecord ->id = 10;
```

- IMPORTANT:
  - Never calculate the size of a data structure yourself. Rely on the sizeof() function
  - Example: Because of memory padding, the size of "struct StudRecord" is 64 (instead of 62 as one might estimate)

The “typedef” Construct

Using typedef to improve readability…

```
struct StudRecord {
  char name[50];
  int id;
  int age;
  int major;
};

typedef struct StudRecord RECORD_t;

int main() {
   RECORD_t   my_record;
   strcpy_s(my_record.name, “Joe Doe”);
   my_record.age = 20;
   my_record.id = 6114;

   RECORD_t* p = &my_record;
   p->major = 643;
   return 0;
}
```

Arrays

Arrays in C are composed of a particular type, laid out in memory in a repeating pattern. Array elements are accessed by stepping forward in memory from the base of the array by a multiple of the element size.

```
/* define an array of 10 chars */
char x[5] = {‘t’,’e’,’s’,’t’,’\0’};

/* access element 0, change its value */
x[0] = ‘T’;

/* pointer arithmetic to get elt 3 */
char elt3 = *(x+3);  /* x[3] */

/* x[0] evaluates to the first element;
 * x evaluates to the address of the
 * first element, or &(x[0]) */

/* 0-indexed for loop idiom */
#define COUNT 10
char y[COUNT];
int i;
for (i=0; i<COUNT; i++) {
  /* process y[i] */
  printf(“%c\n”, y[i]);
}
```

Brackets specify the count of elements. Initial values optionally set in braces.

Arrays in C are 0-indexed (here, 0…4)

x[3] == *(x+3) == ‘t’   (*notice, it’s not* ‘s’!)

For loop that iterates from 0 to COUNT-1.

Symbol	Addr	Value
char x [0]	100	‘t’
char x [1]	101	‘e’
char x [2]	102	‘s’
char x [3]	103	‘t’
char x [4]	104	‘\0’

Q: What’s the difference between “char x[5]” and a declaration like “char *x”?

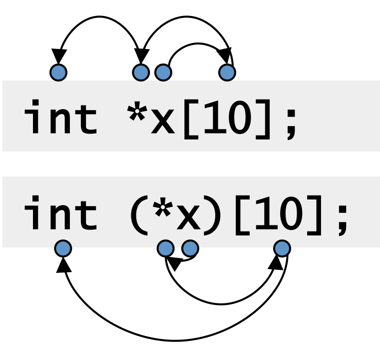
How to Parse and Define C Types

At this point we have seen a few basic types, arrays, pointer types, and structures. So far we've glossed over how types are named.

```
int x;         /* int;                     */  typedef int T;
int *x;        /* pointer to int;          */  typedef int *T;
int x[10];     /* array of ints;           */  typedef int T[10];
int *x[10];    /* array of pointers to int; */ typedef int *T[10];
int (*x)[10];  /* pointer to array of ints; */ typedef int (*T)[10];
```

typedef defines a new type

C type names are parsed by starting at the type name and working <u>outwards</u> according to the rules of precedence:




```
x is
an array of
pointers to
int
```

```
x is
a pointer to
an array of
int
```

Arrays are the primary source of confusion. When in doubt, use extra parens to clarify the expression.

REMEMBER THIS: (), which stands for function, and [], which stands for array, have higher precedence than *, which stands for pointer

Function Types

Another less obvious construct is the "**pointer to function**" type.
For example, qsort: (a sort function in the standard library)

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

The last argument is a comparison function

```
/* function matching this type: */
int cmp_function(const void *x, const void *y);

/* typedef defining this type: */
typedef int (*cmp_type) (const void *, const void *);

/* rewrite qsort prototype using our typedef */
void qsort(void *base, size_t nmemb, size_t size, cmp_type compar);
```

const means the function is not allowed to modify memory via this pointer.

size_t is an unsigned int

void * is a pointer to memory of unknown type.

Row Major and Column Major

REAL * A	1	2	3	4	5	6	7	8	9

Row major

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Column major

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

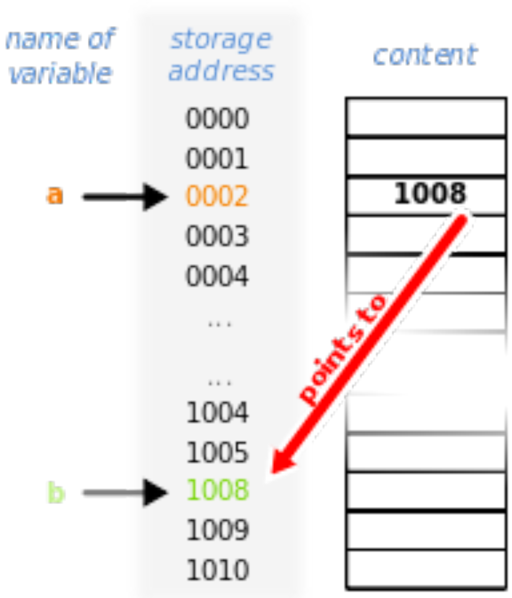
References

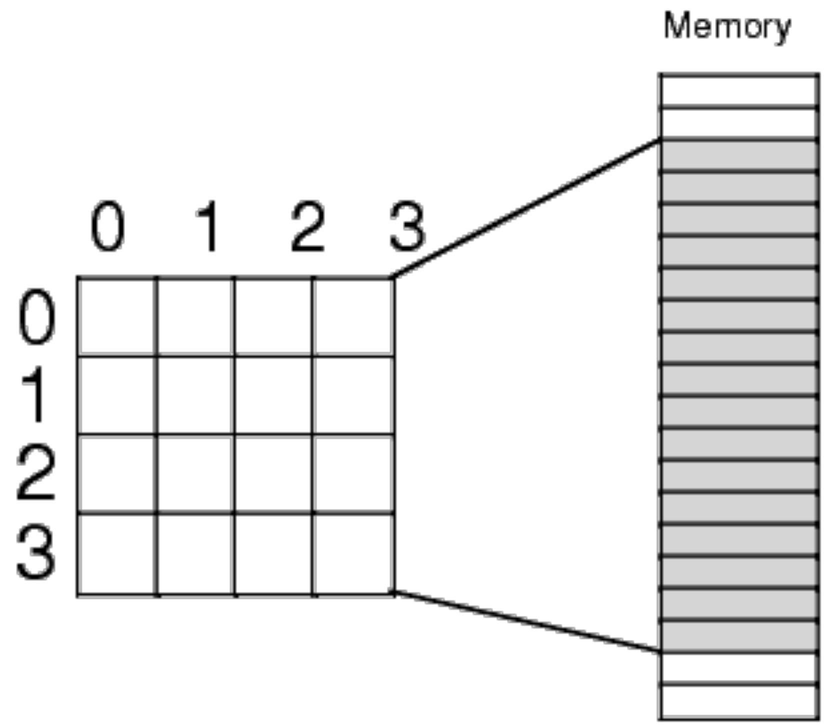
- Linux/Unix Introduction
  – http://www.ee.surrey.ac.uk/Teaching/Unix/
- VI Editor
  – https://www.cs.colostate.edu/helpdocs/vi.html
- C Programming Tutorial
  – http://www.cprogramming.com/tutorial/c-tutorial.html
- Compiler, Assembler, Linker and Loader: A Brief Story
  – http://www.tenouk.com/ModuleW.html

Backup and More

Sequential Memory Regions vs Multi-dimensional Array

- Memory is a sequentially accessed using the address of each byte/word

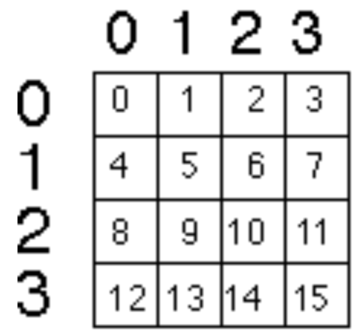


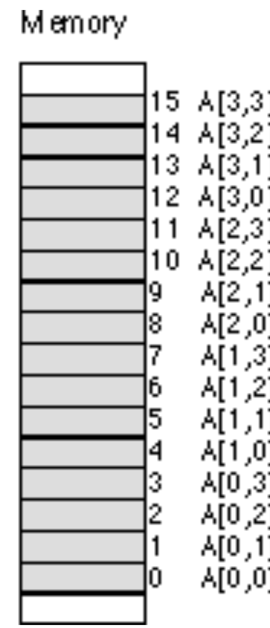


Vector/Matrix and Array in C

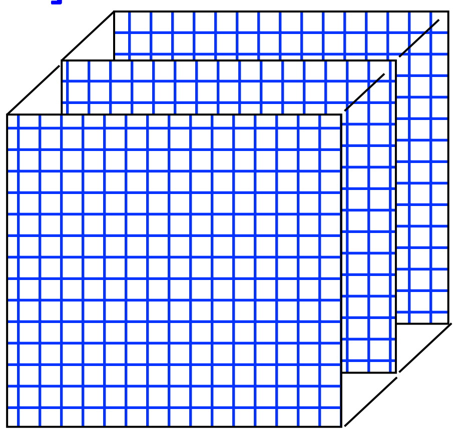
- C has row-major storage for multiple dimensional array
  - A[2,2] is followed by A[2,3]

char A[4][4]





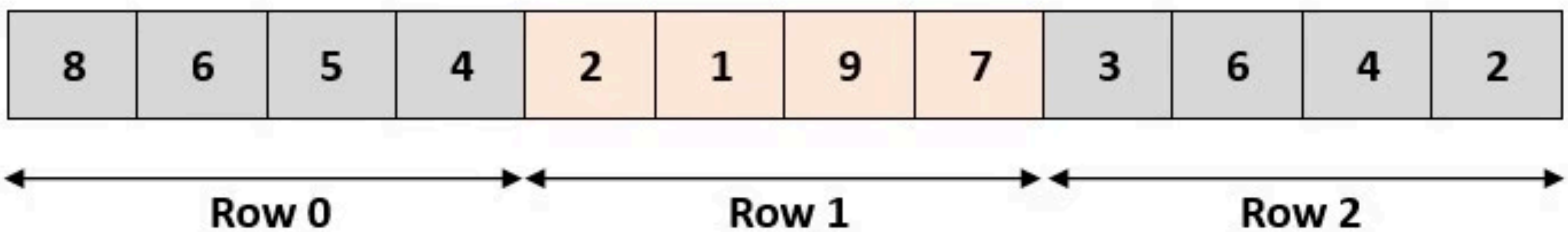
- 3-dimensional array
  - B[3][100][100]



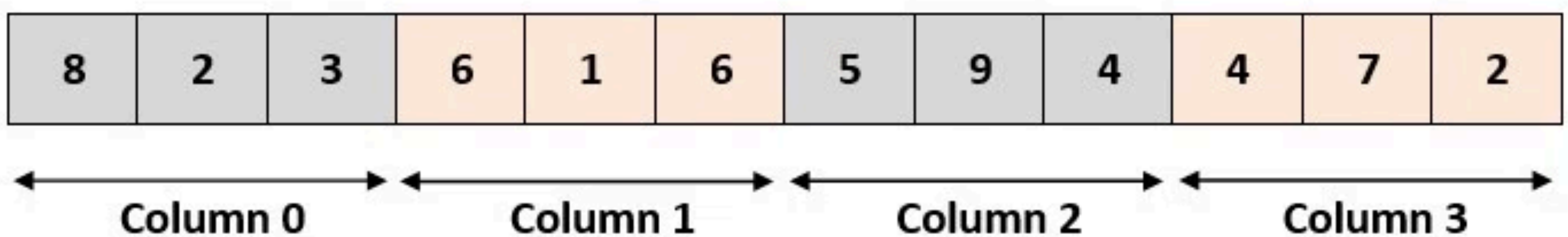
Store Array in Memory in Row Major or Column Major

8	6	5	4
2	1	9	7
3	6	4	2

Row-Major (Row Wise Arrangement)



Column-Major (Column Wise Arrangement)



For a Memory Region to Store Data for an Array in Either Row or Col Major

8	6	5	4	2	1	9	7	3	6	4	2

3 X 4

Row-Major (Row Wise Arrangement)

8	6	5	4
2	1	9	7
3	6	4	2

3 X 4

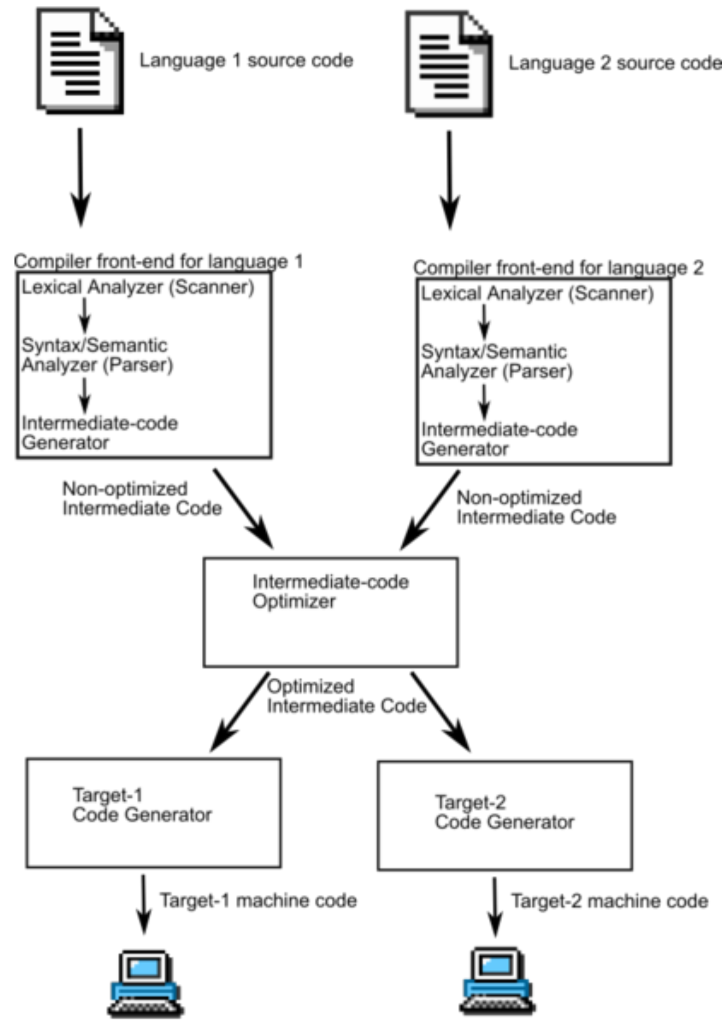
Column-Major (Column Wise Arrangement)

8	4	9	6
6	2	7	4
5	1	3	2

Compiler

- A **programming language** is an artificial language that can be used to control the behavior of a machine, particularly a computer.

- A **compiler** is a computer program (or set of programs) that translates text written in a computer language (the *source language*) into another computer language (the *target language*). The original sequence is usually called the *source code* and the output called *object code*. Commonly the output has a form suitable for processing by other programs (e.g., a linker), but it may be a human-readable text file.



Debug and Performance Analysis

- **Debugging** is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware thus making it behave as expected.

- In software engineering, **performance analysis** (a field of dynamic program analysis) is the investigation of a program's behavior using information gathered as the program runs, as opposed to static code analysis. The usual goal of performance analysis is to determine which parts of a program to optimize for speed or memory usage.

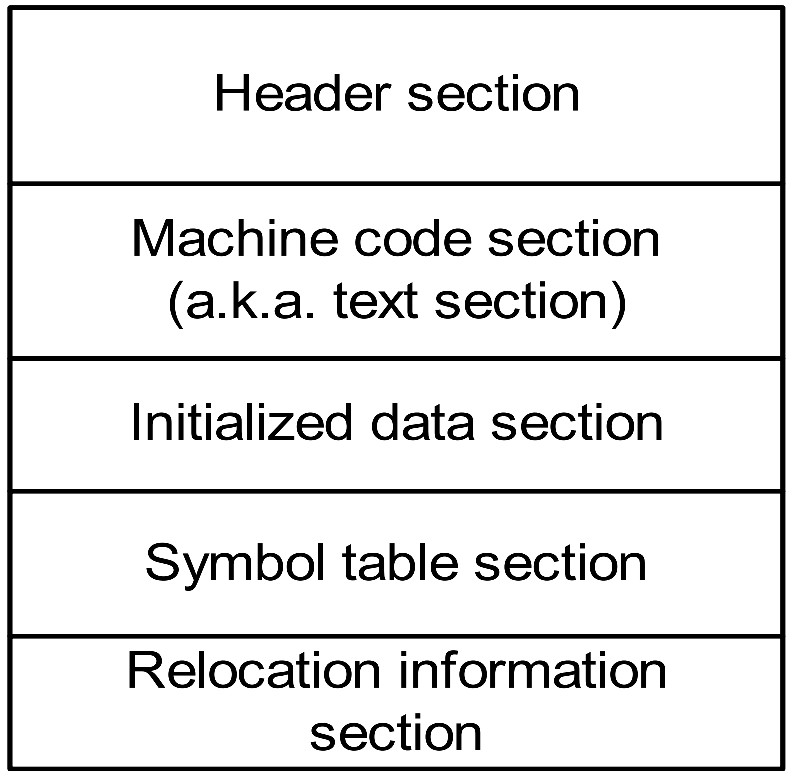
- A **profiler** is a performance analysis tool that measures the behavior of a program as it runs, particularly the frequency and duration of function calls. The output is a stream of recorded events (a **trace**) or a statistical summary of the events observed (a **profile**).

Optimization

- In computing, **optimization** is the process of modifying a system to make some aspect of it work more efficiently or use less resources. For instance, a computer program may be optimized so that it executes more rapidly, or is capable of operating within a reduced amount of memory storage, or draws less battery power in a portable computer. The system may be a single computer program, a collection of computers or even an entire network such as the Internet.

(http://en.wikipedia/org/wiki/Optimization_%28computer_science%29)

Object module structure



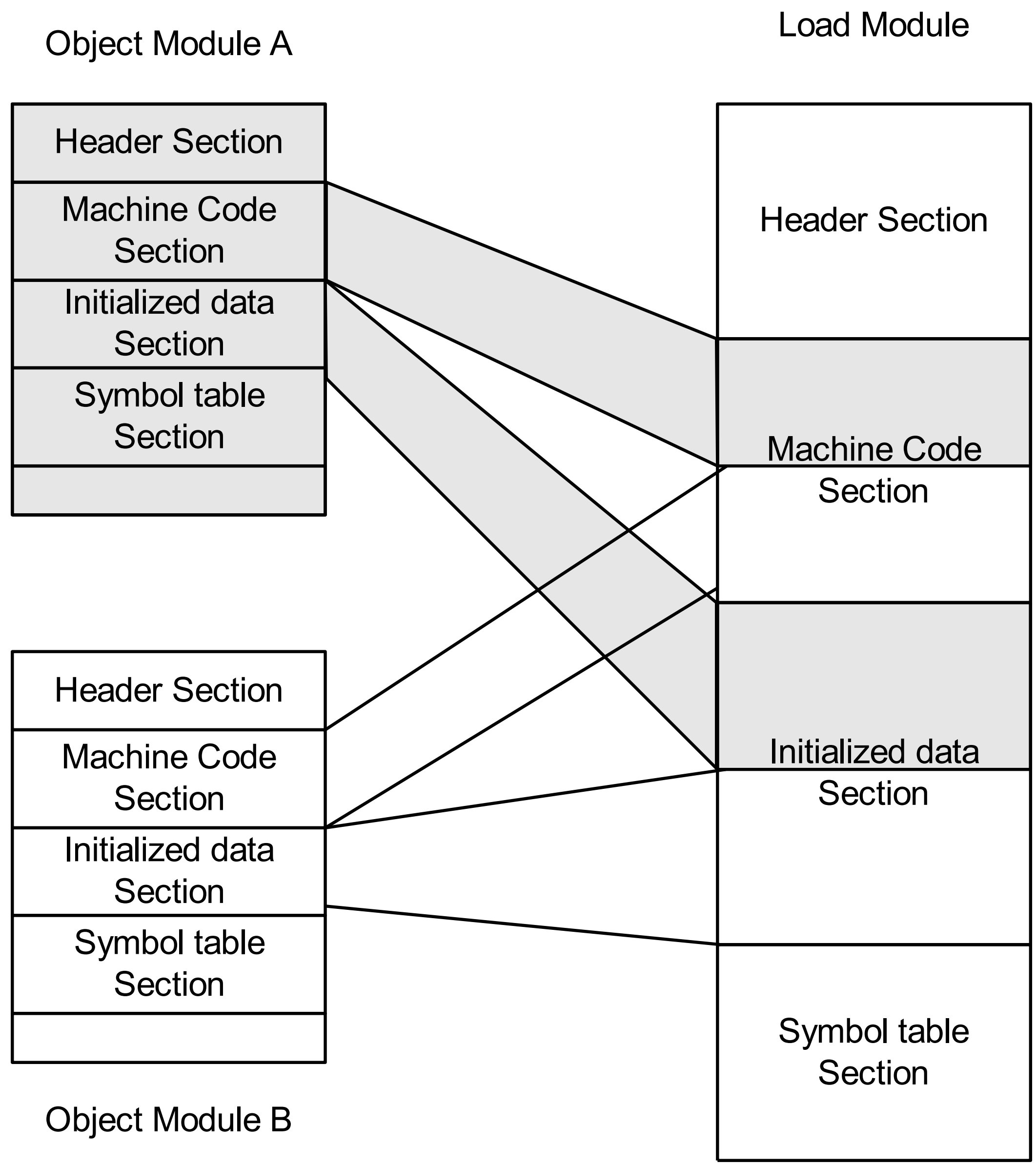
A sample C program:

```
#include <stdio.h>

int a[10]={0,1,2,3,4,5,6,7,8,9};
int b[10];

void main()
{
   int i;
   static int k = 3;

   for(i = 0; i < 10; i++) {
    printf("%d\n",a[i]);
    b[i] = k*a[i];
   }

}
```

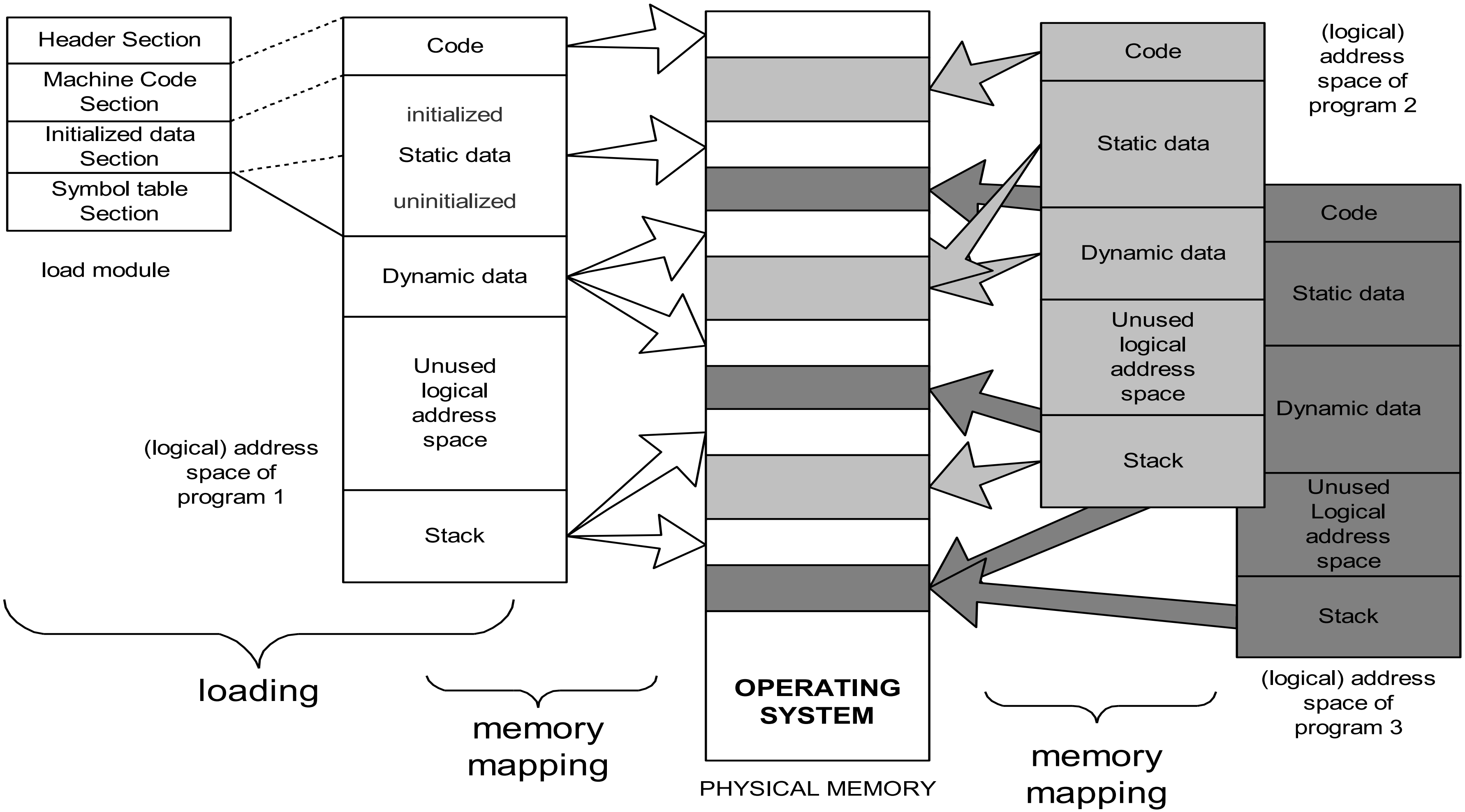
Object module of the sample C program:

Offset	*Contents*	*Comment*
Header section		
0	124	number of bytes of Machine code section
4	44	number of bytes of initialized data section
8	40	number of bytes of Uninitialized data section (array `b[]`) (*not part of this object module*)
12	60	number of bytes of Symbol table section
16	44	number of bytes of Relocation information section
Machine code section (124 bytes)		
20	X	code for the top of the `for` loop (36 bytes)
56	X	code for call to `printf()` (22 bytes)
68	X	code for the assignment statement (10 bytes)
88	X	code for the bottom of the `for` loop (4 bytes)
92	X	code for exiting `main()` (52 bytes)
Initialized data section (44 bytes)		
144	0	beginning of array `a[]`
148	1	
:		
176	8	
180	9	end of array `a[]` (40 bytes)
184	3	variable `k` (4 bytes)
Symbol table section (60 bytes)		
188	X	array `a[]` : offset 0 in Initialized data section (12 bytes)
200	X	variable `k` : offset 40 in Initialized data section (10 bytes)
210	X	array `b[]` : offset 0 in Uninitialized data section (12 bytes)
222	X	`main` : offset 0 in Machine code section (12 bytes)
234	X	`printf` : external, used at offset 56 of Machine code section (14 bytes)
Relocation information section (44 bytes)		
248	X	relocation information

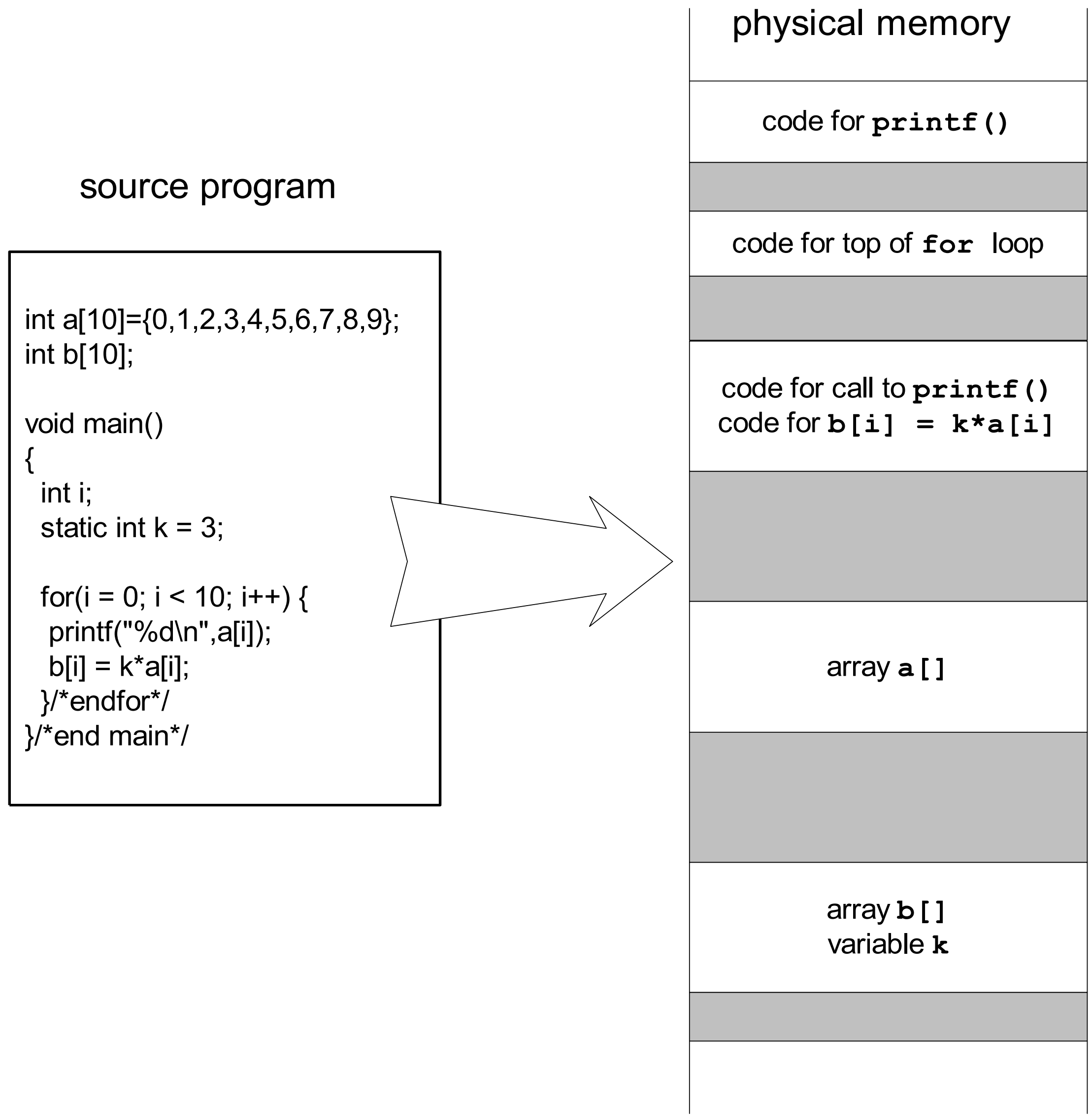
Creation of load module



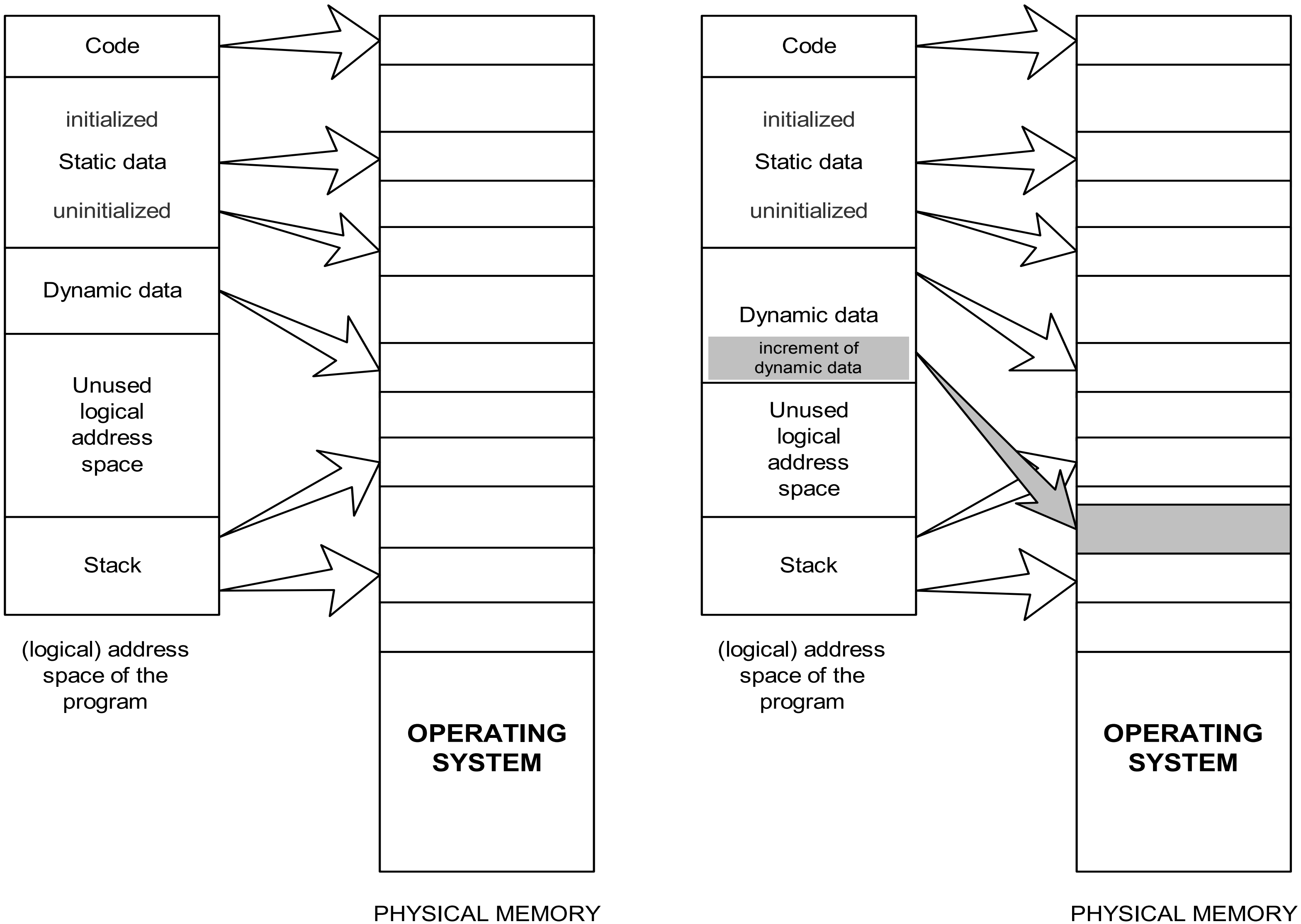
Loading and memory mapping



From source program to “placement” in memory during execution



Dynamic memory allocation



Before dynamic memory allocation After dynamic memory allocation

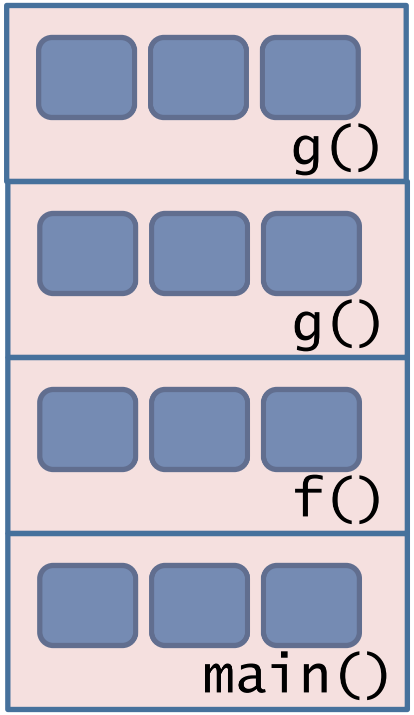
Overview of memory management

- Stack-allocated memory
  - When a function is called, memory is allocated for all of its parameters and local variables.
  - Each active function call has memory on the stack (with the current function call on top)
  - When a function call terminates, the memory is deallocated (“freed up”)

- Ex: `main()` calls `f()`,

  `f()` calls `g()`

  `g()` recursively calls `g()`



Overview of memory management

- Heap-allocated memory
  - This is used for *persistent* data, that must survive beyond the lifetime of a function call
    - global variables
    - dynamically allocated memory – C statements can create new heap data (similar to new in Java/C++)
  - Heap memory is allocated in a more complex way than stack memory
  - Like stack-allocated memory, the underlying system determines where to get more memory – the programmer doesn't have to search for free memory space!

Note: `void *` denotes a generic pointer type

Allocating new heap memory

```
void *malloc(size_t size);
```

- Allocate a block of `size` bytes,
  return a pointer to the block
  (`NULL` if unable to allocate block)

```
void *calloc(size_t num_elements, size_t element_size);
```

- Allocate a block of `num_elements * element_size` bytes,
  initialize every byte to zero,
  return pointer to the block
  (`NULL` if unable to allocate block)

Allocating new heap memory

```
void *realloc(void *ptr, size_t new_size);
```

- Given a previously allocated block starting at `ptr`,
  - change the block size to `new_size`,
  - return pointer to resized block
    - If block size is increased, contents of old block may be copied to a completely different region
    - In this case, the pointer returned will be different from the `ptr` argument, and `ptr` will no longer point to a valid memory region
- If `ptr` is `NULL`, `realloc` is identical to `malloc`

- Note: may need to cast return value of `malloc/calloc/realloc`:

```
char *p = (char *) malloc(BUFFER_SIZE);
```

Deallocating heap memory

```
void free(void *pointer);
```

- Given a pointer to previously allocated memory,
  - put the region back in the heap of unallocated memory

- Note: easy to forget to free memory when no longer needed...
  - especially if you're used to a language with "garbage collection" like Java
  - This is the source of the notorious "memory leak" problem
  - Difficult to trace – the program will run fine for some time, until suddenly there is no more memory!

Memory errors

- Using memory that you have not initialized
- Using memory that you do not own
- Using more memory than you have allocated
- Using faulty heap memory management

Using memory that you have not initialized

- Uninitialized memory read
- Uninitialized memory copy
  - not necessarily critical – unless a memory read follows

```
void foo(int *pi) {
  int j;
  *pi = j;
  /* UMC: j is uninitialized, copied into *pi */
}
void bar() {
  int i=10;
  foo(&i);
  printf("i = %d\n", i);
  /* UMR: Using i, which is now junk value */
}
```

Using memory that you don't own

- Null pointer read/write
- Zero page read/write

```
typedef struct node {
  struct node* next;
  int val;
} Node;

int findLastNodeValue(Node* head) {
  while (head->next != NULL) { /* Expect NPR */
   head = head->next;
  }
  return head->val; /* Expect ZPR */
}
```

What if head is NULL?

Using memory that you don’t own

- Invalid pointer read/write
  - Pointer to memory that hasn’t been allocated to program

```
void genIPR() {
  int *ipr = (int *) malloc(4 * sizeof(int));
  int i, j;
  i = *(ipr - 1000); j = *(ipr + 1000); /* Expect IPR */
  free(ipr);
}

void genIPW() {
  int *ipw = (int *) malloc(5 * sizeof(int));
  *(ipw - 1000) = 0; *(ipw + 1000) = 0; /* Expect IPW */
  free(ipw);
}
```

Using memory that you don't own

- Common error in 64-bit applications:
  - `int`s are 4 bytes but pointers are 8 bytes
  - If prototype of `malloc()` not provided, return value will be cast to a 4-byte `int`

Four bytes will be lopped off this value – resulting in an invalid pointer value

```
/*Forgot to #include <malloc.h>, <stdlib.h>
  in a 64-bit application*/
void illegalPointer() {
  int *pi = (int*) malloc(4 * sizeof(int));
  pi[0] = 10; /* Expect IPW */
  printf("Array value = %d\n", pi[0]); /* Expect IPR */
}
```

Using memory that you don't own

- Free memory read/write
  - Access of memory that has been freed earlier

```
int* init_array(int *ptr, int new_size) {
  ptr = (int*) realloc(ptr, new_size*sizeof(int));
  memset(ptr, 0, new_size*sizeof(int));
  return ptr;
}
int* fill_fibonacci(int *fib, int size) {
  int i;
  /* oops, forgot: fib = */ init_array(fib, size);
  /* fib[0] = 0; */ fib[1] = 1;
  for (i=2; i<size; i++)
   fib[i] = fib[i-1] + fib[i-2];
  return fib;
}
```

Remember: `realloc` may move entire block

What if array is moved to new location?

Using memory that you don't own

- Beyond stack read/write

```
char *append(const char* s1, const char *s2) {
  const int MAXSIZE = 128;
  char result[128];
  int i=0, j=0;
  for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
   result[i] = s1[j];
  }
  for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
   result[i] = s2[j];
  }
  result[++i] = '\0';
  return result;
}
```

`result` is a local array name – stack memory allocated

Function returns pointer to stack memory – won't be valid after function returns

Using memory that you haven’t allocated

- Array bound read/write

```
void genABRandABW() {
  const char *name = “Safety Critical";
  char *str = (char*) malloc(10);
  strncpy(str, name, 10);
  str[11] = '\0'; /* Expect ABW */
  printf("%s\n", str); /* Expect ABR */
}
```

Faulty heap management

- Memory leak

```
int *pi;
void foo() {
  pi = (int*) malloc(8*sizeof(int));
  /* Allocate memory for pi */
  /* Oops, leaked the old memory pointed to by pi */
  …
  free(pi); /* foo() is done with pi, so free it */
}
void main() {
  pi = (int*) malloc(4*sizeof(int));
  /* Expect MLK: foo leaks it */
  foo();
}
```

Faulty heap management

- Potential memory leak
  - no pointer to the beginning of a block
  - not necessarily critical – block beginning may still be reachable via pointer arithmetic

```
int *plk = NULL;
void genPLK() {
  plk = (int *) malloc(2 * sizeof(int));
   /* Expect PLK as pointer variable is incremented
      past beginning of block */
  plk++;
}
```

Faulty heap management

- Freeing non-heap memory
- Freeing unallocated memory

```
void genFNH() {
  int fnh = 0;
  free(&fnh); /* Expect FNH: freeing stack memory */
}
void genFUM() {
  int *fum = (int *) malloc(4 * sizeof(int));
  free(fum+1); /* Expect FUM: fum+1 points to middle
  of a block */
  free(fum);
  free(fum); /* Expect FUM: freeing already freed
  memory */
}
```