CS240 Programming in C

Dr. Gustavo Rodriguez-Rivera

Dr. George Adams

Spring 2017. Purdue University

EMERGENCY PREPAREDNESS – A MESSAGE FROM PURDUE

To report an emergency, call 911. To obtain updates regarding an ongoing emergency, sign up for Purdue Alert text messages, view www.purdue.edu/ea.

There are nearly 300 Emergency Telephones outdoors across campus and in parking garages that connect directly to the PUPD. If you feel threatened or need help, push the button and you will be connected immediately.

If we hear a fire alarm during class we will immediately suspend class, evacuate the building, and proceed outdoors. Do not use the elevator.

If we are notified during class of a Shelter in Place requirement for a tornado warning, we will suspend class and shelter in [the basement].

If we are notified during class of a Shelter in Place requirement for a hazardous materials release, or a civil disturbance, including a shooting or other use of weapons, we will suspend class and shelter in the classroom, shutting the door and turning off the lights.

Please review the Emergency Preparedness website for additional information.
http://www.purdue.edu/ehps/emergency_preparedness/index.html

General Information

- Web Page: http://www.cs.purdue.edu/homes/cs240
- Office: LWSN1210
- E-mail: grr@purdue.edu
  gba@purdue.edu
- Textbook:
  - "The C programming Language" (Second Edition) by Kernighan and Ritchie.

Mailing List

- All announcements will be sent via email
- Post your questions in Piazza.

Labs

- Lab 1 is already posted. It is due Monday.
- Lab attendance is mandatory.
- There is work in the lab that will be graded.
- Help Hours of GTAs and UTAs are posted in the web.

Lab Policy and Academic Dishonesty

- Do your project on your own.
- It is OK to discuss about your labs but at the time of coding do it on your own.
- We will check projects for academic dishonesty.
- Academic Dishonest will be penalized with failing the course and reporting to the Dean of Students.
- Every project has a GIT repository that records your progress every time you run tests.
- The GIT repository will be turned in together with your source code.

Attendance and Quizzes

- The attendance will be done with quizzes at the beginning of every lecture using the i-clicker.
- The Quiz will be about material covered in the previous class and the book.
- Bring your i-clicker to class.

Grading

- Grade allocation
  - Midterms and Final: 50%
  - Projects and Labs: 38%
  - Class Attendance and Quizzes 10%
  - Homework 2%
- Attendance will be done using quizzes and i-clicker.
- We will have live exams for midterms and final.

Course Contents

1. A Tutorial Introduction
2. Types Operators and Expressions
3. Control Flow
4. Functions and Program Structure
5. Pointers and Arrays
6. Structures
7. Input and Output
8. The UNIX System Interface
9. Introduction to C++
10. Classes and Methods

1. A Tutorial Introduction

The C Language

- C was created by Dennis Ritchie in 1972 in Bell Labs
- C was used to implement UNIX
- Operating Systems used to be implemented 100% in assembly language making them architecture dependent.
- C was designed to make UNIX portable:
  - 95% of Unix is in C
  - 5% is in assembly
- Only the assembly language part needs to be rewritten to migrate to other machine.
- Most of the optimizations you can do in assembly language you can do them in C.
- C is a “High-Level” assembly language.

Uses of C

- Linux is written in C
- Most of the libraries (low level) that need speed are written in C, strings, Windowing, Graphics, MP3 decoders, math libraries are also written in C
- Java runs on top of the JVM and the JVM is written in C
- Games that need high performance are written in C (or C++ in some cases or Objective-C)
- Java syntax was based on C&C++, therefore, you will find that many of the elements in C you knew them already
- C++ is a superset of C, so by learning C, you learn a big chunk of C++.
- C is used in Java native libraries that need speed. E.g. User Interface, Access to Database, Animation, Rendering, Math Library etc.

The C Principle

"C will not get in your way to make your program run fast."

- For example an array access such as :
  a[i]=val;

- In Java the code generated is equivalent to:
  // Check boundaries
  if (i >= 0 && i<max) { a[i]=val; } else throw out of boundary exception

- In C the code generated is equivalent to:
  // no boundary checks!!!!
  a[i]=val;
  Assignment will take place in memory even if i is out of range.
  Assignment will happen beyond the end of the array.

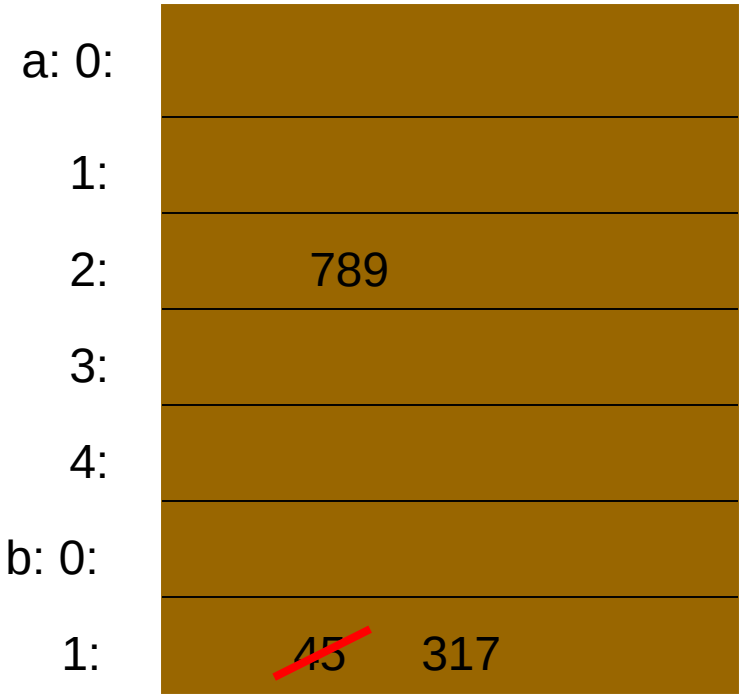
- An out-of-bounds is very difficult to debug since the assignment may happen in different variable located in memory just after the array.

- The same advantage of C that makes it fast makes it vulnerable to safety problems.

Out-of-bounds assignment in C

```
int a[5];
int b[3];
a[2]=789;
b[1]=45;
a[6]=317;
```

a and b in memory



b[1] will be overwritten !!!!!

C principle revisited

"C will not get in your way to make your program run fast."

…. However, C will not protect you if you make a mistake!!!

- You have to know what you are doing.
- Programming and debugging in C is more difficult and time consuming than programming in Java
- Java is used in applications that do not require too much CPU (I/O bound). Example: Web apps, calendar app
- C is used in applications that require a lot of CPU (CPU bound). Example: Games, MP3 Player.

Memory Usage in C and Java

- Java uses Garbage Collection.
  - The JVM will collect the objects that are unreachable by the program.
- C uses explicit memory management.
  - After calling p=malloc(size) to allocate memory, you will have to call free(p) when object p is no longer in use or it will continue using memory in the program.
  - This is called a memory leak.
  - If the program has a lot of memory leaked, the execution will slow down due to excessive memory usage or even crash.
  - If your program calls free(p) and you continue using the object and writing to the object pointed by p then the memory allocator data structures may get corrupted and your program will crash.
  - This is called a premature free.

- ***Memory allocation errors make programming in C difficult.***

Memory Usage in C and Java

- C programs typically use less that half the size of a Java program.
- C programs can be “Fast and Lean” but you have to be careful writing them.
- In general when having a new project, try to write it in Java, C#, Python etc.
- Only if speed is required use C/C++.

Example of a C Program

- Use a text editor to create the file and name it hello.c

```
#include <stdio.h>  //include file from
                    // /usr/include/stdio.h
    int main()
    {
        printf("Hello world\n");
    }
```

- To edit file use gedit, pico, vim, or xemacs.
- Pico and vim can be used in a text terminal using ssh from home.
- Gedit and xemacs need a windows system so they only can be used in the lab machines or running Linux at home.

Compiling a C program

- To compile a program

  gcc -o hello hello.c

  "hello" is the name of the executable.

- Also you may use

  gcc –g –o hello hello.c

  - Compile with debug information

  gcc -std=gnu99 –o hello hello.c

  - Compiles against the GNU C99 standard,

- To run it:

  bash% ./hello

  Hello World

Example: min/max

```
#include <stdio.h>

main() {
        printf("Program that prints max and min of two numbers
   a,b\n");
        int a, b;
        int max,min;

        while (1) {
                printf("Type a and <enter>: ");
                scanf("%d",&a);
                getchar(); // Discard new line

                printf("Type b and <enter>: ");
                scanf("%d",&b);
                getchar(); // Discard new line
```

Example: min/max (cont.)

```
            if (a > b) {
                    max = a;
                    min = b;
            }
            else {
                    max = b;
                    min = a;
            }
            printf("max=%d min=%d\n",max,min);

            printf("Do you want to continue? Type y/n and <enter>");
            char answer;
            answer = getchar();

            if (answer=='n') {
                    break;
            }
    }
    printf("Bye\n");
}
```

Example: min/max (cont.)

```
cs240@data ~/2014Fall/lab1/lab1-src $ gcc -o minmax  minmax.c
cs240@data ~/2014Fall/lab1/lab1-src $ ./minmax
Program that prints max and min of two numbers a,b
Type a and <enter>: 7
Type b and <enter>: 3
max=7 min=3
Do you want to continue? Type y/n and <enter>y
Type a and <enter>: 9
Type b and <enter>: 5
max=9 min=5
Do you want to continue? Type y/n and <enter>n
Bye
```

The For statement

- It is commonly used to iterate a known number of times.
- Syntax is similar to Java

```
// Print Fahrenheit - Celsius table
#include <stdio.h>
main()
{
  int fahr;
  for (fahr = 0; fahr <=300; fahr += 20) {
  printf("%3d %6.1f\n", fahr,
                (5.0/9.0)*(fahr-32));
  }
}
```

Symbolic constants

- #define symbol constant
  - Substitutes symbol with constant
- Example:

```
// Print Fahrenheit - Celsius table
#include <stdio.h>
#define LOWER 0
#define UPPER 300
#define STEP 20
main()
{
  int fahr;
  for (fahr = LOWER; fahr <=UPPER; fahr += STEP) {
   printf("%3d %6.1f\n", fahr,
                    (5.0/9.0)*(fahr-32));
  }
}
```

Simple Input/Output

- Standard input or stdin
  - File already opened when the program starts
  - Usually the keyboard
  - Can be redirected in the shell with “<“ like in “grep a < file”
- Standard output or stdout
  - File already opened when the program starts
  - Usually the screen
  - Can be redirected with “>“ like in “ls > file”
- putchar(c)
  - Outputs a character to the standard output
- int c = getchar(c)
  - Inputs a character from the keyboard or standard input

Count Characters in a file

```
#include <stdio.h>
main()
{
  int nc = 0;
  while (getchar() != EOF) {
    nc++;
  }
  printf("Number of characters: %d\n", nc);
}

gcc -g -o chcount chcount.c
./chcount
Hello
World
<ctrl-d>
Number of characters: 11
./chcount < myfile.txt
Number of characters: 37  (assuming myfile.txt has 37chars)
```

Count lines

- A line terminator character or “new line” is represented as ‘\n’
- Counting the number of lines is the same as counting the number of these characters in a file.

```
#include <stdio.h>
main()
{
  int nlines = 0;
  int c;
  while ((c=getchar()) != EOF) {
    if (c==‘\n’) nlines++;
  }
  printf(“Number of lines: %d\n”, nlines);
}

gcc –o countlines countlines.c
./countlines < countlines.c
Number of lines: 10
```

Word counting program

```
#include <stdio.h>

#define IN 1 /* Inside word */
#define OUT 2 /* Outside word */

/* count lines, words, and characters in input */
main()
{
  int c, nl, nw, nc, state;
  state = OUT;
  n1 = nw = nc = 0;
  while ((c= getchar()) != EOF) {
    nc++;
    if (c == '\n') {
      nl++;
    }
```

Word counting program (count.)

```
    if (c==' ' || c == '\n' || c=='\t') {
      state = OUT;
    }
    else if (state == OUT) {
      /* Character is not space and we were out of a word.
         Start inside a word */
      state = IN;
      nw++;
    }
 }
 printf("lines=%d words=%d chars=%d\n", nl, nw, nc);
}
```

Arrays. Count occurrences of each digit in a text

```
#include <stdio.h>
/* Count digits, whitespaces etc */
main()
{
  int c, i, nwhite, nother;
  int ndigit[10];
  nwhite = nother = 0;
  for (i=0; i < 10; i++) {
    ndigit[i] = 0;
  }
```

Arrays. Count occurrences of each digits in a text (cont.)

```
  while ((c=getchar()) != EOF) {
    if (c >= '0' && c <='9') {
      ndigit[c-'0']++;
    }
    else if ( c == ' ' || c =='\n' || c == 't') {
      nwhite++;
    }
    else {
      nother++;
    }
  }
  printf("Digits occurrence:\n");
  for (i=0; i < 10; i++) {
    printf("%d: %d\n", i, ndigit[i]);
  }
  printf("lines=%d chars=%d\n", nl, nc);
}
```

Example: Implementing "grep"

- ***grep*** is a UNIX command that is used to print the lines of a file that match a given pattern.

  grep pattern file

- Example:

```
lore 141 $ grep size index.html
<frame name="left" scrolling="no" noresize target="rtop" src="c.htm">
<frame name="rtop" scrolling="no" target="rbottom" src=".htm" noresize>
```

Mygrep implementation

```
/*
 * mygrep: Print the lines of a file that match a string
 *
 * mygrep pattern file
 */
#include <stdio.h>
#define MAXLINE 1023 //define a marco
char line[MAXLINE+1]; //global variable

void mygrep(char * fileName, char * pattern); //forward definition (prototype)

int main(int argc, char **argv)
{
    char * fileName;
    char * pattern;
    // Check that there are at least 2 arguments
    // mygrep file pattern
    // argv[0] argv[1] argv[2]
    // argc == 3
    if (argc<3) {
        printf("Usage: mygrep pattern file\n");
        exit(1);
    }
    pattern = argv[1];
    fileName = argv[2];
    mygrep(fileName, pattern);
    exit(0);
}
```

Mygrep implementation (cont.)

```
void mygrep(char * fileName, char * pattern) {
    FILE * fd = fopen(fileName,"r");
    if (fd == NULL) {
        printf("Cannot open file %s\n", fileName);
        exit(1);
    }
    while(fgets(line, MAXLINE, fd)!= NULL) {
        if (strstr(line,pattern)!=NULL) {
            printf("%s", line);
        }
    }
    fclose(fd);
}
```

2. Types, Operators an Expressions

Bits and Bytes

- Bits are grouped in bytes, where each byte is made of 8 bits.
- In modern computers a byte is the smallest unit that can be addressed by a CPU.
- A byte can be used to store values such as 00000000 (0 in decimal) to 11111111 (255 in decimal).
- These are very small numbers, so usually larger groups of bytes are used to represent other types of data .

Representation of Numbers in Memory

- Integers are represented in groups of
  - 1 byte (char)
  - 2 bytes (short int or short),
  - 4 bytes (int) ,
  - 8 bytes (long int or long)
  - and in some architectures 16 bytes (long long int or long long) variables.
  - Example of an 8 byte number in memory

```
00000000 00000000 00000000 00000000 00000000 10001001 00100100 10010010
63       55       47       39       31       23       15       7        0
```

$2^{23}+2^{19}+2^{16}+2^{13}+2^{10}+2^{7}+2^{4}+2^{1} = 8987794$

Representation of Negative Numbers in Memory

- Negative numbers typically use a representation called “complements of two”,
- A negative number is obtained by inverting the corresponding positive number and then adding 1.
- This representation allows using common positive integer arithmetic to do the addition and subtraction operations.
- For instance, the number represented above as a negative number can be obtained as:

```
Original: 00000000 00000000 00000000 00000000 00000000 10001001 00100100 10010010
Negated:  11111111 11111111 11111111 11111111 11111111 01110110 11011011 01101101
Plus 1    11111111 11111111 11111111 11111111 11111111 01110110 11011011 01101110
          63       55       47       39       31       23       15       7      0
Negative Original (Complements of 2):
          11111111 11111111 11111111 11111111 11111111 01110110 11011011 01101110
```

Original=8987794
Complements of 2 = -8987794

Complements of Two and Addition

- If we have the binary representation of 8987794 ***added*** to same number in complements of two representing -8987794 we will obtain 0 as expected.

```
 8987794: 00000000 00000000 00000000 00000000 00000000 10001001 00100100 10010010
+
-8987794: 11111111 11111111 11111111 11111111 11111111 01110110 11011011 01101110
 -------  -----------------------------------------------------------------------
       0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Representation of Strings

- Basic strings in C language are represented in memory as a sequence of bytes delimited by a 0 value.
- Each byte represents a character in ASCII representation.
- ASCII is the standard that translates characters in the English alphabet to numbers. ASCII stands for American Standard Code for Information Interchange.

ASCII Table

```
32:  48:0 64:@ 80:P 96:`  112:p
33:! 49:1 65:A 81:Q 97:a  113:q
34:" 50:2 66:B 82:R 98:b  114:r
35:# 51:3 67:C 83:S 99:c  115:s
36:$ 52:4 68:D 84:T 100:d 116:t
37:% 53:5 69:E 85:U 101:e 117:u
38:& 54:6 70:F 86:V 102:f 118:v
39:' 55:7 71:G 87:W 103:g 119:w
40:( 56:8 72:H 88:X 104:h 120:x
41:) 57:9 73:I 89:Y 105:i 121:y
42:* 58:: 74:J 90:Z 106:j 122:z
43:+ 59:; 75:K 91:[ 107:k 123:{
44:, 60:< 76:L 92:\ 108:l 124:|
45:- 61:= 77:M 93:] 109:m 125:}
46:. 62:> 78:N 94:^ 110:n 126:~
47:/ 63:? 79:O 95:_ 111:o 127:
```

C String Representation

- For example, the string “Hello world is represented by the equivalent ASCII characters delimited by a NULL character.

	H	e	l	l	o	\0
Bytes	72	101	108	108	111	0
Addr:	800	801	802	803	804	805

UNICODE

- To be able to represent characters in other languages, the Unicode standard was created.
- Unicode extends the ASCII standard and it uses two bytes to represent a character instead of one.
- In Unicode it is possible to represent the characters of mostly all languages in the world.

Data Types and Bytes

- Types such as integers, floats, doubles, or strings are represented as one or more of these bytes in memory.
- Everything stored in memory is represented with bytes.
- In C it is up to the program and the programmer to give meaning to what is stored in these bytes in memory.

What is GDB

- GDB is a debugger that helps you debug your program.
- The time you spend now learning gdb will save you days of debugging time.
- A debugger will make a good programmer a better programmer.

Compiling a program for gdb

- You need to compile with the "-g" option to be able to debug a program with gdb.
- The "-g" option adds debugging information to your program

```
gcc –g –o hello hello.c
```

Running a Program with gdb

- To run a program with gdb type

```
gdb progname
(gdb)
```

- Then set a breakpoint in the main function.

```
(gdb) break main
```

- A breakpoint is a marker in your program that will make the program stop and return control back to gdb.
- Now run your program.

```
(gdb) run
```

- If your program has arguments, you can pass them after run.

Stepping Through your Program

- Your program will start running and when it reaches "main()" it will stop.

```
gdb>
```

- Now you have the following commands to run your program step by step:

(gdb) step

It will run the next line of code and stop. If it is a function call, it will enter into it

(gdb) next

It will run the next line of code and stop. If it is a function call, it will not enter the function and it will go through it.

Example:

```
(gdb) step
(gdb) next
```

Setting breakpoints

- You can set breakpoints in a program in multiple ways:

  `(gdb) break function`

  Set a breakpoint in a function E.g.

  `(gdb) break main`

  `(gdb) break line`

  Set a break point at a line in the current file. E.g.

  `(gdb) break 66`

  It will set a break point in line 66 of the current file.

  `(gdb) break file:line`

  It will set a break point at a line in a specific file. E.g.

  `(gdb) break hello.c:78`

Regaining the Control

- When you type

  `(gdb) run`

  the program will start running and it will stop at a break point.

- If the program is running without stopping, you can regain control again typing ctrl-c.

Where is your Program

- The command

  (gdb)where

  Will print the current function being executed and the chain of functions that are calling that fuction.

  This is also called the backtrace.

  Example:

```
(gdb) where
#0  main () at test_mystring.c:22
(gdb)
```

Printing the Value of a Variable

- The command

  (gdb) print var

  Prints the value of a variable.

E.g.

```
(gdb) print i
$1 = 5
(gdb) print s1
$1 = 0x10740 "Hello"
(gdb) print stack[2]
$1 = 56
(gdb) print stack
$2 = {0, 0, 56, 0, 0, 0, 0, 0, 0, 0}
(gdb)
```

Exiting gdb

- The command "quit" exits gdb.

```
(gdb) quit
The program is running.  Exit
  anyway? (y or n) y
```

Debugging a Crashed Program

- This is also called “postmortem debugging”
- It has nothing to do with CSI ☺
- When a program crashes, it writes a **core file**.

```
bash-4.1$ ./hello
Segmentation Fault (core dumped)
bash-4.1$
```

- The core is a file that contains a snapshot of the program at the time of the crash. That includes what function the program was running.

Debugging a Crashed Program

- Sometimes the sysadmins disable the generation of core files to reduce the disk space waste. This happens in the CS machines.
- To find out if your system is able to generate cores type:

```
grr@data ~/cs240 $ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
```

- If you see that the core file size is 0. Enable core file generation by typing:

  grr@data ~/cs240 $ ulimit -c 1000000

Debugging a Crashed Program

- To run gdb in a crashed program type

  gdb program core

  E.g.

```
bash-4.1$ gdb hello core
GNU gdb 6.6
Program terminated with signal 11, Segmentation fault.
#0  0x000106cc in main () at hello.c:11
11          *s2 = 9;
(gdb)
```

  - Now you can type ***where*** to find out where the program crashed and the value of the variables at the time of the crash.

```
(gdb) where
#0  0x000106cc in main () at hello.c:11
(gdb) print s2
$1 = 0x0
(gdb)
```

  - This tells you why your program crashed. Isn't that great?

Now Try gdb in Your Own Program

- Make sure that your program is compiled with the –g option.
- Remember:
  - One hour you spend learning gdb will save you days of debugging.
  - Faster development, less stress, better results

String Functions in C

- C Provides string functions such as:
  - strcpy(char *dest, char *src)
    - Copy string src to dest.
  - int strlen(char * s)
    - Return the length of a string
  - char * strcat(char * dest, char *src)
    - Concatenates string src after string dest. Dest should point to a string large enough to have both dest and src and the null terminator.
  - int strcmp(char * a, char * b)
    - Compares to strings a,b. Returns >0 if a is larger alphabetically than b, < 0 if b is larger than a, or 0 if a and ba re equal.
  - See man string

String Functions in C

- char *strstr(const char *haystack, const char *needle);
  - Find the first occurrence of the substring needle in the string haystack, returning a pointer to the found substring.
- char *strdup(const char *s);
  - Return a duplicate of the string s in memory allocated using malloc.
- char *strncpy(char *dest, const char *src, size_t n);
  - Copy at most n bytes from string src to dest, returning a pointer to the start of dest.

Implementation of strlen

```
char * strlen(char * src) {
    int i = 0;
    while (src[i] != '\0') {
       i++;
    }
    return i;
}
```

Implementation of strcpy using array operator.

```
char * strcpy(char * dest, char * src) {
    int i = 0;
    while (src[i] != '\0') {
       dest[i] = src[i];
         // Copy character by char from src to dest.
       i++;
    }
   dest[i] ='\0'; // Copy null terminator
   return dest;
}
```

IMPORTANT: dest should point to a block of memory large enough to store the string pointed by dest.

Allocating an array using malloc

- When memory is no longer in use call free(p);
- If memory is not freed, the memory used by the program will increase and your program will start to slow down.
- Memory that is no longer in use and not freed is called a memory leak.
- Example:

```
int * array = new (N* sizeof(int));
// Use array
free(array);
```

Lexical Structure of C

A program is a sequence of characters in a text file.

```
hello.c
|*......*|
int main()
{
      printf("Hello World\n");
}
```

The Compiler groups characters into units called "tokens"(lexical units).

Comments in C

- Comments
- /*.....*/ same as Java
- //..... available in most compilers but it is not in standard.

Identifiers

- Sequence of letters, underscore and digits that do not start with a digit
- Only first 37 chars are significant
- Mix lower and upper case characters to make variable readable

Keywords

- Special keywords.

  while,break, for, case, break,continue, int.....

Types of variables

```
int a;              // Signed Integer 4 bytes
long b;             // Signed integer 8 bytes
short s;            // Signed integer 2 bytes
char c;             // Signed integer 1 byte
unsigned int d;     // Unsigned int 4 bytes
unsigned long e;    // Unsigned int 8 bytes
unsigned short f;   // Unsigned int 2 bytes
unsigned char g;    // Unsigned int 1 byte
```

Types of variables

```
float ff;          // Floating Point number 4 bytes
double dd;         // Floating Point number 8 bytes
long double ddd;   // Floating Point 16 bytes
```

char hello[20]; // A string is an array of chars. Last character is a 0. This string is able to store a string with 19 characters.

Constants

- Same as Java.

```
const double PI = 3.14192654;
```

- constants cannot be changed

- Also it is common in C to use the C pre-processor to define variables

```
#define PI 3.14192654
```

Assignment

- The element in the left side has to be an “l-value”. An “l-value” can be interpreted as an address.
- A variable is an l-value but a constant is not an l-value.

  x = 5; ← l-value

  8 = 5; ← not l-value

- In C, an assignment is an expression
- This implies that an assignment may appear anywhere an expression is allowed.

  Example:

```
  j = (i = 5) +3; // this will assign 5 to i and it will assign 8 to j.
```

- Example:

```
  while( (c = getChar() ) != -1){ …
  }
```

The Main Program

- Execution starts in main

```
int main(int argc, char**argv){
}
```

or

```
int main(){
}
```

- argc-store # of args.
- argv-it is an array of the argument entries as strings.

```
int main(int argc, char**argv){
    int i;
    for(i = 0; i <argc; i++){
        printf("argv[%d] = %s\n, i, argv[i]);
    }
}
```

Integer Constants

- Integer Constants

  1 2 3 4 → decimal

  031 → octal constant 3*8+1 = 25 Starts with 0

  0x4A3 → hexadecimal constant 4*16^2+10*16+3=
  or in binary 0100 1010 0011

- The type of integer constant will be:
  - int → if v does not exceed the int range.
  - long → if v exceeds an int but not a long.
  - unsigned long → if v exceeds long

Integer Constants

- You can add suffix to force type
  - 123456789L → long
  - 55u → unsigned int
  - 234Lu → unsigned long

Floating Point Constants

- 3.14 → type is always double
- To force the type float, add “f” suffix.
- 3.14f → float constant

Character constant

- 'q' enclosed with a single quote.
- Also you can use escape sequences with '\octal number' e.g. '\020’

  ascii: 2*8+0=16
- '\hex number with two digits' e.g. '\AE’

  ascii: 10*16+14

Character Constants

- Also there are some common escape sequences '\n' → new line
  - '\r' → return
  - '\t' → tab
  - '\' ' → single quote
  - '\" ' → double quote
  - '\\' → back slash

Character Constants

- Character constants have type int.

```
int i;
i = 'A'; //assign ascii 65 to I
Or
i = 65;
printf("A = %d\n", 'A');
```

output
A=65

- Example:

```
//check if a letter is a lowercase
   if(c >= 'a' && c <= 'z'){
        printf("%c char %d is lower case \n", c, c);
   }
```

String Constants

- “My String” is a string constant of type (char *)
- There are no operations with string in Java like “Hello”+”world”.
- However, two consecutive string constants can be put together by the complier

  `“Hello. ” “world”` is equivalent to

  `“Hello. world”`
- So you can have multi-line strings like,

  char * class =

  “CS240 \n”

  “Programming in C \n”;
- The compiler will put both constants in a single string.

Short-Circuit && (and) and || or expression

- ***e1 && e2*** is the short circuit "and" expression
  - If e1 is false, e2 is not evaluated.

```
if(x && (i = y))
//  If x is false
// then i=y is never evaluated.
```

- ***e1 || e2*** is the short circuit "or" expression
  - If e1 is true, then e2 is never evaluated

```
if (x || (i=y))
  // If x is true, then i=y
  // is never evaluated.
```

Boolean and Int

- There is no Boolean type
- A 0 is False and anything different than 0 is True.
- ```
  if(5){
        //always executed
  }
  if(0){
        //never executed
  }
  ```
  ```

Conditional Expressions

- e = (e1 ? e2 : e3) *conditional expression

Equivalent to

```
if(e1){
    e = e2;
}
else{
    e = e3;
}
```

- Example:

```
x = 3 + ( ( b == 4 ) ? 7 : 8 )
```

Equivalent to:

```
if(b==4){
    x = 3+7;
}
else{
    x=3+8;
}
```

Comma Expressions

- i = (e1, e2) *comma expressions
  - the value of i is the last expression e2.
- i=(e1, e2, e3)
  - the value of i is the last expression e3
- Useful for the “for” statement to execute multiple increments or assignments.
- Example:

```
for( i=0, j=3; i<7; i++, j—) {…}
```

Arithmetic Conversion

- If operands in an expression have different types the operands will have their types changed from the lower precision type to the higher precision type
- → 5 / 2 = 2 types is an int
  (int) (int) division of 2 ints
- → 5 / 2.0 = the compiler evaluates 5 and convert it to 5.0(double) = 2.5 type is double
  (int) (double) division between int and double

- (1 / 2) * (3.0 + 2) = 0

use (1.0 / 2) * (3.0 + 2) instead so the result is 2.5 double

- C does not perform arithmetic at precision smaller than int
  精确度小于int的会直接当成int处理，因为小于int精确度的不能直接算数

Arithmetic Conversion

- *int i;*
  i = 'A' + 'B'; → 131 (int)
- Even though 'A' and 'B' is char it is converted to int (65 + 66)

- - A lower precision type converted to a higher precision type

- int → unsigned → long → unsigned → float → double → long double

- *int i = 10;*
  unsigned u = 20; 大于0的整数
- *i + u* -> i is converted to unsigned since unsigned has more precision to int

Assigment conversion

- They happen when an expression in the right hand side has to be converted to the type in the left hand side in an assignment.

```
int i = 3;
double d;
d = i;  // i is converted to double
```

低精确度转高精确度没事

- You have to be careful if you are assigning to a variable with a smaller precision

```
d = 2.5;
i = d;  // 2.5 converted from double to int 2
```

- You will get a warning and some cases an error. Use a cast instead

```
i = (int)d;
```

高精确度转低精确度要cast

Cast Conversion

- Done by the programmer

  (type) expression

  (int) 2.5 results into 2

typedef

- typedef provides a synonym of an existing type

```
        typedef int Boolean;
        Boolean b;
#define FALSE 0
#define TRUE  1
        b = 1;
        b = FALSE;
        b = TRUE;
```

Common Errors

- “a” and ‘a’ are different
  - “a” is a string constant type (char *)
  - ‘a’ is a char constant type (int)

- `if (c = getchar() != EOF)`

  is not the same as

  `if ((c = getchar()) != EOF)`

Bit Operations: Use unsigned numbers when doing bit operation

Left and Right Shift << >>

- x >> i
  - Shifts bits of a number x to the right i positions
- x << i
  - Shifts bits of a number x to the left i positions
- Example:

```
int i, j;
i = 5; // In binary i is 00000101
j= (5 << 3); // In binary j is 00101000
printf("i=%d j=%d\n"); // Output: i=5 j=40
```

- **x<<i is equivalent to x*2^i**
- **x>>i is equivalent to x/2^i**

因为负数是most significant number为1的，shift可能会导致直接变成负数

Bitwise Operations: OR |

- The "|" operator executes "OR" bit operation.

```
unsigned x = 0x05; // 00000101
unsigned y = (x | 0x2);
          // 00000101 | 00000010=00000111
printf("x=0x%x 0x%x\n", x,y); // x=0x5 y=0x7
```

anything with 1 is 1

```
Example:
/usr/include/unistd.h:
#define O_READ 0x01
#define O_WRITE 0x02
In your code
d = open("/path/to/dir", O_READ| O_WRITE);
// 0x01 | 0x02 == 0x03
```

Bitwise Operations: AND &

anything with 0 is 0

- The “&” operator executes “AND” bit operation.

```
unsigned x = 0x05; // 00000101
unsigned y =(x & 0x3);// 00000101 & 00000011 =00000001
printf(“x=0x%x y=0x%x\n”, x,y); // x=0x5 y=0x1
```

Bitwise Operations: XOR ^

- The "^" operator executes "XOR" bit operation.
- XOR : 0^0==0, 0 ^1 == 1, 1^0==1, 1^1==0

餐厅菜单，A或者B，只能选一个

```
unsigned x = 0x05; // 00000101
unsigned y =(x ^ 0x3);// 00000101 ^ 00000011 =00000110
printf("x=0x%x 0x%x\n", x,y); // x=0x5 y=0x6

Example:
// Inverting the bits of a number
unsigned x = 0x05; // 00000000 00000000 00000000 00000101
unsigned y = x ^ 0xFFFFFFFF;
//0xFFFFFFFF=11111111 11111111 1111111 11111111
      // y = 11111111 1111111 1111111 11111010
```

Bitwise Operations: NOT ~

- The "~" negates bits.

```
unsigned x = 0x05; // 0000000 0000000 0000000 00000101
unsigned y = ~x;
    // ~00000101 = 11111111 11111111 11111111 11111010
printf("x=0x%x 0x%x\n", x,y); // x=0x5 y=0xFFFFFFFA
```

Using Bitwise Operations: Test if bit i is set:

```
int i = 4;
unsigned x = 23; // x = 00010111

// Test if bit i is set in x
// Create mask with bit i set.
unsigned mask = (1 << i); // mask == 00010000

// Test if bit i is set
unsigned y = (x & mask); // y = 00010111 & 00010000 =00010000
int bit = (y >> i);      // bit = 00000001
                         // bit i in x is set.
```

Using Bitwise Operations: Set bit i :

```
int i = 3;
unsigned x = 23; // x = 00010111

// Set bit i in x
// Create mask with bit i set.
unsigned mask = (1 << i); // mask == 0001000

// Set bit i
unsigned y = (x | mask); // y = 00010111 | 0001000 = 00011111
```

Using Bitwise Operations: Clear bit i :

```
int i = 2;
unsigned x = 23; // x = 00010111

// Set bit i in x
// Create mask with bit i set.
unsigned mask = (1 << i); // mask == 00000100
unsigned mask0 = ~mask;  // mask0 == 11111011

// Clear bit i
unsigned y = (x & mask0); // y = 00010111 & 11111011 = 00010011
```

Right Shift >> with sign extension.

- When using i >> n (shift right) the behavior will change if i is a signed int or i is unsigned int.
- If i is signed int and i is negative then i >> n will add *n* 1s in the left side.
- If i is unsigned then i >> n will add *n* 0s in the right side.
- This is called “signed extension”.

Example:

```
int i = 0xFFFFFFFB; // i=11111111 11111111 11111111 11111011
int j = (i >> 2);  // j=11111111 11111111 11111111 11111110 =0xFFFFFFFFFE
```

and

```
unsigned i = 0xFFFFFFFB; // i=11111111 11111111 11111111 11111011
unsigned j = (i >> 2); // j=00111111 11111111 11111111 11111110=3FFFFFFE
```

3. Control Flow

if Statement

- if(..expression..){
  .....
  .....
  }
- Expression in if (exp) statement is converted to int.
- If (expression) != 0 then expression is true and the body of the statement is executed.
- if(expression) == 0 then it is false and the body is not executed. Then continues to check against else if or else statements are used to then be executed.
- Example:

```
int i;
// i get some value
if(i != 0){
      // do something
}
```

- equivalent to:

```
if(i){
      // do something
}
```

while statement

- while(..expression..){
  .....
  .....
  }
- Expression in while statement is converted to an int.

- while(expression) != 0 then the program continuous to loop until expression does equal 0.
- Example:

```
int i, j;
while(i > j){
    ....          // code body that will continue to executed
                  // until j >= i
}
```

for statement

- "for" statement is typically used in situation where you know in advance the number of iterations.
- Syntax:

```
for(expression1; expression2; expression3){
     ....
}
```

- Example:

```
//Assuming the variable i has been declared above.
for(i = 0; i < 10; i++){
  ....... // code to be executed goes here
          // this specific for statement will
          // loop until !(i < 10) for a total
          // of 10 iterations
}
```

for statement

- However you could use the for statement where a while statement is also used.

  Example:

```
expr1;
while(expr2){
    // body of statement to be executed
    expr3;
}
```

  expr1 and expr3 are usually 'coma expressions'

```
        expr1        expr2     expr3
for(  i = 0, j = 0;   i < 10;     i++    ){
   // body of statement to be executed
}
```

switch statement

- switch(expr) {
  case const1: .... break;
  case const2: .... break;
  default: .... break;
  }
- expr is evaluated and converted to an int.
- If ....expr.... is equal to any of const values that block of code is executed.
- Default is evaluated if expr does not match any of the case consts.

Forever Loops (Infinite Loops)

- ```
  while(1){
      // body runs forever
  }
  ```

- ```
  for(;;){
      // body runs forever
  }
  ```

Example: Count the number of lines, tabs and lower case characters in the input

```
#include <stdio.h>

int main(){
   int countBlanks = 0;
   int countTabs = 0;
   int countNewline = 0;
   int countLower = 0;
   int c;
```

Example: Count the number of lines, tabs and lower case characters in the input

```
while((c = getchar()) != EOF){
     switch(c){
             case ' ':
                     countBlank++;
                     break;
             case '\t':
                     countTabs++;
                     break;
             case '\n':
                     countNewline++;
                     break;
             default:
                     if(c >= 'a' && c <= 'z'){
                             countLower++;
                     }
                     break;
     }       //End of switch statement
} //End of while loop
```

Example: Count the number of lines, tabs and lower case characters in the input

```
    //Within a switch statement when a "break"
    //is hit it exits the switch statement.
   //If there is no break statement within a
  // case the program would then continue to
  // execute each case until a break statement is hit.

  printf("blanks=%d tabs=%d newlines=%d lower= %d\n",
     countBlanks, countTabs, countNewLines, countLower);
}  //End of main
```

Midterm 1

- Do the example exam. Turn it in before exam.
- Study:
  - slides,
  - example programs in the slides (important)
  - Labs:
    - String Functions
    - rpn-calculator
  - Other
    - Strings as arrays, convert upper to lower

Midterm

- 4 Problems in 4 dirs
  - Problem1/
  - Problem2/
  - Problem3/
  - Problem4
- There is a README file with instructions
- Modify the file that starts as problem*.c
- Run testall
- Use gdb

4. Functions and Program Structure

Structure of a C program

- A C program is a sequence of definitions of functions, and global variables, type definitions and function prototypes:

```
global var1
global var2
func1
func2
func3
```

- Example:

```
int s; // Global variable. Lifetime is the entire duration of the program.
int sum(int a, int b) {
  int x; // Local variable. Lifetime is only when this funciton is invoked.
  x = a + b;
  return x;
}
int main() {
{
    int y;
    y = sum(3,4)
   printf("y=%d\n", y);
}
```

- C does not have classes interfaces etc.

Compiling from multiple files

- Compiling object files:
- You may compile each file separately and then link them:

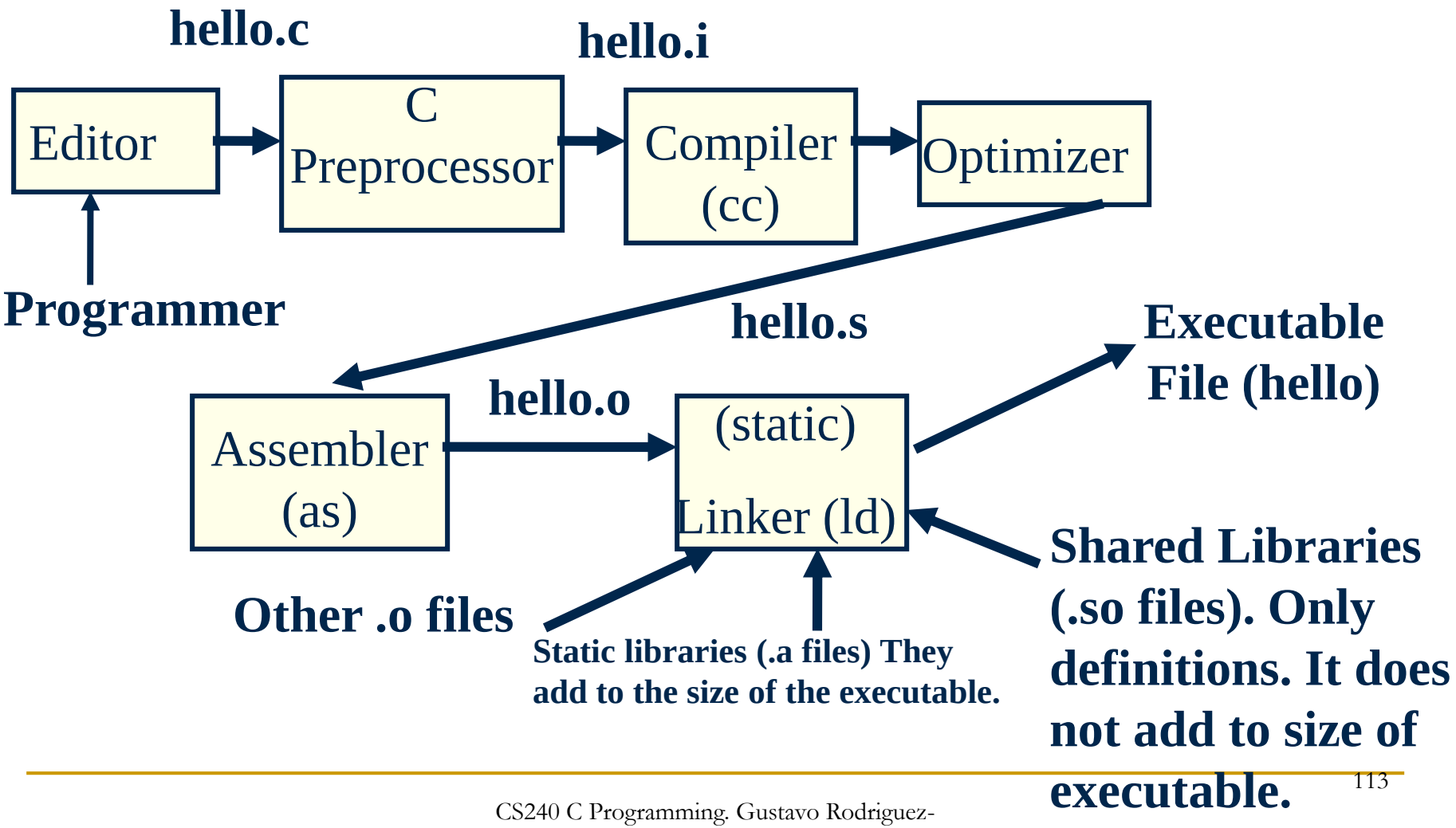
```
gcc –c sum.c          - Generates sum.o
gcc –c main.c         - Generates main.o
gcc –o main sum.o main.o  - Generates main
```

- Alternatively you may generates main in a single step.

```
gcc –o main main.c sum.c
```

- The first option is preferred if many .o files are used and only a few are modified at a time.

Building a Program



Building a Program

- The programmer writes a program hello.c
- The ***preprocessor*** expands #define, #include, #ifdef etc preprocessor statements and generates a hello.i file.
- The ***compiler*** compiles hello.i, optimizes it and generates an assembly instruction listing hello.s
- The ***assembler*** (as) assembles hello.s and generates an object file hello.o
- The compiler (cc or gcc) by default hides all these intermediate steps. You can use compiler options to run each step independently.

Building a program

- The linker puts together all object files as well as the object files in static libraries.
- The linker also takes the definitions in shared libraries and verifies that the symbols (functions and variables) needed by the program are completely satisfied.
- If there is symbol that is not defined in either the executable or shared libraries, the linker will give an error.
- Static libraries (.a files) are added to the executable. shared libraries (.so files) are not added to the executable file.

Original file hello.c

```
#include <stdio.h>
main()
{
   printf("Hello\n");
}
```

After preprocessor

```
gcc -E hello.c > hello.i
    (-E stops compiler after running preprocessor)
 hello.i:
    /* Expanded /usr/include/stdio.h */
     typedef void *__va_list;
     typedef struct __FILE  __FILE;
     typedef int     ssize_t;
     struct FILE {…};
     extern int fprintf(FILE *, const char *, ...);
     extern int fscanf(FILE *, const char *, ...);
     extern int printf(const char *, ...);
     /* and more */
     main()
     {
        printf("Hello\n");
     }
```

After assembler

gcc -S hello.c (-S stops compiler after assembling)

hello.s:

```
        .align 8
.LLC0:  .asciz  "Hello\n"
.section        ".text"
        .align 4
        .global main
        .type    main,#function
        .proc   04
main:   save    %sp, -112, %sp
        sethi   %hi(.LLC0), %o1
        or      %o1, %lo(.LLC0), %o0
        call    printf, 0
        nop
.LL2:   ret
        restore
```

After compiling

- "gcc -c hello.c" generates hello.o
- hello.o has undefined symbols, like the ***printf*** function call that we don't know where it is placed.
- The main function already has a value relative to the object file hello.o

csh> nm -xv hello.o

```
hello.o:
[Index]   Value      Size       Type  Bind  Other Shndx   Name
[1]     |0x00000000|0x00000000|FILE |LOCL |0    |ABS    |hello.c
[2]     |0x00000000|0x00000000|NOTY |LOCL |0    |2      |gcc2_compiled
[3]     |0x00000000|0x00000000|SECT |LOCL |0    |2      |
[4]     |0x00000000|0x00000000|SECT |LOCL |0    |3      |
[5]     |0x00000000|0x00000000|NOTY |GLOB |0    |UNDEF  |printf
[6]     |0x00000000|0x0000001c|FUNC |GLOB |0    |2      |main
```

After linking

- "**gcc –o hello hello.c**" generates the hello executable
- Printf does not have a value yet until the program is loaded

```
csh> nm hello
[Index]   Value      Size       Type  Bind  Other Shndx   Name
[29]    |0x00010000|0x00000000|OBJT |LOCL |0     |1      |_START_
[65]    |0x0001042c|0x00000074|FUNC |GLOB |0     |9      |_start
[43]    |0x00010564|0x00000000|FUNC |LOCL |0     |9      |fini_dummy
[60]    |0x000105c4|0x0000001c|FUNC |GLOB |0     |9      |main
[71]    |0x000206d8|0x00000000|FUNC |GLOB |0     |UNDEF  |atexit
[72]    |0x000206f0|0x00000000|FUNC |GLOB |0     |UNDEF  |_exit
[67]    |0x00020714|0x00000000|FUNC |GLOB |0     |UNDEF  |printf
```

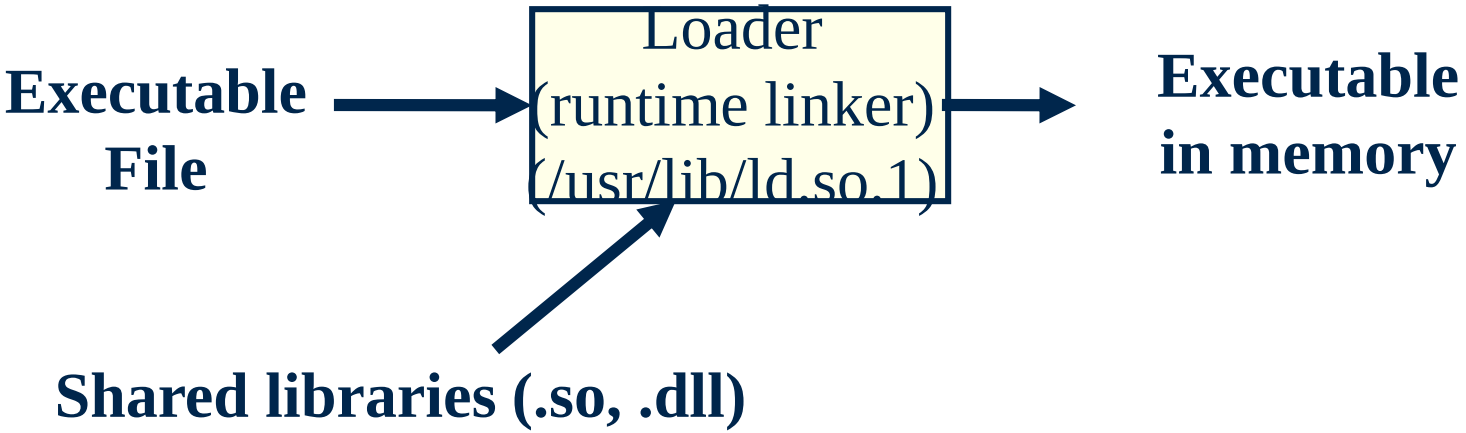
Loading a Program

- The loader is a program that is used to run an executable file in a process.
- Before the program starts running, the loader allocates space for all the sections of the executable file (text, data, bss etc)
- It loads into memory the executable and shared libraries (if not loaded yet)

Loading a Program

- It also writes (resolves) any values in the executable to point to the functions/variables in the shared libraries.(E.g. calls to printf in hello.c)
- Once memory image is ready, the loader jumps to the ***_start*** entry point that calls init() of all libraries and initializes static constructors. Then it calls ***main()*** and the program begins.
- ***_start*** also calls ***exit()*** when ***main()*** returns.
- The loader is also called “runtime linker”.

Loading a Program



Static and Shared Libraries

- Shared libraries are shared across different processes.
- There is only one instance of each shared library for the entire system.
- Static libraries are not shared.
- There is an instance of an static library for each process.

The C Preprocessor: Macro Definitions #define

- Macro Definitions are used to define constants or functions that need to be in-lined

```
#define PI 3.14
#define printHi printf("Hi")
```

- macros can have parameters

```
//returns true if c is lowercase…
#define islower(c) (c>='a'&&c<='z')

if(islower(x)){
    //Do something
}
```

Macro Expansion

- Look at the following macro and expansion:

```
//returns true if c is lowercase…
#define islower(c) ( c >= 'a'&& c<= 'z')

 if(islower(x+1)){
     //Do something
 }
```

- After the C preprocessor it becomes:

```
if( (x+1 >= 'a'&& x+1 <= 'z') ){
    //Do something
}
```

Be careful with macros

- Be careful sometimes with macros
- For example:

```
#define mysqr(x) x*x
y = mysqr(3+z)
```

- Becomes after the C preprocessor:

```
y = 3+z*3+z
```

that is not what we want

Always Put Parenthesis Around Arguments in Macros.

- To fix this problem always put parenthesis around arguments.

```
#define mysqr(x) ((x)*(x))
y = mysqr(3+z)
```

- Becomes

```
y = ((3+z)*(3+z))
```

that is what we want

The C Preprocessor: Macro Definitions #define

- Another example of a macro

```
// get one character from a file
#define getchar() fgetc(stdin)
```

- A macro can be used from where it is defined to the end to the file

- Un-defining macros

  #undef PI

The C Preprocessor: File Inclusion #include

- Examples of file inclusion

```
#include “FILENAME”       // It will search for the file in the current directory----relative path
#include “/home/grr/a.h”   // Absolute path
#include <FILENAME>       // look for include file in the system’s directories----/usr/include
```

- Example of an include file

mydefs.h:

```
#define PI 3.14
#define MAX_STUDENTS 14
```

myProgram.c:

```
#include “mydefs.h”

int main(){
        printf(“pi = %lf\n”, PI);
}
```

- We can even include a header file in another header file!

The C Preprocessor: Conditional Compilation #if

Examples of conditional compilation

```
#if constantexpr1   //evaluated by preprocessor, before compilation
… Included if constatnexpr is not 0
#end

#if constantexpr2
……
#else
……
#end
```

The C Preprocessor: Conditional Compilation #if

- Conditional Compilation is useful if we have various environment, say Solaris, Linux, Windows……

- Example

```
#define MACHINEARCH 64
     //Notice that the 64 here is subject to change from
    // one architecture to another
#if MACHINEARCH == 64
      //Code for 64 bits……
#elif MACHINEARCH ==32
     //Code for 32  bits
#else
      //Unknown
#end
```

Or at compilation you can pass:

```
gcc –DMACHINEARCH=64 –o myfile myfile.c
```

The C Preprocessor: Conditional Compilation #if

- You may use conditional compilatiion to comment multiple lines of code that have comments already
- Example:

```
#if 0
   // We cannot use /**/ here to comment comments
   /*Hello*/
   int main(){
         /*Another comment*/
   }
#endif
```

The C Preprocessor: Conditional Compilation #ifdef and #ifndef

- #ifndef is often used in include files to prevent include files from being included multiple times.

stdio.h:

```
#ifndef STDIO_H
#define STDIO_H
                    //This code is included only once…
#endif
```

---

hello.h

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

---

hello.c

```
#include <stdio.h>   //include stdio.h above
#include "hello.h"   // hello.h will not include stdio.h again.

main()
```

Some Predefined Macros

Some predefined macros:

_ _LINE_ _ Expands to the line number

_ _FILE_ _ Expands to the file name

_ _TIME_ _ Expands to the time of translation (compilation)

Assertions

- EXAMPLE

```
#define MyAssert(x) \
    if(!(x)){ \
     printf("Assertion failed!%s:%d\n", \
            _ _FILE_ _, _ _LINE_ _); \
          }\
     exit(1)

main{
        …
            MyAssert(i>=0 && i<MAX);
            a[i] = 5;
```

Assertions

- Assertions like the one above are useful for “defensive” programming.
- It is better to have an assertion failure than a segmentation fault.
- Assertions are already defined in

```
#include <assert.h>
main{
 …
        assert(i>=0 && i<max) ;
…
}
```

- If the expression in the assertion is false, it will print the expression, file name, and line number

Functions in Multiple Files

- A function defined in one file can be used in another file.

sum.c:

```
extern int total; // extern declaration
void sum( int a, int b) {
   total = a + b;
}
```

- The second file needs to have an extern definition:

header file就是把func extern了

main.c:

```
#include <stdio.h>
int total; //definiton
extern void sum(int a, int b);
main() {
  printf("sum=%d\n", sum(5,8));
}
```

```
gcc -o main main.c sum.c

gcc -c main.c     main.c
gcc -c sum.c      sum.o
gcc -o main main.o sum.o
```

Making functions private

- You can make a function private to a file by adding the *static* keyword before the function declaration.

```
static void mylocalfunction(int x) {
  ….
}
// The mylocalfunction() function can only be used in
   this file.
```

- In this way no other C file can see this function.

Scope and Lifetime of a variable

- Scope = The *place* in the program where a variable can be used.
- Lifetime = The *time* in the execution when a variable is valid.

Global Variables

- Global Variables are defined outside a function.
- *The Scope* of a Global Variable is from where it is defined to the end of the file.
- *The Lifetime* of a Global Variable is the whole duration of the program.
- If the C program spans more than two files a global variable can be used in both files
- One file has the *definition*:

```
int total;
```

- The other file may have an “extern” declaration to tell that the variable is defined in another file.

```
extern int total;
```

- The extern declaration in this case is optional but recommended.
- Global Variables are intialized with 0s.

Global Variables in Multiple Files

sum.c:

```
extern int total; // extern declaration
void sum( int a, int b) {
   total = a + b;
}
```

main.c:

```
#include <stdio.h>
int total; //definition
extern sum(int a, int b);
main() {
  sum(5,8)
  printf("sum=%d\n", total);
}
```

Local Variables

- Local variables are defined inside functions
- *The Scope* of a Local Variable is from where it is defined to the end of the function.
- *The Lifetime* of a Local Variable is from the time the function starts to the time the function returns.
- Local Variables are stored in the execution stack.
- Local variables are not initialized. The initial variable will be whatever value is in the stack when the function starts.

Static Local Vars

```
static-globally known in this file, other files cant use it
extern-globally known in all file, other files can use it
```

- If you add the keyword *static* before a local variable it will make the value of the variable be preserved across function invocations.
- The variable will be stored in data/bss instead of the stack.
- The Scope of a static local var is the function it is defined.
- The Lifetime of a static local var is the whole execution of the program.

Example static local var

```
int sum(int a) {
  static int i;
  i += a;
  printf("i=%d\n");
}

main() {
  sum(4);
  sum(5);
  sum(6);
}
```

static variable will go through the whole execution of the program, so it wont get initialized again

Output:

```
  4
  9
  15
```

5. Pointers and Arrays

Memory of a Program

- From the point of view of a program, the memory in the computer is an array of bytes
- This array goes from address 0 to address 2^{64}-1 (0x0 to 0xFFFFFFFFFFFFFFFF) assuming a 64-bit architecture.

Computer Memory as an Array of Bytes

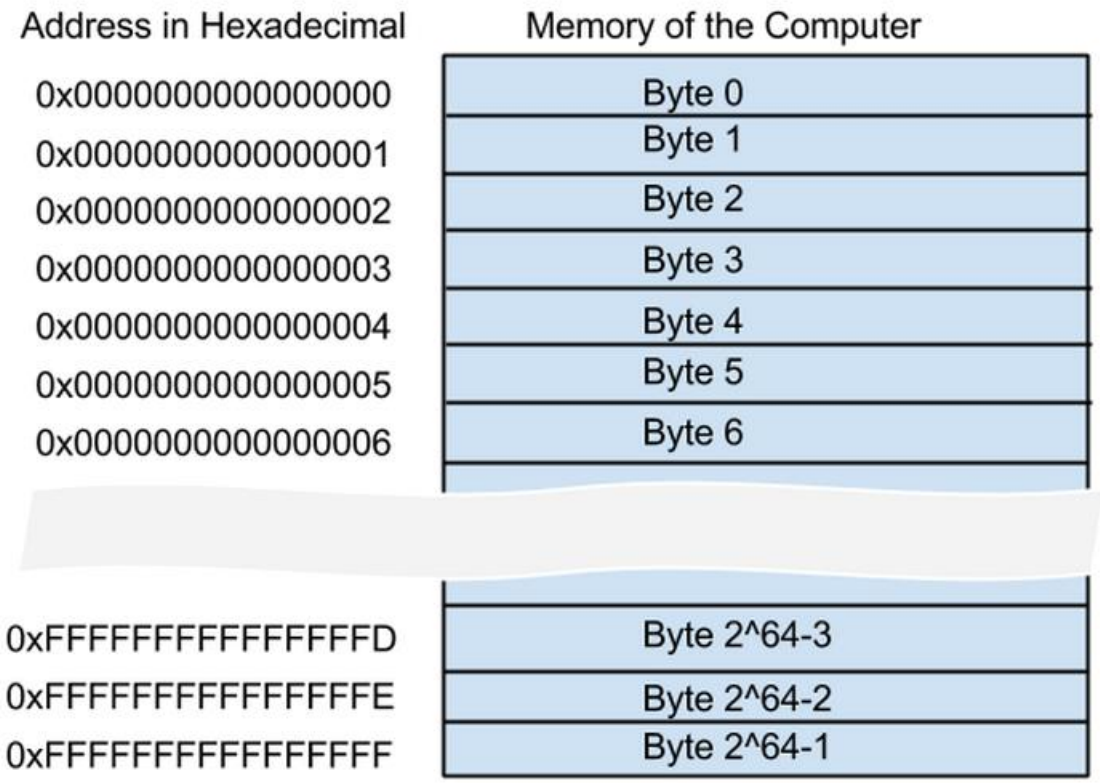


Figure 1.1: Computer Memory as an Array of Bytes (char[])

Memory Gaps

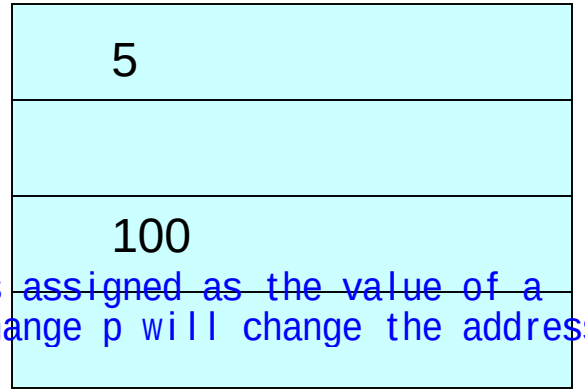
- Every program that runs in memory will see the memory this way.
- In C/C++ or assembly language it is possible to access the location of any of these bytes using pointers and pointer dereferencing.
- Theoretically a program may access any of these locations.
- However, there are gaps in the address space.
- Not all the addresses are “mapped” to physical memory.
- When accessing memory in these gaps, the program will get an exception called Segmentation Violation or SEGV and the program will crash.

Pointers

A pointer is an address in memory.

```
int a;
int *p;
a=5;
p=&a;
```

p is assigned to the address of a
simultaneously, *p is assigned to *a
so p print the address of a
&p print the address of p
*p print the value of p, in this case is assigned as the value of a
change *p will change the value of a, change p will change the address of a




```
printf("a= %d",a);   // prints a=5
printf("&a = %ld",&a); // prints &a=100
printf("p=%ld",p);  //prints p=100
printf("&p=%ld",&p);  //prints &p=200
printf("*p = %d",*p);   //prints *p=5
```

Printing Pointers

- * is “value at” operator.
- & is “address of” operator.
- To print pointers we often use the “0x%lx” instead of %ld format since %ld may print pointers as negative numbers
- If you want to print pointers as unsigned decimals, use “%lud” or just “%lu” (will print positive number)

Pointer Types

- Pointers have types :

```
int i;
int * p;   // p is an integer pointer
unsigned char  *q; // q is a pointer to unsigned char
double * pd;  // pd is a pointer to a double
```

- Sometimes it is necessary to convert one pointer type to another:

```
q = (unsigned char *) p;
```

- This allows to store in or read from the same memory but as a different type.

Little Endian /Big Endian

- Assume the following statements:

```
int i = 5;
unsigned char * p;
p = (unsigned char*) &i;
```

- 5 may be stored in two ways:

Little Endian:

Least Significant Digit in Smallest Memory Location

i: 100	5
101:	0
102:	0
103:	0

Big Endian:

Least Significant Digit in Highest Memory Location

i: 100	0
101:	0
102:	0
103:	5

Code to Determine if machine is Little Endian

```
Int isLittleEndian()
{
  int i = 5;
  unsigned char * p = (unsigned char *) &i;

  if (p[0] == 5) {
    return 1;
  }
  else {
    return 0;
  }
}
```

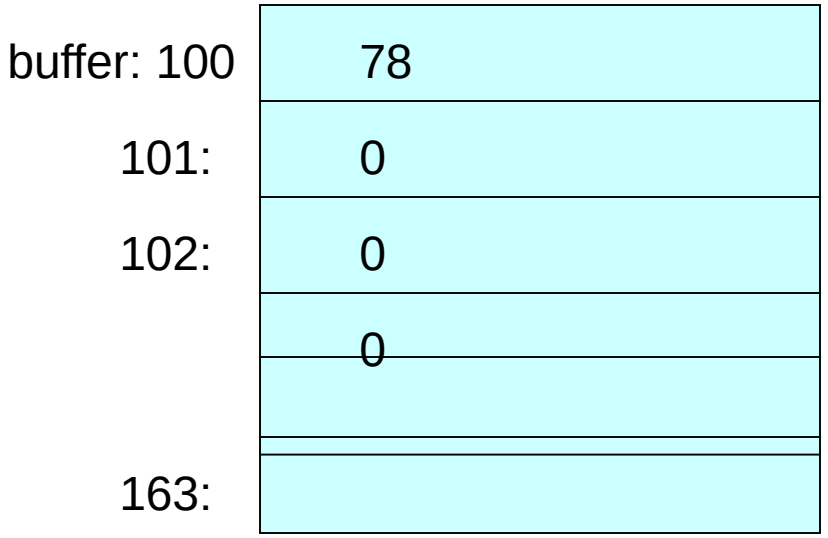
Little Endian /Big Endian

- **Little Endian** : Puts 5 at 100 (Intel, ARM, VAX)
  - Lowest significant byte is placed in the lowest address
- **Big Endian** : Puts 5 at 103 (Sparc, Motorola)
  - Lowest significant byte in the highest address

Pointer Conversion

- Pointer conversion is very powerful because you can read or write values of any type anywhere in memory.

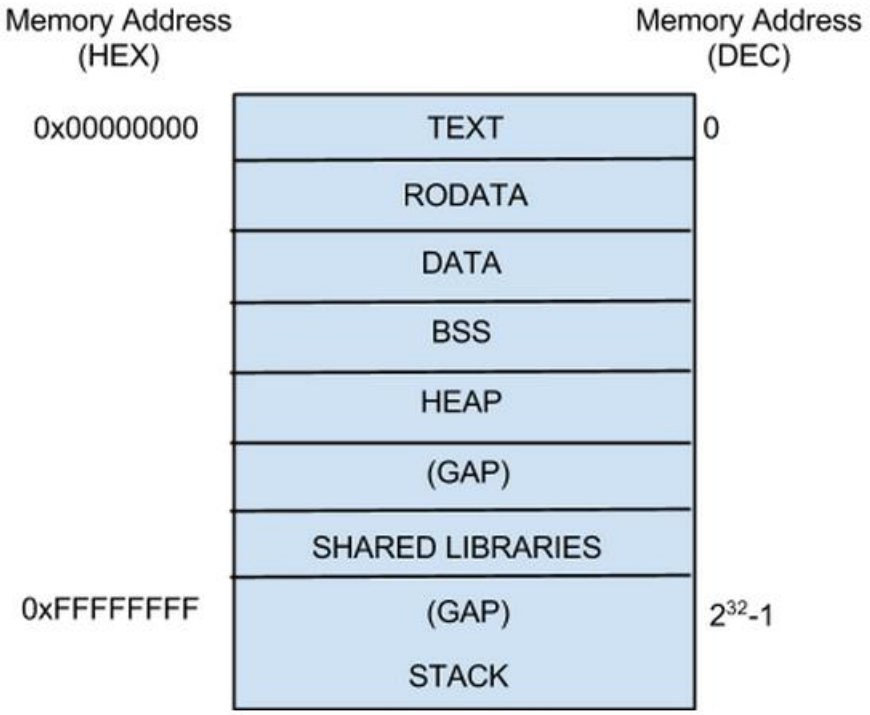
```
char buffer[64];
int *p;
p=(int *)&buffer[0];
*p = 78;
```



Sections of a Program

- The memory of the computer is used to store both the program code, and the data that the program manipulates.
- An executable program in memory is divided into sections.
  - TEXT - Instructions that the program runs
  - RODATA - Stores Read-only data. These are constants that the program uses, like strings constants or other variables defined like “const int
  - DATA – Initialized global variables.
  - BSS – Uninitialized global variables. They are initialized to zeroes.
  - HEAP – Memory returned when calling malloc/new. It grows upwards.
  - SHARED LIBRARIES – Also called dynamic libraries. They are libraries shared with other processes.
  - STACK – It stores local variables and return addresses. It grows downwards.

Sections of a Program



Address Space

- Each program has its own view of the memory that is independent of each other.
- This view is called the “Address Space” of the program.
- If a process modifies a byte in its own address space, it will not modify the same location of the address space of another process.

Printing Program Memory Addresses

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

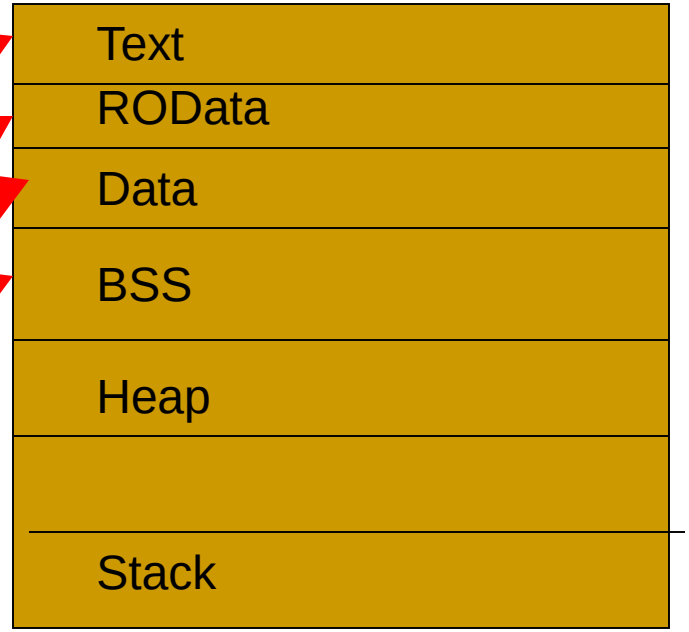
int a = 5; // Stored in data section
int b[20]; // Stored in bss
const char * hello = "Hello world";

int main() { // Stored in text
        int x; // Stored in stack
        int *p =(int*) malloc(sizeof(int)); //Stored in heap
        printf("sizeof(int)=%ld\n", sizeof(int));
        printf("sizeof(long)=%ld\n", sizeof(long));
        printf("sizeof(int*)=%ld\n", sizeof(int*));
        printf("(TEXT) main=0x%lx\n", (unsigned long)main);
        printf("(ROData) Hello=0x%lx\n", (unsigned long)hello);
        printf("(Data) &a=0x%lx\n", (unsigned long)&a);
        printf("(Bss) &b[0]=0x%lx\n", (unsigned long)&b[0]);
        printf("(Heap) p=0x%lx\n", (unsigned long)p);
        printf("(Stack) &x=0x%lx\n", (unsigned long)&x);
}
```

Note: &x means the address of variable x or where x is stored in memory.

Printing Program Memory Addresses

```
cs240@data ~/2014Fall/LectureNotes/test $ ./test1
sizeof(int)=4
sizeof(long)=8
sizeof(int*)=8
(TEXT) main=0x4005bc
(RODataa) Hello=0x400744
(Data) &a=0x601048
(Bss) &b[0]=0x601080
(Heap) p=0x1075010
(Stack) &x=0x7fff93cdd4d4
```



Simple Memory Dump of a Program

- In Lab4 you will write a memory dump function that will print the memory dump of a program:

Hint: Use char *p like it was an array

hintdump.c:

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

void mymemdump(FILE * fd, char * p , int len) {
   int i;
   fprintf(fd, "0x%016lX: ", (unsigned long) p); // Print address of the beginning of p. You need to print it every 16 bytes
   for (i=0; i < len; i++) {
      int c = p[i]&0xFF; // Get value at [p]. The &0xFF is to make sure you truncate to 8bits or one byte.

      // Print first byte as hexadecimal
      fprintf(fd, "%02X ", c);

      // Print first byte as character. Only print characters >= 32 that are the printable characters.
      fprintf(fd, "%c ", (c>=32 && c <127)?c:'.');

     if (i % 16 == 0 ) {
      fprintf(fd,"\n");
    }
   }
}

main() {
   char a[30];
   int x;
   x = 5;
   strcpy(a,"Hello world\n");
   mymemdump(stdout,(char*) &x, 64);
}
```

Simple Memory Dump of a Program

```
cs240@data ~/cs240/lab4-src $ gcc -o hintdump hintdump.c
cs240@data ~/cs240/lab4-src $ ./hintdump
0x00007FFF150B1A9C: 05 .
00 . 00 . 00 . 48 H 65 e 6C l 6C l 6F o 20   77 w 6F o 72 r 6C l 64 d 0A . 00 .
00 . 00 . 00 . A0  1B . 0B . 15 . FF  7F  00 . 00 . 00 . 00 . 00 . 00 . 00 .
00 . 00 . 00 . 00 . 00 . 00 . 00 . 00 . 00 . 00 . A5  3C < 35 5 49 I 63 c
7F  00 . 00 . 00 . 00 . 00 . 00 . 00 . 00 . 00 . 00 . A8  1B . 0B . 15 .
cs240@data ~/cs240/lab4-src $

The challenge for lab4 is to make the output of mymemdump look like this:

cs240@data ~/cs240/lab4-src $ ./mem
&x=0x7FFF89A62890
&y=0x7FFF89A628A8
0x00007FFF89A62890: 41 33 40 50 09 00 00 00 30 06 9C 50 D7 7F 00 00  A3@P....0.P..
0x00007FFF89A628A0: 94 28 A6 89 FF 7F 00 00 00 00 00 00 00 00 28 40  (........(@
0x00007FFF89A628B0: 48 65 6C 6C 6F 20 77 6F 72 6C 64 0A 00 00 00 00  Hello world.....
0x00007FFF89A628C0: FF B2 F0 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
head=0x1e83010
```

Examining the memory of the program:

```
cs240@data ~/2014Fall/lab2/lab2-src-sol $ cat mem.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void mymemdump(FILE *fd, char * p , int len);

struct X{
  char a;
  int i;
  char b;
  int *p;
};

struct List {
  char * str;
  struct List * next;
};

int
main() {
  char str[20];
  int a = 5;
  int b = -5;
  double y = 12;
  struct X x;
  struct List * head;
```

Examining the memory of the program:

```
  x.a = 'A';
  x.i = 9;
  x.b = '0';
  x.p = &x.i;
  strcpy(str, "Hello world\n");
  printf("&str=0x%lX\n", (unsigned long)str);
  printf("&a=0x%lX\n", (unsigned long)&a);
  printf("&b=0x%lX\n", (unsigned long)&b);
  printf("&x=0x%lX\n", (unsigned long)&x.a);
  printf("&y=0x%lX\n", (unsigned long) &y);

  mymemdump(stdout, (char *) &x.a, 64);
  head = (struct List *) malloc(sizeof(struct List));
  head->str=strdup("Welcome ");
  head->next = (struct List *) malloc(sizeof(struct List));
  head->next->str = strdup("to ");
  head->next->next = (struct List *) malloc(sizeof(struct List));
  head->next->next->str = strdup("cs250");
  head->next->next->next = NULL;
  printf("head=0x%lx\n", (unsigned long) head);
  mymemdump(stdout, (char*) head, 128);
}
```

Examining the memory of the program

```
cs240@data ~/2014Fall/lab2/lab2-src-sol $ ./mem
&str=0x7FFFCFB29B50
&a=0x7FFFCFB29B4C
&b=0x7FFFCFB29B48
&x=0x7FFFCFB29B20
&y=0x7FFFCFB29B40
0x00007FFFCFB29B20: 41 E3 D1 41 09 00 00 00 30 B6 2D 42 30 7F 00 00  AA....0-B0..
0x00007FFFCFB29B30: 24 9B B2 CF FF 7F 00 00 B7 B1 DA 41 30 7F 00 00  $..A0..
0x00007FFFCFB29B40: 00 00 00 00 00 00 28 40 FB FF FF FF 05 00 00 00  ......(@....
0x00007FFFCFB29B50: 48 65 6C 6C 6F 20 77 6F 72 6C 64 0A 00 00 00 00  Hello world.....
head=0x1adc010
0x0000000001ADC010: 30 C0 AD 01 00 00 00 00 50 C0 AD 01 00 00 00 00  0.....P.....
0x0000000001ADC020: 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00  ........!.......
0x0000000001ADC030: 57 65 6C 63 6F 6D 65 20 00 00 00 00 00 00 00 00  Welcome ........
0x0000000001ADC040: 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00  ........!.......
0x0000000001ADC050: 70 C0 AD 01 00 00 00 00 90 C0 AD 01 00 00 00 00  p...........
0x0000000001ADC060: 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00  ........!.......
0x0000000001ADC070: 74 6F 20 00 00 00 00 00 00 00 00 00 00 00 00 00  to ..............
0x0000000001ADC080: 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00  ........!.......
cs240@data ~/2014Fall/lab2/lab2-src-sol $

Colors:
   str a b

Variable            Address          Value
------------------- ---------------- -------------
str                 0x7FFFCFB29B50 48 65 6C 6C 6F 20 77 6F 72 6C 64 0A 00 Hello world
a                   0x7FFFCFB29B4C 05 00 00 00  (5)
b                   0x7FFFCFB29B48 FB FF FF FF  (-5)
```

Common Errors with Pointers

- Using a pointer without initializing crashes the program with SEGV

```
int * p; // p value is NULL (0) or unknown
*p = 10; // Program writes to invalid
                    // address and crashes.
```

Fix:

Initialize pointer:

```
int x;
int *p;
 p = &x;
*p= 10;
```

another solution:

```
int *p;
p = (int*)malloc(sizeof(int));
*p = 10;
// Use *p
free(p)
```

Common Errors with Pointers

- Not allocating enough memory

```
int *array;
int n=20;
array=(int *)malloc(n * sizeof(int) );
                // 20 * 4 bytes = 80 bytes allocated
array[5]=20;    //OK
array[25]=7;    //Wrong. C blindly tries to assign 7 to the
                // 25th position, which does not
               // exist as valid memory
```

Sum of a pointer and an int

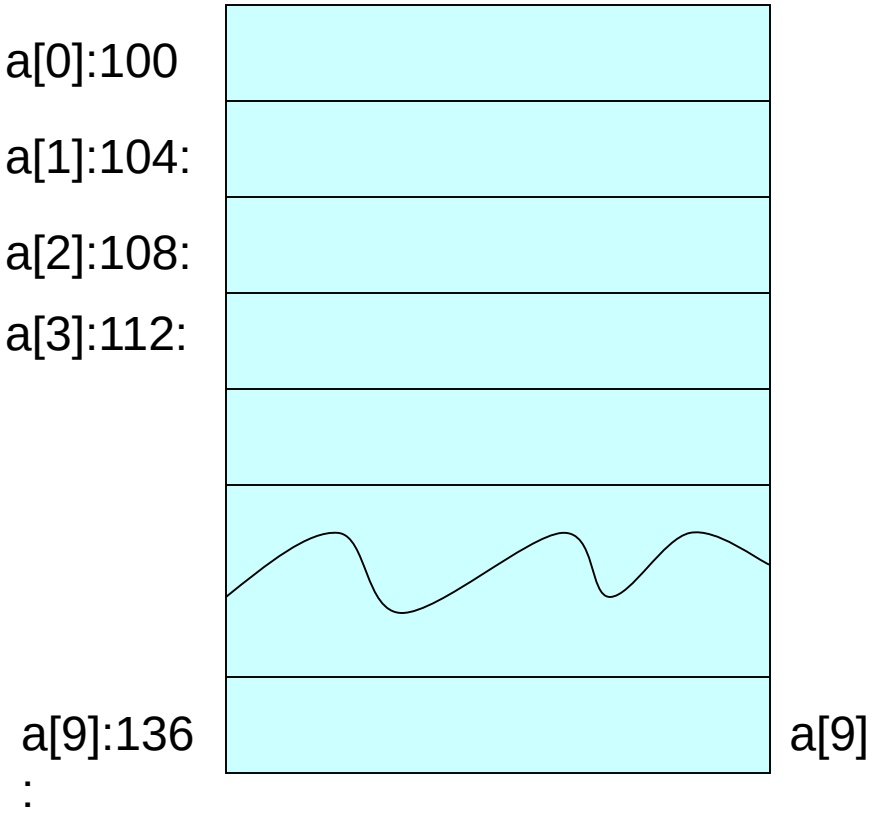
- A pointer is an address when you add an integer i to a pointer of type T, the integer is multiplied by the size of the type T and added to the pointer.

```
int a[10];  // Assume a is in location 100
int *p;
p = &a[0];  // Assume p is 100
p = p + 1; // Since *p is of type int the
           // constant 1 will be multiplied
           // by sizeof(int) before adding.
           // p now is 104 and points to a[1].

p = p + 2; // p is now 104 + 2 * sizeof(int)) =
       // 104 + 2 * 4 = 112 which points to a[3]
```

Pointers and Arrays

int a[10];




```
int a[10];
int *p;
p = &a[0];  // p == 100
p = p + 1;  // Since p is of type int* the
            // constant 1 will be multiplied
            // by sizeof(int) before adding.
            // p is now  104 and points to a[1].

p = p + 2;
    // p is now 104 + 2 * sizeof(int)) =
    // 104 + 2 * 4 = 112 which points to a[3]
```

Pointers And Arrays

- In C, pointers are arrays and arrays are pointers.

  For example,

  int a[10];

- ***a*** is an array of 10 elements of type int.
- also, ***a*** is a pointer to the first element of the array

  a is the same as &a[0]

  a[0] is the same as *a

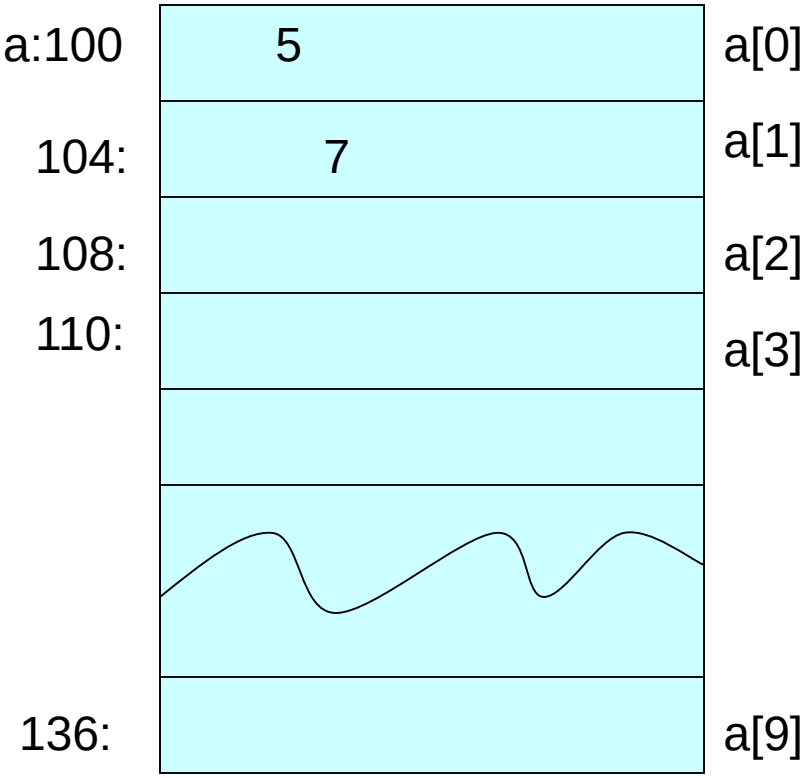
  a[1] is the same as *(a+1)

  in general:

  a[i] is the same as *(a+i)

Pointers and Arrays

int a[10];



a[1]=7
a + 1 == 104
*(a+1) == 7

Pointer Equivalence

- Assume that

  int a[10];

  int i=8;

- We have that

  a[i] is the same as

  *(a+i) is the same as

  *(&a[0]+i) is the same as

  (int)((char*)&a[0] + i*sizeof(int))

Pointer Comparison < >= >= == !=

- You can compare two pointers.
- A pointer is an unsigned long that is the address of the value it is pointing to

```
// Version 1. Adding elements of an array of
// integers using array operator.
int sum (int *a, int n) {
    int s = 0;
    for (int  i = 0 ; i < n ; i++) {
        s += a[i];
   }
   return s;
 }
```

Add Elements in an Array Using Pointers

```
// Version 2. Add elements of an array using pointers.
int sum (int *a, int n) {
        int *p;
        int *pend;
        p = &a[0];
        pend = p + n;
        int sum = 0;
        while (p < pend){
          sum += *p;
          p++; // point p to the next element
        }
        return sum;
}
```

- The Version 2 that uses pointers is faster than the one using arrays because array indexing a[i] requires multiplication.
- a[i] is computed as *(int*)((char*)a+ i * sizeof(int))

Pointers as arrays

By the same token, pointers can be treated as arrays

```
int *p;
p = a;
a[0] = 5;
printf("p[0] = %d\n", p[0]);

Output:
p[0] = 5;

a[7] = 19;
printf("p[7] = %d\n", p[7]);

Output:
p[7] = 19;
```

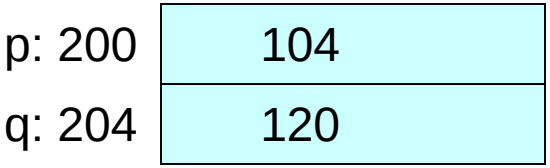
Pointer subtraction

- Assuming that p and q are pointers of the same type, (q – p) will give the number of objects between q and p.

```
int *p, *q;
int buffer[20];
p = &buffer[1];
q = &buffer[5];
printf("q - p =%d\n",q-p);
// it is going to print 16 / 4 = 4
```

Pointer Subtraction

p = &buffer[1];
q = &buffer[5];



q-p == (120-104)/4 ==4

printf("q - p =%ld\n",q-p); //Prints 4

Passing Arguments by Value

- All arguments in a function in C are passed by a “value”, that is the value of the variable or constant is passed to the function

```
void f (int x)
        printf(“x=%d\n”,x);
        x = x+1;
}
...
int y = 5;
f(y); // output: x = 5
printf(“y = %d\n”,y); // output: y = 5
```

- y’s final value is 5 even when x = x+1 in f(y)
- That is because only the value of y (that is 5) is passed to f(y).
- Internally, x and y are 2 different variables.

Passing an Argument by Reference

- Passing by reference means that the variable passed in the argument can be modified inside the function.
- This is emulated by passing a pointer as a variable and treating the variable as a pointer.

Example: we want to print the value of the variable and increase it.

```
void f2(int *x){
        printf("*x=%d\n",*x);
        *x = *x+1;
}
...
int y = 5;
f2(&y); // pass a pointer to y
printf("y = %d\n",y); // output: y = 6
```

Swap Two Numbers Using Passing Arguments by Reference

```
void swap(int *px, int *py)
{
        int temp;
        temp = *py;
        *py = *px;
        *px =  temp;
}

int main(){
        int x,y;
        x = 5;
        y = 9;
        printf("Before x=%d y=%d\n", x,y,);// Before x=5 y=9
        swap(&x,&y);
        printf("After x=%d y=%d\n",x,y); // After x=9 y=5
}
```

Dangling Reference Problem

- The lifetime of local variables is limited to the time the function that contains the variables is called.
- You should not return a pointer to any of these variables.

```
Example:
int *m()
{
  int i;
  i=2;
  return &i;
}
main()
{
 int *x;
 x=m();
 printf(“Hello”);
 printf("*x=%d",*x);
}
OUTPUT is undefined *x=?????;
The space used by i in m() will be reused by other local
   variable in the first printf so the value of *x is
   undefined.
```

String Operations with Pointers

- We can rewrite many of the string functions using pointers

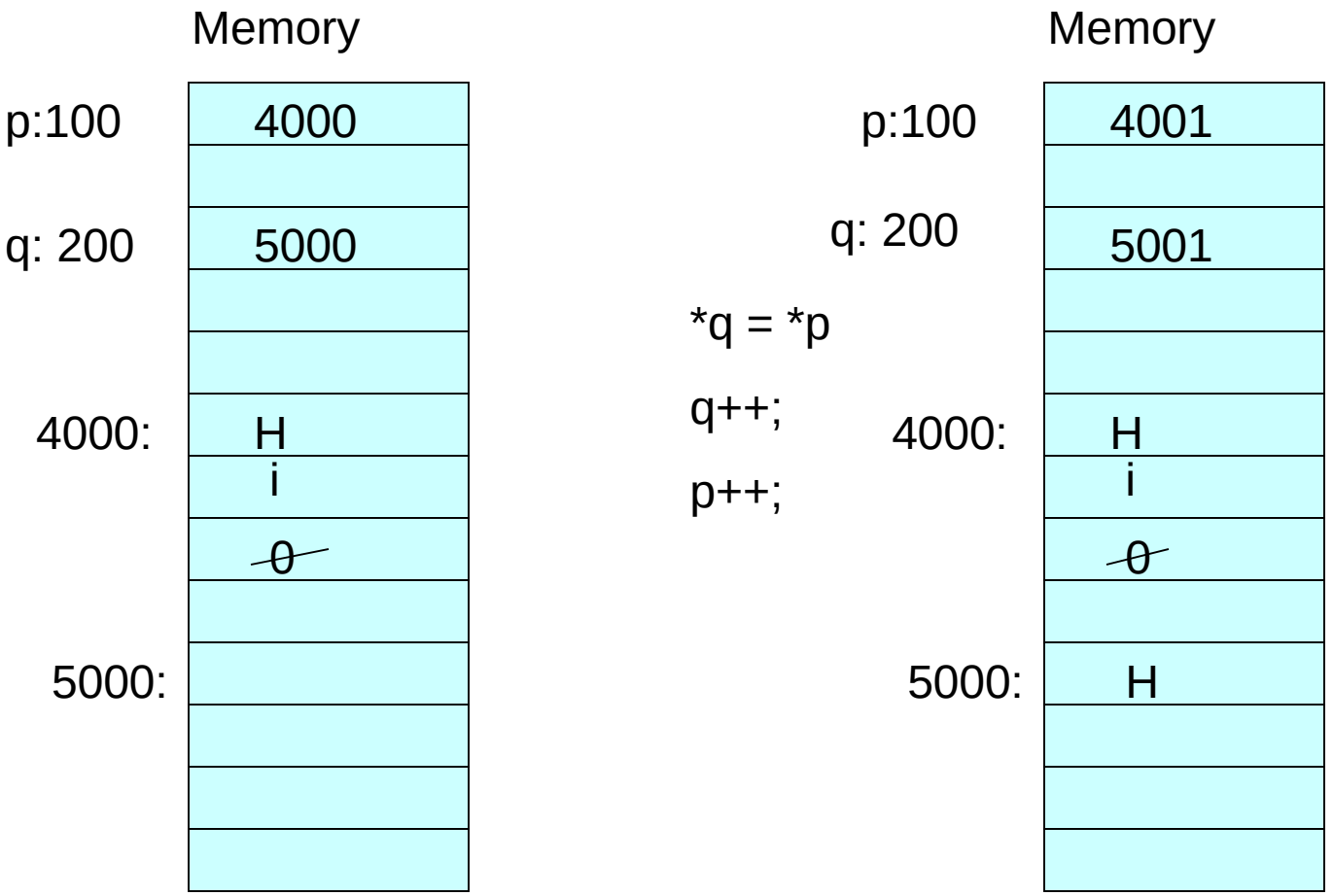
Implementation of strcpy using pointers

```
Implementation of strcpy using pointer operations.

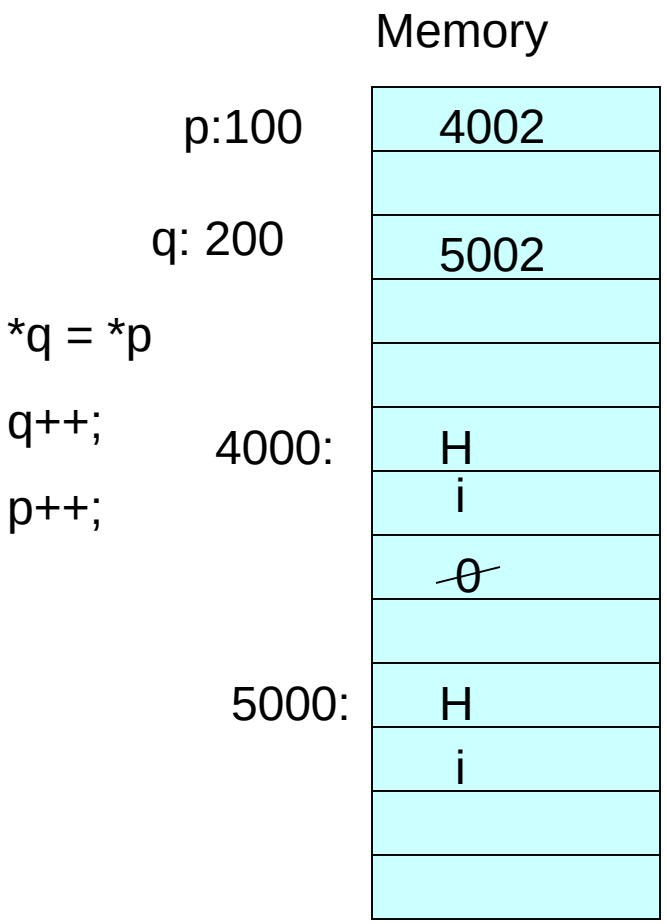
   char * strcpy(char * dest, char * src) {
       char * p = src;
       char * q = dest;
       while (*p != '\0') {
          *q = *p;  // the three lines can be written as *q++=*p++
           p++;     // but this is difficult to read so don't do
          q++;      // it.
       }
       *q = '\0';
      return dest;
   }
```

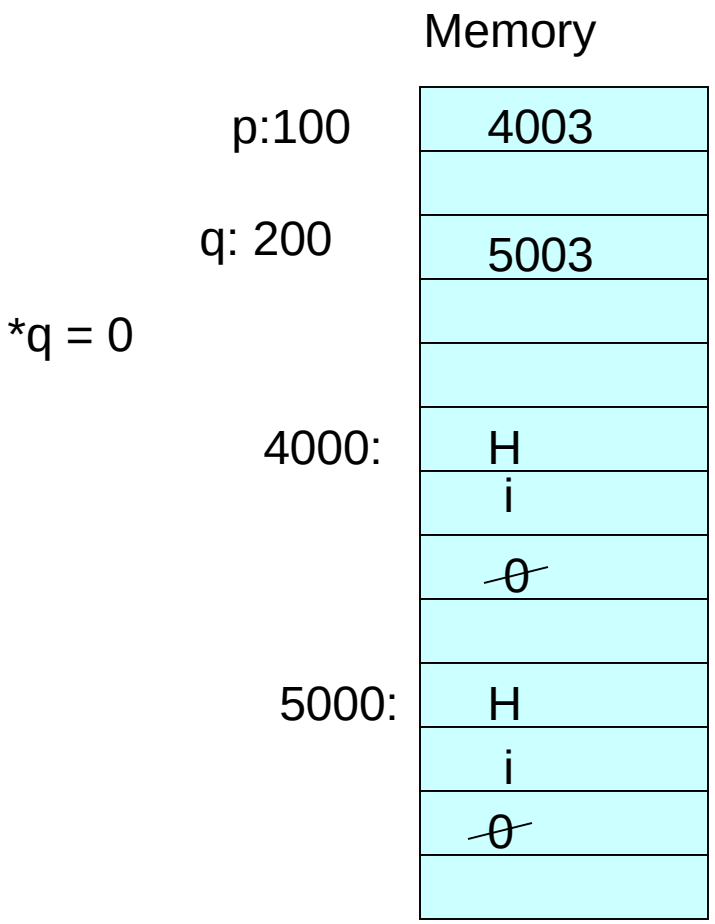
src is a pointer (address) that points to the string to copy.
*src is the character at that location.

Implementation of strcpy using pointers



Implementation of strcpy using pointers





strlen using array operator

```
int strlen(char*s)
{
 int i=0;
 while(s[i])
 {
  i++;
 }
 return i;
}
```

strlen using pointers

```
int strlen(char *s)
{

 int i=0;
 while(*s)
 {
  i++;
  s++;
 }

 return i;
}
```

strcpy using pointers

```
char *strcpy(char*dest, char *src)
{
 char*p =src;
 char*q=dest;

 while(*p)
 {
  *q=*p;
  q++;
  p++;
 }

 *q='\0';

 return dest;
}
```

Strcat using pointers

```
char *strcat(char*dest, char*src)
{
 char *p,*q;
 // make p point to end of dest
 p=dest;
 while(*p) { p++; }
 q=src;
 //copy from *q to *p
 while(*q)
 {
  *p=*q;
  p++;
  q++;
 }
 *p='\0';
 return dest
}
```

Common Mistakes: Uninitialized string pointers

```
char*s;
strcpy(s,"Hello\n");
// wrong s is pointing to NULL or to an
// unknown value
// strcpy will cause SEGV because
// s is not pointing to valid memory
```

- Solution:

```
char s[7];
strcpy(s,"Hello\n");

or
int len = strlen(str);
char *s=(char*)malloc(len+1);
            // len has to be longer than string
strcpy(s,str);
```

Common Mistakes: Not enough space for a string null terminator

- Not enough space. ***s*** needs to include '\0' at the end of the string. "Hello\0" takes 6 chars and not 5.

  ```
  char s[5];
  strcpy(s,"Hello");
  ```

- Fix:

  ```
  char s[6];
  strcpy(s,"Hello");
  ```

Common Mistakes: Not enough space

```
char s[6];
strcpy(s,"Hello");
strcat(s,"world");
```

- s needs to have a tleast 12 characters to store "Hello world \0"
- The last strcat may give a SEGV or overwrite to the memory of other variables
- Solution: Allocate enough memory

```
char s[20];
strcpy(s,"Hello");
strcat(s," world");
```

strcmp(s1, s2) - String Comparison

```
r=strcmp(s1,s2);
r==0; //if s1 is equal to s2
r > 0 if s1 > s2
r < 0 if s1 < s2

strcmp("banana","apple"); > 0

apple
.
.
.
banana

strcmp("apple", "banana");< 0
```

strdup –duplicate string

- strdup creates a copy of the string using malloc

```
char*strdup(char *s1)
{
 char*s=(char *)malloc(strlen(s1)+1);
 strcpy(s,s1);
 return s;
}

 char a[20];
 strcpy( a,"Hello ");
 char*s=strdup(a);
```

- s and a are different strings with the same content.

Using string operations safely with strings functions that check the string length

- The functions we have seen do not check for the length of destination string.
- There are equivalent functions that assume maximum of "n" characters in the destination. They are safer.

```
strncpy(char*dest, char *src, int n)
   //copies src in dest
   //a maximum of n chars
   // if src is longer than n chars
   // dest will not have '\0'
   // at the end otherwise it will put a
   //'\0' at the end of dest

strncat(char*dest, char *src, int n)
   //concatenates src into dest
   //upto n chars the length of
   //dest should be enough to
   //store dest and src. n includes both dest and src
```

Passing arrays as a parameter

- You can pass an array as a parameter by passing a pointer
- Arrays are pointers and viceversa

int sum(int a[], int size)

same as

int sum(int *a,int size)

Passing arrays as a parameter

```
int sum(int *a,int size)
{
	int sum=0;
	int i;

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

main()
{
	int a[]={7,8,3,2,1}; // Initializing array

	n =sizeof(a)/sizeof(int);
	printf("sum(a)=%d\n",sum(a,n) );
}
```

Allocating arrays

- Allocating arrays statically

```
int array[200];
```

- Allocating arrays using malloc

```
int *array;
int n=200;
array=(int*)malloc(n*sizeof(int));
if(array==NULL)
{
  perror("malloc");
  exit(1);
}
```

structs

- A struct is a compound type:

```
struct z
{
 int a;
 double x;
 char *s;
}r;
```

- If you want to refer to any of the members:

```
r.x = 23.5;
r.s = "Hello";
r.a = 3 ;
```

structs member variables

- You can name a struct so you can refer to it in multiple places

```
struct STUDENT
{
  char*name;
  double grade;
};
```

- Then define a variable as follows:

```
struct STUDENT peter, mary;
peter.name="peter";
peter.grade=100;
mary.name="mary";
mary.grade=100;
```

Pointers to struct

- If the struct variable is a pointer

```
struct STUDENT *p;
p=&peter;
```

- Then we can refer to fields of peter as follows:

```
(*p).name="peter";
(*p).grade=100;
```

- Since this way of referring to structs using pointers is common C has also the equivalent notation.

```
p->name="peter";
p->grade=100;
```

typedef struct

- Using typedef and structs we can also write:

```
typedef struct
{
 char*name;
 double grade;
}STUDENT;
STUDENT peter, mary;
```

- Or we can use both the struct name and typedef:

```
typedef struct STUDENT
{
 char*name;
 double grade;
}STUDENT;
```

- Now you can define variables as:

```
struct STUDENT peter, mary;
 //or
STUDENT peter, mary;
```

Malloc and Dynamic Memory

- malloc() allows us to request memory from the OS while the program is running.
- Instead of pre-allocating memory (eg. Maximum matrix) when the program is compiled, we can allocate it at runtime using malloc().

  p=(T*) malloc(sizeof(T)); // T is the type of the object to allocate.

- This allocates memory of sizeof(T) bytes and assigns it to ‘p’ (after casting)
- p=calloc(1,sizeof(T)) allocates memory for an object of type T like in malloc
- The only difference is that calloc() initializes memory to 0.
- calloc() calls malloc() internally and then initializes memory to 0

Example Using malloc

```
int *p;
p=(int *) malloc(sizeof(int));
if(p==NULL) {
  //Not enough memory
  perror("malloc");
  exit(0);
}
*p = 5;

// malloc may return NULL if there is not
// enough memory in the process.
```

Using malloc

- Malloc is used to allocate memory from the system.
- It is similar to “new” in Java but there is no constructor associated to it.
- Example:

```
int *p; // p is a pointer to an int. However, since
        // p is not initialized, it can be
        // pointing anywhere in memory
p = (int*) malloc(sizeof(int)) // Now we can store a value
*p = 5;
```

If we try to store into p without initializing it the program will crash.

Example:

```
int *p;
*p = 5 // ***** CRASH!!!!
```

Allocating an array using malloc

- You may allocate memory for an array in the same way:
- Example:

```
int * array;
int N = 10;
array = (int *) malloc(N*sizeof(int));
// Now array can be used:
array[4] = 10;
```

Memory Deallocation

- When the program does not need the memory anymore, you can free it to return it to the malloc free list.

```
free(p);
```

- The free list contains the list of objects that are no longer in use and that can be allocated in future malloc() calls.
- There is no garbage collection in C.

realloc(oldblock, newsize)

- If you allocate memory with malloc then you can resize it.

  n=2*n;

  array=(int*)realloc(array, n*sizeof(int));

- realloc(oldblock, newsize) does the following:
  - Allocates a new block with the new size
  - Copies the old block into the new one block
  - Frees the old block
  - Returns a pointer to the new block

Single Linked List - Header File

single_linked_list.h:

```
struct SLLENTRY
{
 char * name;
 char * address;
 struct SLLENTRY * next;//pointer to the next entry
};
typedef struct SLLENTRY SLLENTRY;

struct SLLIST{
   SLLENTRY * head;
};
typedef struct SLLIST SLLIST;
```

Single Linked List - Header File

```
//Interface

// Initialize a new list
Void sllist_init(SLLIST * sllist);

// Print List
void sllist_print(SLLIST * sllist);

// Given a name lookup the address. Return null if name not in list.
char * sllist_lookup(SLLIST * sllist, char * name);

// Add a new name, address to the list. Return 1 if name exist or 0ow
int sllist_add(SLLIST * sllist, char *name, char*address);

// Remove name from linked list. Return 1 if name exists or 0 otherwi
int sllist_remove(SLLIST*sllist, char*name);
```

Single Linked List - Test Main

sllist_test.c:

```
#include "single_linked_list.h"

int main() {
   SLLIST sl;
   int result;
   sllist_init(&sl);   // we create a single linked list sl


   //Add two items to the list
   result = sllist_add(&sl,"Peter Parker", "38 2nd,NY");
   result = sllist_add(&sl,"Clark Kent", "78 Super,Metro");

   // Print list
   sllist_print(&sl);
```

Single Linked List - Test Main

```
   char* addr  = sllist_lookup(&sl,"Clark Kent");
   if( addr == NULL) {
      printf("cannot find Clark's address\n");
      exit(1);
   }
   else
   {
      printf("Clark's address is %s\n", addr);
   }

   result = sllist_remove(&sl,"Clark Kent");
   if (result == 0)
   {
     printf("Cannot remove Clark's address\n");
     exit(1);
   }

   sllist_print(&sl);
}
```

Single Linked List - Initialize List

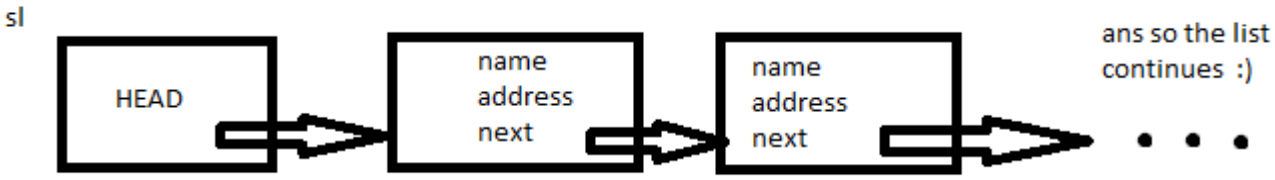
single_linked_list.c:

```
#include"single_linked_list.h"

// Initialize Linked list
void sllist_init(SLLIST * sl)
{
    // the list is initially empty so we initialize it to NULL
    //This is equivalent to saying (*sl).head = NULL
    sl->head = NULL;
}
```

Single Linked List - Print List

```
void sllist_print(SLLIST * sl)
{
  // Traverse the list and print each element
  SLLENTRY *p;
  p = sl->head;  //we initialize p to the head
  while(p != NULL) {
    printf("name = %s addr = %s\n",
           p->name, p->address);
    p = p->next;
  }
}
```



Single Linked List - Lookup

```
char *sllist_lookup(SLLIST*sl, char* name)
{
   SLLENTRY *p;
   p = sl->head;
   while(p != NULL)
   {
        if(strcmp(name,p->name) == 0)
        {
                // YES ! we have found the name
           return(p -> address);
        }
        p = p->next;
   }
   // Now we are outside the while loop
   // this will happen if we have reached the end
   // of the list and still not found the name.
   // If this is the case then we return NULL

   return NULL;
}
```

Single Linked List – Add 1

```
int sllist_add(SLLIST* sl, char* name, char*address)
{
   SLLENTRY *p;
   // we first have to check if the name already exists
   p = sl->head // initialize p to head
   while(p != NULL)
   {
        if(strcmp(p->name,name) == 0)
        {
                // this means name already exists
                // if it already exists then we substitute
                // the old address for the new one
                // we need to free the previous address
                // since it was allocate with strdup
                free(p->address);
                p->address = strdup(address); //create a duplicate
                return(0);
        }
        p = p->next;  // incrementation for the while loop
   }
```

Single Linked List – Add 2

```
   // we have exited out of the while loop and
   // name does not exist. We need to create a new entry

   p = (SLLENTRY*)malloc(sizeof(SLLENTRY));

   // we make duplicate of name and address
   p->name = strdup(name);
   p->address = strdup(address);

   // we now put the new entry at the
   // beginning of the list

   p->next = sl->head;
   sl->head = p;

   return(1);
}
```

Single Linked List – Remove 1

```
int sllist_remove(SLLIST *sl, char*name);
{
    SLLENTRY* p = sl->head;
    SLLENTRY* prev = NULL;
    //prev is a pointer which points to the previous entry
    //we first have to find the entry to remove
    while(p != NULL)
   {
        if(!strcmp(p->name,name))
        {
              break; // entry is found
        }
        prev = p;
        p = p->next;
   }
```

Single Linked List – Remove 2

```
if(p == NULL)
{
     // element does not exist
     return 0;
}
// Now the entry pointed by p has the name we are looking for.
// prev points to the element before p

// There are two cases for p
// First Case: p is the first element. Therefore,
//    prev will point to NULL
// Second case: p is an internal node (includes the
//    last node in this case)
```

Single Linked List – Remove 3

```
    if (prev == NULL) {
       sl->head = p->next;
    }
    else
    {    prev->next = p->next ;
    }
    // Now we have skipped the element but we are not
    // yet done removing it since we have to free the memory
    // Before we free p , we have to free the address and
    // the name associated with p because we used strdup
    // to allocate memory to it

    free(p->name);
    free(p->address);
    free(p);

    // the order here is critical, we first free the name and address since
       p points to them

    return 1;
}
```

Memory Allocation Errors

- Explicit Memory Allocation (calling free) uses less memory and is faster than Implicit Memory Allocation (GC)
- However, Explicit Memory Allocation is Error Prone
  1. Memory Leaks
  2. Premature Free
  3. Double Free
  4. Wild Frees
  5. Memory Smashing

Memory Leaks

- Memory leaks are objects in memory that are no longer in use by the program but that ***are not freed***.
- This causes the application to use excessive amount of heap until it runs out of physical memory and the application starts to swap slowing down the system.
- If the problem continues, the system may run out of swap space.
- Often server programs (24/7) need to be “rebounced” (shutdown and restarted) because they become so slow due to memory leaks.

Memory Leaks

- Memory leaks is a problem for long lived applications (24/7).
- Short lived applications may suffer memory leaks but that is not a problem since memory is freed when the program goes away.
- Memory leaks is a “slow but persistent disease”. There are other more serious problems in memory allocation like premature frees.

Memory Leaks

Example:

```
int * ptr;
ptr = (int *) malloc(sizeof(int));
*ptr = 8;
… Use ptr …
ptr = (int *) malloc(sizeof(int));
// Old block pointed by ptr
// was not deallocated.
```

Premature Frees

- A premature free is caused when an object that is still in use by the program is freed.
- The freed object is added to the free list modifying the next/previous pointer.
- If the object is modified, the next and previous pointers may be overwritten, causing further calls to malloc/free to crash.
- Premature frees are difficult to debug because the crash may happen far away from the source of the error.

Premature Frees

Example:

```
int * p = (int *) malloc(sizeof(int));
* p = 8;
free(p); // delete adds object to free list
         // updating header info
…
*p = 9;  // next ptr will be modified.
… Do something else ...
int *q = (int *) malloc(sizeof(int));
 // this call or other future malloc/free
 // calls will crash because the free
 // list is corrupted.
 // It is a good practice to set p = NULL
// after delete so you get a SEGV if
// the object is used after delete.
```

Premature Frees. Setting p to NULL after free.

- One way to mitigate this problem is to set to NULL the ptr after you call free(p)

```
int * p = (int *) malloc(sizeof(int));
* p = 8;
free(p); // delete adds object to free list
p = NULL; // Set p to NULL so it cannot be used
*p = 9;  // You will get a SEGV. Your program will
   crash and you will know that you have already
   freed p.
```

Double Free

- Double free is caused by freeing an object that is already free.
- This can cause the object to be added to the free list twice corrupting the free list.
- After a double free, future calls to malloc/free may crash.

Double Free

Example:

```
int * p = (int *) malloc(sizeof(int));
free(p); // delete adds object to free list
.. Do something else ….
free(p); // deleting the object again
         // overwrites the next/prev ptr
         // corrupting the free list
         // future calls to free/malloc
         // will crash

// Also to prevent this problem you may set p
   to NULL after free.
```

Double Free. Setting p to NULL after free.

```
int * p = (int *) malloc(sizeof(int));
free(p); // delete adds object to free
  list
P = NULL;
.. Do something else ….
free(p); // deleting the object again
         // SEGV. And the you can
         // see the stack trace
```

Wild Frees

- Wild frees happen when a program attempts to free a pointer in memory that was not returned by malloc.
- Since the memory was not returned by malloc, it does not have a header.
- When attempting to free this non-heap object, the free may crash.
- Also if it succeeds, the free list will be corrupted so future malloc/free calls may crash.

Wild Frees

- Also memory allocated with *malloc()* should only be deallocated with *free()* and memory allocated with *new()* should only be deallocated with delete().
- Wild frees are also called “free of non-heap objects”.

Wild Frees

Example:

```
int q;
int * p = &q;

free(p);
 // p points to an object without
 // header. Free will crash or
 // it will corrupt the free list.
```

Wild Frees

Example:

```
char * p = (char*)malloc(100);
p=p+10;

free(p);
// p points to an object without
// header. Free will crash or
// it will corrupt the free list.
```

Memory Smashing

- Memory Smashing happens when less memory is allocated than the memory that will be used.
- This causes overwriting the header of the object that immediately follows, corrupting the free list.
- Subsequent calls to malloc/free may crash
- Sometimes the smashing happens in the unused portion of the object causing no damage.

Memory Smashing

Example:

```
char * s = (char*)malloc(8);
strcpy(s, "hello world");

// We are allocating less memory for
// the string than the memory being
// used. Strcpy will overwrite the
// header and maybe next/prev of the
// object that comes after s causing
// future calls to malloc/free to crash.
// Special care should be taken to also
// allocate space for the null character
// at the end of strings.
```

Debugging Memory Allocation Errors

- Memory allocation errors are difficult to debug since the effect may happen farther away from the cause.
- Memory leaks is the least important of the problems since the effect take longer to show up.
- As a first step to debug premature free, double frees, wild frees, you may comment out free calls and see if the problem goes away.
- If the problem goes away, you may uncomment the free calls one by one until the bug shows up again and you find the offending free.

Debugging Memory Allocation Errors

- There are tools that help you detect memory allocation errors.
  - IBM Rational Purify
  - Bounds Checker
  - Insure++
  - Valgrind
  - Dr. Memory

Pointer to Functions

- In the same way that you have pointers to data, you can have pointers to functions.
- Pointers to functions point to functions in the text segment.

Example:

```
//FUNCPTR is a type of pointer to a function
//that takes no arguments and returns void.
typedef void (* FUNCPTR)(void);

void hello(){
       printf("Hello world\n");
}

main(){
       FUNCPTR funcptr;
       hello();

       //call hello through funcptr
       funcptr = hello;
      (* funcptr)();  //the same as calling funcptr();
}
Output:
   Hello world -> Printed by "hello()"
   Hello world -> Printed by (*funcptr)();
```

Use of pointer to functions: Sorting Any Array 多态性

- Polymorphism: You can write in C functions that can be used for variables of multiple types. Ex: Sorting function which is able to sort arrays of any type; Comparison function is passed as argument.

```
//generic pointer that can be used to point to any type
typedef int (*compare_func)(void *e1, void *e2);

// Function that compares two integers
int compInt(void *e1, void *e2)
{
        int * p1 = (int *)e1;
        int *p2 = (int *)e2;
        if(*p1 > *p2){ return 1; }
        else if(*p1< *p2){return -1;}
        else{return 0;}
}
```

Use of pointer to functions: Sorting Any Array

```
void sortAnyArray(void * array, int n, int elementsize,
                          compare_func comp)
{
   // Temporal memory used for swapping
   void * tmp = malloc(elementsize)
   int i, j;
   //use bubble sort
   for(i=0; i<n; i++)
   {
       for(j=0; j<i;j++)
       {
                //compute pointer to entry j
                void *e1 = (void*)((char*)array+j*elementsize);

                //The reason we convert to char* because
                //it is not possible to do pointer
                //arithmetic with void

                //compute pointer to entry j+1
                void *e2 = (void*)((char*)array+(j+1)*elementsize);
```

array[j] = *array+j*sizeof(array)
array is type void, sizeof(array) doesnt work
So we need to pass by elementsize
manually

Use of pointer to functions: Sorting Any Array

```
            //sort in ascending order
            //swap if e1>e2
            //element at j is larger than in j+1
            if((*comp)(e1,e2)>0)
            {
                  //now we need to swap
                  //we need to swap the entries pointed by e1 and e2;
                  memcpy(tmp, e1,elementsize);
                  memcpy(e1,e2,elementsize);
                  memcpy(e2, tmp, elementsize;)
            }
        }
    }
    // Free memory used for swap
   free(tmp);
}
```

Use of pointer to functions: Sorting Any Array

```
//Using sorting function
int main(){
        // Sorting array of type int
        int a[] = {7,8,1,4,3,2}        # of elements = sizeof(array)/sizeof(typeofarray)
        int n=sizeof(a)/sizeof(int);
        sortAnyArray(a, n, sizeof(int), compInt);
        int i=0;
        for(i=0;i<n;i++){
                printf("%d \n", a[i]);
        }

        // Sorting array of type string
        char * strings[] = {"pear", "banana", "apple", "strawberry"}
        n = sizeof(strings)/sizeof(char*);
        sortAnyArray(strings, n, sizeof(char*), compstr);
        for(i=0; i<n; i++)
        {
                printf("%s\n", strings[i]);
        }
}
```

Use of pointer to functions: Sorting Any Array

char* strings[] visualize as a list of pointers of chars
a pointer to the pointer

```
// String comparison
int compstr(void *e1, void *e2)
{
        char **p1 = (char**)e1;
        char **p2 = (char**)e2;
        //p1, p2 is a pointer to the string
        if((strcmp(*p1, *p2)>0) {
          return 1;
        }
        else if((strcmp(*p1, *p2)<0)
        {
          return -1;
        }
        else
        {
          return 0;
        }
}
```

Example of Pointers to Functions: Iterating over a List

- We can use pointers to functions to iterate over a data structure and call a function passed as parameter in every element of the data structure.
- This is called an iterator or mapper

Implementation of llist_mapper:

single_linked_list.h:

… **Other definitons**…

```
typedef void (*SLLISTFUNC)(char* name, char* value);
```

Example of Pointers to Functions: Iterating over a List

single_linked_list.c

```
//call llistfunc() in every element of linked list
void sllist_mapper(SLLIST *sl, SLLISTFUNC func))
{
      SLLENTRY *e;
      e = sl->head;
      while(e != NULL) {
        (*func)(e->name, e->value);
        e = e->next;
      }
}
```

Example of Pointers to Functions: Iterating over a List

main.c

```
#include "linked_list.h"
void printEntry(char *name, char* value)
{
        printf("name:%s  value:%s\n", name, value);
}

Void printNamesWithA(char* name, char* value)
{
        if(name[0] =='A' || name[0] == 'a')
        {
          printf("name = %s value = %s\n", name, value);
        }
}
```

Example of Pointers to Functions: Iterating over a List

```
main()
{
        //read a linkedlist from disk
        SLLIST list;
        list = llist_init(&llist);
        list_read(&llist, "friends.rt");

        // Print all entries
        sllist_mapper(&list, printEntry);

        // print only entries that start with "a"
        list_mapper(&list, printNamesWithA);

}
```

Unions

- A union is like a struct but all the elements use the *same memory*.
- That means that modifying one element will overwrite the other elements.
- Unions are used to see the same memory area as different types.

- Example:

```
#include <stdio.h>
union A {
  int i;
  double f;
  char s[4];
};

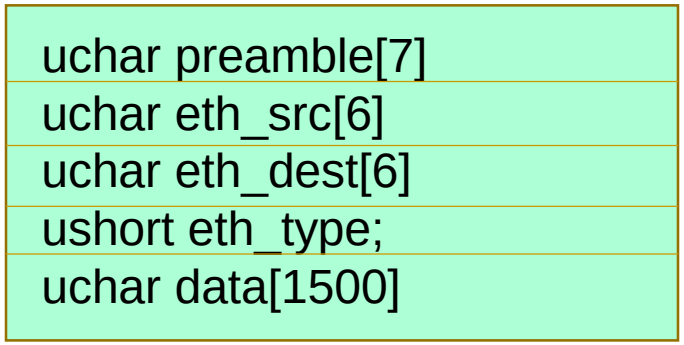
main() {
        union A x;
        x.i = 65;
        printf("x.i=%d\n",x.i);
        printf("x.s=%s\n",x.s);
        printf("x.f=%lf\n",x.f);
}

grr@data ~/tmp $ ./a
x.i=65
x.s=A
x.f=0.000000
```

Accessing Raw Memory

- Very often we need to read arbitrary values from memory stored in a char buffer[] array.
- Example:
  - An Ethernet driver receives a packet in an array char packet[MAXPACKET].
  - We know that the packet has the following format:

char packet[MAXPACKET]:



Accessing Raw Memory

- We have that the Ethernet packet is:

  char packet[MAXPACKET];

  ethernet_read(&packet); // Get one packet from driver

- *And the contents can be represented as a struct:*

```
struct EHDR {
    uchar preamble[7]
    uchar eth_src[6]
    uchar eth_dest[6]
    ushort eth_type;
    uchar data[1500];
    };
```

- To get the elements we create a pointer to this struct and

```
struct EHDR *pehdr = (struct EHDR *)  packet;
printf("eth_src=%2x:%2x:%2x:%2x:%2x:%2x\n", pehdr->eth_src);
printf("eth_type=%d\n", pehdr->eth_type);
```

The UNIX Interface

Stream Files

- Declared in the header
  ```
  #include <stdio.h>
  ```
- You need a file handle to use files
  ```
  FILE *fileHandle;
  ```
- To open a file you would use:
  ```
  fileHandle = fopen(fileName, fileMode);
  ```
- fileMode

  "r" -> open file for reading

  "w" -> open file for writing

  “a" -> open file for appending. Created if it does not exist.

  "r+"-> open file for both read and write. But the file must exist.

  "w+"-> open file for both read and write. If file exists it is overwritten, if !exists then the file is created

  “a+"-> open file for reading or appending, if does not exist file is created.
- fopen will return NULL if it fails.

Example of fopen

```
FILE *f;
f = fopen("hello.txt", "r");
if(f == NULL){
   printf("Error cannot open hello.txt\n");
   perror("fopen");  // print why last called failed
   exit(1);          // passing the value of 1 to exit
                     // means that the reason for the
                     // sudden exit was because of
                     // an error.
}

// Read file using fscanf, fgetc or fread.

// close the file
fclose(fileHandle);
```

Standard Files

- There are three FILE's that are already opened when a program starts running:
- stdin - standard input, It is usually the terminal unless input is redirected:
  - bash$: hello (stdin is terminal)
  - bash$: hello < in1.txt (stdin is file in1.txt)
- stdout: standard output. It is usually the terminal unless redirected to output to a file:
  - bash$: hello (stdout is terminal)
  - bash$: hello >> out.txt (stdout is out.txt. Output is appended)
- stderr: Errors are redirected to stderr. It is usually the terminal unless redirected to a file.
  - bash$: hello > out 2>&1 (Redirect both stdout and stderr to out)

Basic Operations for stdin/stdout

- For stdout:
  - int printf(....) prints formatted output to stdout
  - int putchar(int c) writes c to stdout
- For stdin
  - int getchar() reads one character at a time
  - int scanf(.....) reads formatted input from stdin -

Basic Operations for generic FILE's

- int fgetc(fileHandle)
  - reads one char from fileHandle
  - c = getchar is equivalent to c = fgetc(stdin)
- int fputc(fileHandle, c)
  - puts/output char to stdout
  - putchar(c) is equivalent to fputc(stdout,c)
- int fscanf(fileHandle, format, &var....)
  - read formatted input from file handle,
  - scanf("%d", &j); is equivalent to fscanf(stdin, "%d", &i);

Basic Operations for generic FILE's

- int fprintf(fileHandle, format, ...);
  - prints formatted output to fileHandle
  - printf("Hello world %d\n, i); is equivalent to fprintf(stdout, "Hello world%d\n", i);
- sprintf(char *str, format, ...);
  - equivalent to printf/fprintf but the output is a string
  - "str" is passed as argument. str should have enough space to store the output.

Example: Read file with student grades and compute average.

students.txt:

```
Mickey 91
Donald 90
Daisy 92
```

students.c

```
#include<stdio.h>
#include<stdlib.h>   // perror, exit and others

#define MAX_STUDENTS 100
#define MAX_NAME     50
#define STUDENTS_FILE        "students.txt"

char names[MAX_STUDENTS][MAX_NAME + 1];
int grades[MAX_STUDENTS];
```

Example: Read file with student grades and compute average.

```
int main(int argc, char **argv){
      FILE *fd;
      char name[MAX_NAME + 1];
      int grade;
      int n;
      int i;
      double sum;

      fd = fopen(STUDENTS_FILE, "r");

      if(fd == NULL){
            printf("cannot read %s\n", STUDENT_FILE);
            perror("fopen");
            exit(1);
      }
```

Example: Read file with student grades and compute average.

```
    n = 0;
    while(fscanf(fd, "%s %d",
          names[n], &grades[n]){
          n++;
    }

    //computer average sum
    for(i = 0; i < n; i++){
          sum += grades[i];
    }
    printf("Average: %lf\n", sum / n);
    fclose(fd);
}
```

System Files

- They are lower-level than stream files that use FILE *.
- int fd = open(const char * pathname, int flags)
  - Opens a file in pathname.
  - flags: O_APPEND, O_RDONLY, O_RDWR
  - Open returns an int number called “file descriptor” that is used in further read and write operations to identify the file.

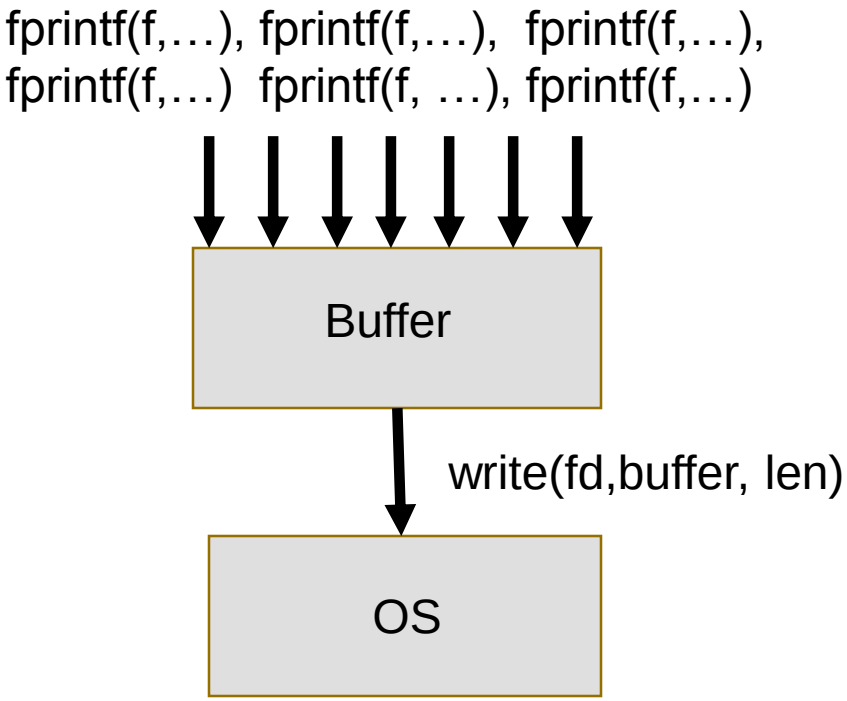
System Files

- int write(int fd, char * buffer, int len)
  - Writes len bytes from buffer into fd.
  - Returns number of bytes written or -1 if error.
- int read(int fd, char * buffer, int len)
  - Reads len bytes into buffer from fd.
  - Returns number of bytes read or -1 if error.
- close(int fd)
  - Closes file

Stream vs System Files

- The main difference between Stream Files (FILE * f) and Systems files (int fd) is that Stream Files are Buffered.
- fprintf(f, …) writes the characters into a buffer in memory.
- When the buffer is full then it calls write(fd,buffer, len)
- Every Stream file FILE * f has associated a system file descriptor int fd.
- By buffering, the number of System calls to the OS is reduced.

Stream vs System Files



Associating a Stream File with a File Descriptor

- Using write() is more difficult than using fprintf()
- To make formatting easier in your server, you may associate a Stream with a file descriptor.

```
FILE * fssock = fdopen( fd, "r+"); // fd is returned by accept()
                                   // in your server.
if ( fssock == NULL ) {
  perror("fdopen");
  exit(1);
}
fprintf( fssock, "ADD-USER %c%c",LF,LF);
fprintf( fssock, "%s\r\n", user);
fclose( fssock );  // It will close fd as well.
```

C++ Introduction

- C++ was designed by Bjarne Stroustrup of AT&T Bell Labs in the early 1980s
- Now C++ is widely accepted as an object-oriented low-level computer language.
- C++ is a superset of C, that is, code written in C can be compiled by a C++ compiler.

C++ and Java

- James Gosling created the Java Language
- He used C++ as base to design the Java syntax
- In this way it was easy for C++ programmers to learn Java.
- You will find Java and C++ to be very similar

Example of a C++ Program: Stack

- A C++ class is divided into two files:
  - The Header file
    - It ends with .h, or .hh
    - Example: Stack.h
  - An implementation file
    - It ends with .cc, .cpp, or .cxx
    - Example: Stack.cpp
- Programs that use the C++ class should include the header file
  - #include “Stack.h”

Stack.h

```
// Make sure that this file is included only once
#ifndef Stack_h
#define Stack_h

// Definition of a Stack class to store double values

class Stack {
 private:
   int maxStack;   // Max number of elements
   int top;        // Index to top of the stack
   double * stack; // Stack array
 public:
   // Constructor
   Stack(int maxStack);

   // Push a value into the stack.
   // Return false if max is reached.
   bool push(double value);
```

Stack.h (cont)

```
   // Pop a value from the stack.
   // Return false if stack is empty
   bool pop(double & value);  passing by references

   // Return number of values in the stack
   int getTop() const; the method wont modify the objects

   // Return max number of values in stack
   int getMaxStack() const;

   // Prints the stack contents
   void print() const;

  // Destructor
  ~Stack();
};

#endif
```

Stack.cpp

```
// Stack Implementation

// Used for cout << "string"
#include <iostream>

using namespace std;

#include "Stack.h"

// Constructor constructor里要把private的var设置了
Stack::Stack(int maxStack) {
   this->maxStack = maxStack;
   stack = new double[maxStack];
   top = 0;
}
```

Stack.cpp (cont.)

```
// Push a value into the stack.
// Return false if max is reached.
bool
Stack::push(double value) {
  if (top == maxStack) {
      return false;
  }

  stack[top]=value;
  top++;
  return true;
}
```

Stack.cpp (cont.)

```
// Pop a value from the stack.
// Return false if stack is empty
bool
Stack::pop(double & value) {
   if (top == 0) {
       return false;
   }

   top--;

   // Copy top of stack to variable value
   // passed by reference
   value = stack[top];
   return true;
}
```

Stack.cpp (cont.)

```
// Return number of values in the stack
int
Stack::getTop() const {
   return top;
}

// Return max number of values in stack
int
Stack::getMaxStack() const {
   return maxStack;
}
```

Stack.cpp (cont.)

```
// Prints the stack contents
void
Stack::print() const {
   cout << "Stack:" << endl;
   if (top==0) {
        cout << "Empty" << endl;
   }
   for (int i = 0; i < top; i++) {
        cout << i << ":" <<  stack[i] << endl;
   }
   cout << "------" << endl;
}
// Destructor
Stack::~Stack() {
  delete [] stack;
}
```

TestStack.cpp

```
// Example program to test Stack class
#include <iostream>
#include "Stack.h"

using namespace std;
```
pre-append cout automatically

```
int
main(int argc, char **argv) {
   Stack * stack = new Stack(10);
   stack->push(40);
   stack->push(50);
   stack->push(60);
   stack->push(70);
   stack->push(80);
```

TestStack.cpp (cont.)

```
   stack->print();

   double val;
   stack->pop(val);
   cout << "val=" << val << endl;

   stack->print();

   delete stack;
}
```

Makefile

```
all: TestStack

TestStack: TestStack.cpp Stack.cpp Stack.h
	g++ -g -o TestStack TestStack.cpp Stack.cpp

clean:
	rm -f TestStack core
```

Output

```
bash-4.1$ make
g++ -o TestStack TestStack.cpp Stack.cpp
.bash-4.1$ ./TestStack
Stack:
0:40
1:50
2:60
3:70
4:80
------
val=80
Stack:
0:40
1:50
2:60
3:70
------
```

Creating an Instance of an Object

- You can create an instance of an object dynamically by calling “new”. Example:

```
Stack * stack = new Stack();
```

- There is no garbage collector in C++, therefore you need to “delete” an object when it is no longer needed. Example:

```
delete stack;
```

- Not calling delete may cause your program to use more and more memory. This is called a “memory leak”.
- Be careful not to delete an object while it is still in use. This is called a “premature free”.

Objects as Local Variables

- You may also create an object as a local variable. Example:

```
void pushMany() {
  Stack stack(10);
  stack.push(78.9);
  stack.push(89.7);
  stack.print();
}
```

- The object will be deleted automatically when the function returns. No explicit call to “delete” is needed.
- The destructor will be called before returning.
- Try to define objects as local variables when possible.

TestStackWithLocalVars.cpp

```
// Example program to test Stack class
#include <iostream>
#include "Stack.h"

using namespace std;

int
main(int argc, char **argv) {
   Stack stack(10);
   stack.push(40);
   stack.push(50);
   stack.push(60);
   stack.push(70);
   stack.push(80);
```

TestStackWithLocalVars.cpp(cont.)

```
   stack.print();

   double val;
   stack.pop(val);
   cout << "val=" << val << endl;

   stack.print();
}
```

Objects as Global Variables

- Alternatively, you can define an object as a global variable.
- The variable will be created and the constructor called before main() starts.
- These constructors called before main starts are called “static constructors”.
- The destructor of the object defined as a global variable is called when the program exits.

Objects as Global Variables

```
// Example program to test Stack class
#include <iostream>
#include "Stack.h"

using namespace std;

// Create stack before main starts
Stack stack(10);

int
main(int argc, char **argv) {
        stack.push(40);
   stack.push(50);
   stack.push(60);
   stack.push(70);
   stack.push(80);
```

Constructors and Destructors

- When an object is created, either with new or as a local variable, the constructor is called.
- The constructor is a method with the same name as the class of the object that contains initialization code.
- The destructor is a method in the class that starts with "~" and the name of the class that is called when the object is deleted.

Constructors and Destructors

```
Stack.h
   class Stack {
    private:
      int maxStack;    // Max number of elements
      int top;         // Index to top of the stack
      double * stack; // Stack array
    public:
      // Constructor
      Stack(int maxStack);
      ~Stack();
   …
   }
```

Constructors and Destructors

Stack.cpp:

```
// Constructor
Stack::Stack(int maxStack) {
   this->maxStack = maxStack;
   stack = new double[maxStack];
   top = 0;
}

// Destructor
Stack::~Stack() {
  // delete the array
  delete [] stack;
}
```

Constructors and Destructors

```
int foo() {
  Stack * stack = new Stack(10);
   ...
   // Prevent memory leak
   delete stack;
}
```

C++ Reference Data Types

- It is used to create aliases of variables and to pass by references in functions.

```
 int i = 1;
int & r = i; // Now r is an alias for i
r++;         // Increments i
```

- References have pointer semantics.
- You can do the same in pure C as follows:

```
int i = 1;
int * r = &i;
(*r)++; // Increments i indirectly.
```

Passing by reference

- If a parameter type of a function is a reference and the parameter is modified, then it will modify the variable passed as parameter.

```
void swap(int & a, int & b) {
  int tmp = a;
  a = b;
  b = tmp;
}

main () {
  int x = 5;
  int y = 6;
  swap(x, y);
  // Now x ==6 and y ==5
}
```

- **Notice that no pointer operators * are not necessary so it looks simpler.**

Passing by Reference

- If we want a similar behavior using pure C we will need to write swap as follows:

```
void swap(int * pa, int * pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}

main () {
  int x = 5;
  int y = 6;
  swap(&x, &y);
  // Now x ==6 and y ==5
}
```

- References is a “syntax sugar” for pointers, that is, the code generated is the same in both cases.

Constants

- You can define variables as constant.
- If you try to modify them in your code, the compiler will generate an error.
- In this way the compiler will detect misuses of variables.
- Early detection of errors is important.
- Example:

```
const double pi = 3.14;
pi = 5; // Compiler Error!
```

Constant Parameters

- Also parameters to functions can be defined as constant.
- If the parameter is modified in the function then the compiler will generate an error.
- Example:

```
printInt(const Rect & rect) {
  rect.modify(2,3,10,9); // Compiler Error!
  . . .
}
```

Default Parameters

- You can set default parameters.
- The default parameters should be the last ones in a function declaration.
- Example:

```
void drawRect(int x, int y,
              int width=100, int height=100);

drawRect(45, 67); // Use defaults.
drawRect(78, 96, 45); // Use default height only.
```

- Only the declaration includes the default parameters and not the implementation.

Overloading Functions

- Also in C++ you can have multiple functions with the same name as long as the types of the arguments are different.
- When calling a function C++ will use the function that best matches the types.

Overloading Functions

- Example:

```
print(int a) { …}
print(const char * s) {… }
print(double d) {…}
…
int x = 9;
double y = 89.78;
const char * z =  “Hello world”;
print(x); // Uses print(int a);
print(y); // Uses print(double d)
print(z); // Uses print(const char * s)
```

Operator Overloading

- You can also define your own operators in C++.
- For example, if you have a class

```
class A {
  …
}
```

Then you can define an operator

```
A operator + (const A &s1, const A &s2) {
  …
}
```

And use it as:

```
A a;
A b;
// Initialize a and b
A c = a + b;
```

We will see more of this later.

Classes

- Remember that classes in C++ use two files.
- <class>.h (or .hh) has the class declaration with the names of the methods and variables.
- <class>.cpp (or .cc or .cxx) has the implementation.

private:/protected:/public:

- Variables inside a class can be defined private, protected or public.
- private:
  - It means that every method or variable after a ***private:*** declaration can be used only by the class that defines it.
- protected:
  - It means that every method or variable after a ***protected:*** declaration can be used only by the class or subclass that defines it.
- public:
  - It means that every method or variable after a ***public:*** declaration can be used by anybody.
- See Stack.h

friends

- In some cases you would like to allow some special classes to access the private method or variables of a class.
- For this you use friends.
- Example:

```
class ListNode {
  friend class List; // Class list can access
  char * val;        // anything in ListNode
  ListNode * next;
}
class List {
  …
}
```

Inline Functions

- You can define functions as inline.
- This will be a hint for the compiler to “inline” or expand the function instead of calling it.
- This may be faster in some cases.
- Example:

```
inline int min(int a, int b) {
  return (b<a)?b:a;
}
```

- The inline function may appear in a header file.

Allocating Arrays Dynamically

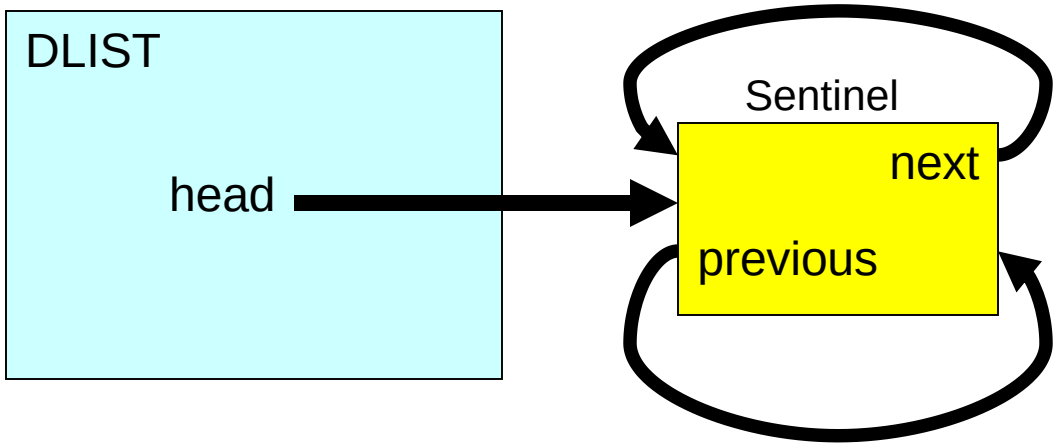
- To allocate arrays dynamically you can use new.

```
double array = new double[100];
```

- To delete an array call “delete []”

```
delete [] array;
```

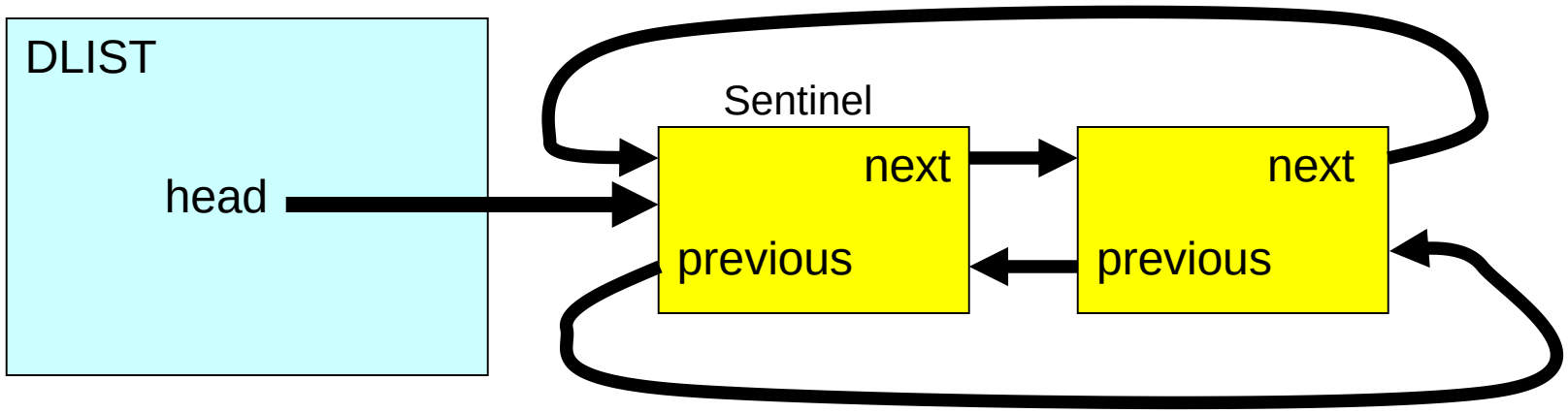
Implementing a Double Linked List



Sentinel – It does not store any data but simplifies implementation.

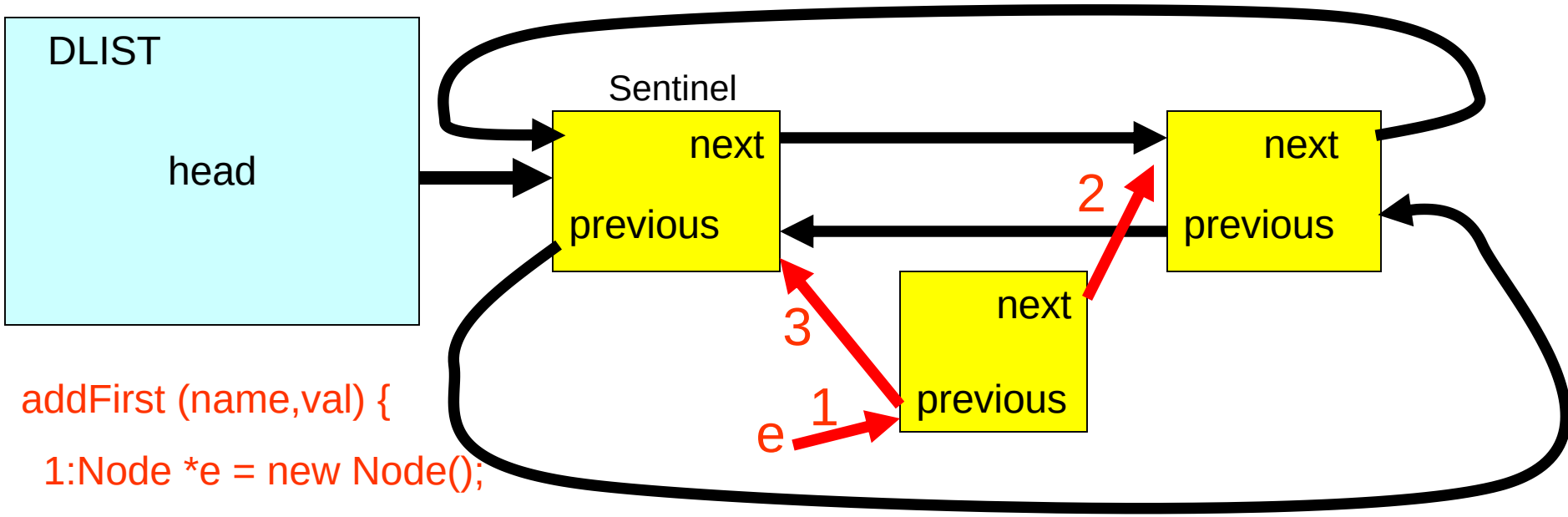
Empty List

Implementing a Double Linked List



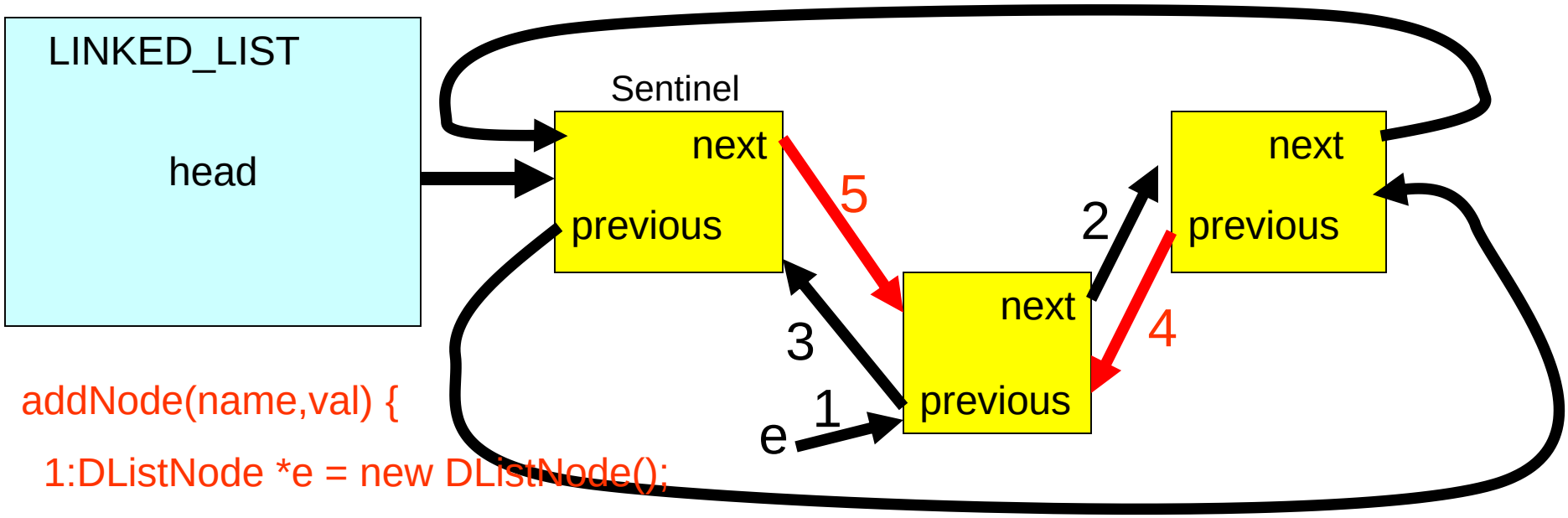
List with one element

Adding a Node at the Beginning of a Double Linked List




```
addFirst (name,val) {
  1:Node *e = new Node();
  e->name = strdup(name)
  2: e->next = head->next;
  3: e->previous = head;
```

Adding a Node at the Beginning of a Double Linked List




```
addNode(name,val) {
  1:DListNode *e = new DListNode();
  e->name = strdup(name)
  e->val = val
  2: e->next = head->next;
  3: e->previous = head;
  4:e->next->previous = e;
  5: head->next = e;
}
```

Implementing a double-linked list

```
dlist.h:
#if !defined DLIST_H
#define DLIST_H

struct DListNode{
        char * name;
        void * value;
        struct DListNode * next;
        struct DListNode * previous;
};

class DList {
        int nElements;
   DListNode * head;
 public:

   DList();
   void print(void);
   void addFront(char * name, void * value);
   void * lookup(char * name);
   int removeFront(char * name);
   ~DList();
};

#endif
```

Implementing a double-linked list

```
dlist.cpp:
#include "dllist.h"

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


//
// It returns a new LINKED_LIST. It allocates it dynamically,
// and initializaes the values. The initial list is empty.
//
DList::DList()
{

        // Create Sentinel node. This node does not store any data
        // but simplifies the list implementation.
        head = new DListNode();
        nElements = 0;
        head->next = head;
        head->previous = head;
}
```

Implementing a double-linked list

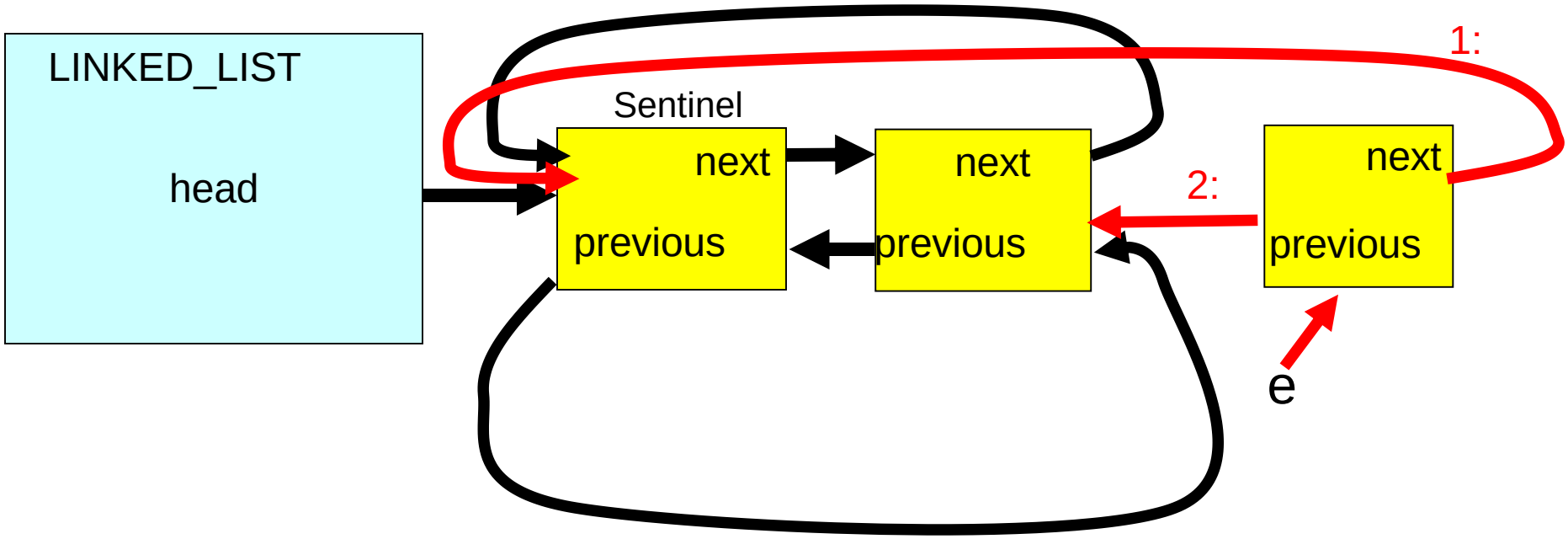
```
void
DList::addFront(char * name, void * value) {

        DListNode * e = new DListNode();
        e->name = strdup(name);
        e->value = value;

        //Add at the beginning
        e->next = head->next;
        e->previous = head;
        e->next->previous = e;
        e->previous->next = e;

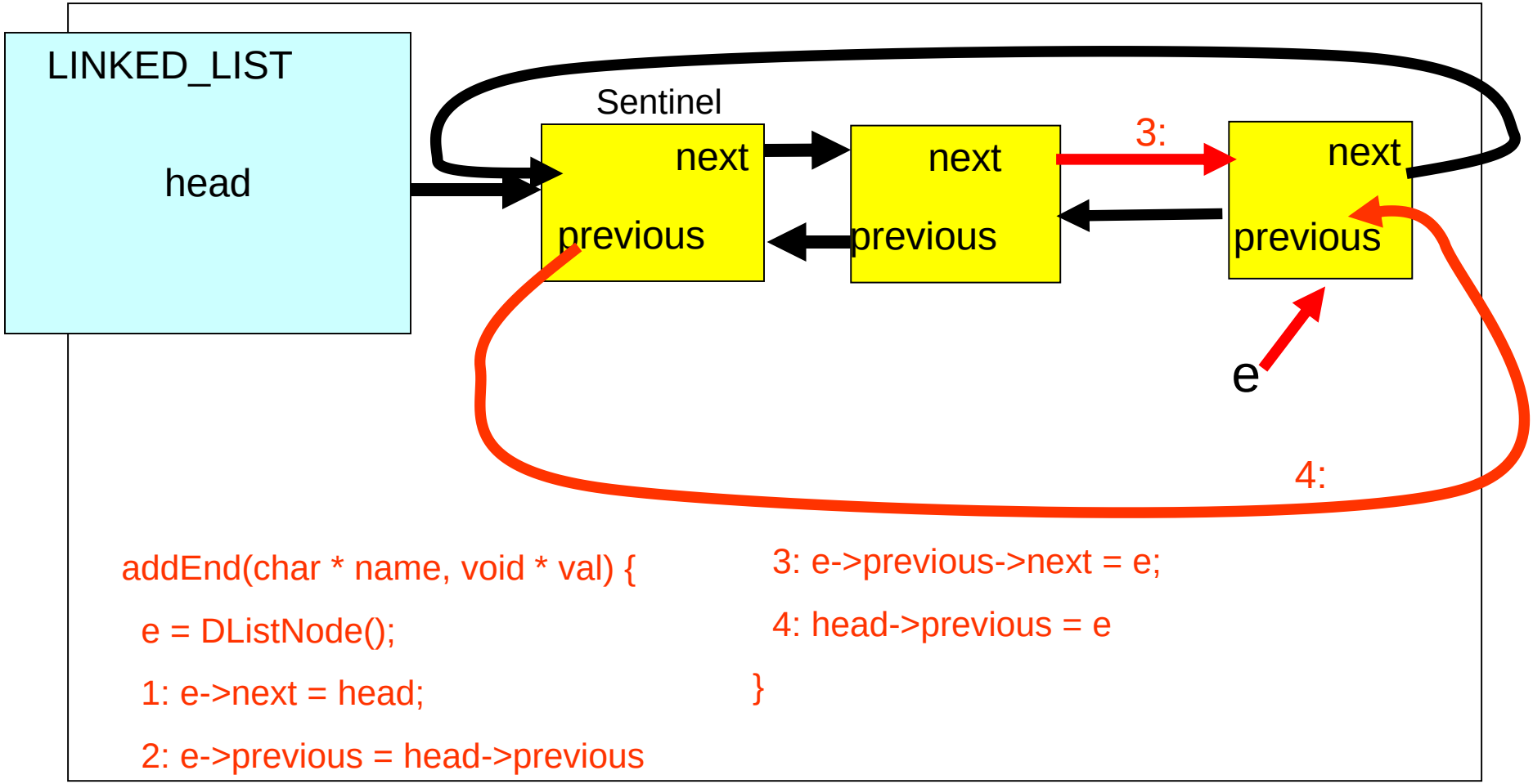
        nElements++;
}
```

Adding at the end of a double linked list




```
addEnd(char * name, void * val) {
  e = DListNode();
  1: e->next = head;
  2: e->previous = head->previous
}
```

Adding at the end of a double linked list




```
addEnd(char * name, void * val) {
  e = DListNode();
  1: e->next = head;
  2: e->previous = head->previous
  3: e->previous->next = e;
  4: head->previous = e
}
```

Implementing a double-linked list

```
void
DList::addEnd(char * name, void * value) {

        DListNode * e = new DListNode();
        e->name = strdup(name);
        e->value = value;
        e->next = head;
        e->prev = head->prev;
        head->prev->next = e;
        head->prev=e;


        nElements++;
}
```

Implementing a double-linked list

```
void *
DList::lookup(char * name)
{
   DListNode * e = head->next;
   while (e!= head) {
     if (!strcmp(e->name,name)) {
       return e->value;
     }
     e = e->next;
   }

   return NULL;
}
```

Implementing a double-linked list

```
//
// It removes the entry with that name from the list.
// Also the name and value strings will be freed.
//
bool
Dlist::remove(char * name) {
        return 0;
}
```

Parameterized Types

- In C++ we have three kind of types:
  - Concrete Type:
    - It is a user defined class that is tied to a unique implementation. Example: an int or a simple class.
  - Abstract Type:
    - It is user-defined class that is not tied to a particular implementation. Example Figure is an abstract class where draw can be Line::draw, Rectangle::draw(). It uses virtual methods and subclassing.
  - Parameterized type:
    - It is a type that takes as parameter another type. Example: Stack<int> creates a stack of type int, Stack<Figure *> will build a stack of entries of type Figure *. This is the base for “Templates”.

Templates

- They are parameterized types.
- They allow to implement data structures for different types using the same code, for example :
  - Stack<int> - Stack of type int
  - Stack<double>, Stack of type double
  - Stack<Figure>, Stack of type Figure.

Templates

- A generic class starts with the template definiton:

  template <typename T>
- typename T indicates that T is a type parameter.
- There can be also compile time constants or functions

  template <typename T, int SIZE>

Writing a Template

- Before writing a template it is recommended to write the code of the class without the parameters using a concrete type.
- For example, if you want to write a List template for any type, write first a List class for “int”s (ListInt).
- Once that you compile, test and debug ListInt, then write the template by substituting the “int” by “Data” (the parameter type).
- Also add template <typename Data> before every class, function, and struct.

A ListInt Class

```
ListInt.h
// Each list entry stores int
struct ListEntryInt {
   int _data;
   ListEntryInt * _next;
};
```

A ListInt Class

```
class ListInt {
 public:
  ListEntryInt * _head;

  ListInt();
  void insert(int data);
  bool remove(int &data);
};
```

A ListInt Class

```
ListInt::ListInt()
{
  _head = NULL;
}
```

A ListInt Class

```
void ListInt::insert(int data)
{
  ListEntryInt * e = new ListEntryInt;
  e->_data = data;
  e->_next = _head;
  _head = e;
}
```

A ListInt Class

```
bool ListInt::remove(int &data)
{
  if (_head==NULL) {
    return false;
  }

  ListEntryInt * e = _head;
  data = e->_data;
  _head = e->_next;
  delete e;
  return true;
}
```

A ListGeneric Template

- To implement the ListGeneric Template that can be used for any type we start with ListInt.
- Copy ListInt.h to ListGeneric.h.
- Add “template <typename Data> “ before any class, struct or function.
- Substitute “int” by ”Data”
- Where “ListEntryInt” is used, use “ListEntry<Data>” instead.
- Where “ListInt” is used, use “ListGeneric<Data>” instead.

A ListGeneric Template

ListGeneric.h

```
// Each list entry stores data
template <typename Data>
struct ListEntry {
   Data _data;
   ListEntry * _next;
};
```

A ListGeneric Template

```
template <typename Data>
class ListGeneric {
 public:
  ListEntry<Data> * _head;

  ListGeneric();
  void insert(Data data);
  bool remove(Data &data);
};
```

A ListGeneric Template

```
template <typename Data>
ListGeneric<Data>::ListGeneric()
{
  _head = NULL;
}
```

A ListGeneric Template

```
template <typename Data>
void ListGeneric<Data>::insert(Data data)
{
  ListEntry<Data> * e = new ListEntry<Data>;
  e->_data = data;
  e->_next = _head;
  _head = e;
}
```

A ListGeneric Template

```
template <typename Data>
bool ListGeneric<Data>::remove(Data &data)
{
  if (_head==NULL) {
    return false;
  }

  ListEntry<Data> * e = _head;
  data = e->_data;
  _head = e->_next;
  delete e;
  return true;
}
```

Using the Template

- To use the template include “ListGeneric.h”
  #include “ListGeneric .h”

- To instantiate the ListGeneric :

```
//List of int’s
ListGeneric<int> * listInt =
   new ListGeneric<int>();

//List of strings
ListGeneric<const char *> * listString =
        new ListGeneric<const char *>();

Or as local/global vars
ListGeneric<int> listInt; // List of int’s
ListGeneric<const char *> listString; // list of strings
```

A test for GenericList

```
#include <stdio.h>
#include <assert.h>
#include " ListGeneric.h"

int
main(int argc, char **argv)
{
  ////////////////////////////////////
  // testing lists for ints

  ListGeneric<int> * listInt = new ListGeneric<int>();

  listInt->insert(8);
  listInt->insert(9);

  int val;
  bool e;
  e = listInt->remove(val);
  assert(e); assert(val==9);

  e = listInt->remove(val);
  assert(e);
  assert(val==8);
```

Using the Template

```
//////////////////////////
  // testing lists for strings

  ListGeneric<const char *> * listString = new ListGeneric<const char *>();

  listString->insert("hello");
  listString->insert("world");

  const char * s;
  e = listString->remove(s);
  assert(e);
  assert(!strcmp(s,"world"));

  e = listString->remove(s);
  assert(e);
  assert(!strcmp(s,"hello"));
}
```

Iterator Template

- An iterator is a class that allows us to iterate over a data structure.
- It keeps the state of the position of the current element in the iteration.

```
template <typename Data>
class ListGenericIterator {
  ListEntry<Data> *_currentEntry; // Points to the
   current node
  ListGeneric<Data> * _list;
 public:
  ListGenericIterator(ListGeneric<Data> * list);
  bool next(Data & data);
};
```

Iterator Template

```
template <typename Data>
ListGenericIterator<Data>::ListGenericIterator(ListGeneric<Data> * list)
{
  _list = list;
  _currentEntry = _list->_head;
}

template <typename Data>
bool ListGenericIterator<Data>::next(Data & data)
{
  if (_currentEntry == NULL) {
    return false;
  }

  data = _currentEntry->_data;
  _currentEntry = _currentEntry->_next;

  return true;
}
```

Iterator Template

```
void testIterator() {
  ListGeneric<const char *> * listString = new ListGeneric<const char *>();
  const char * (array[]) = {"one","two","three","four","five","six"};
  int n = sizeof(array)/sizeof(const char*);

  int i;
  for (i=0;i<n;i++) {
    listString->insert(array[i]);
  }

  const char * s;
  ListGenericIterator<const char *> iterator(listString);
  while (iterator.next(s)) {
    printf(">>%s\n",s);
    i--;
    assert(!strcmp(s,array[i]));
  }

  printf("Tests passed!\n");
}
```

Default Template Parameters

- You can provide default values to templates. Example:

Stack.h

```
 template <typename T = int, int n = 20>
class Stack {
   T array[n];
    …
};
```

- At instantiation time:

```
Stack stack1; // Stack of type int of size 20 (default)
Stack<double> stack2; // Stack of type double of size 20
Stack<Figure, 100> stack3; // Stack f type Figure of size 100
```

Function Templates

- Also functions can be parameterized.

```
template <typename T>
void swap(T &a, T &b) {
   T tmp = a;
   a = b;
   b = tmp;
}
```

…

```
int x = 3; int y = 4;
swap(x,y); // Swaps int vars x, y
double z1 = 3.567; double z2 = 56;
swap(z1, z2); // Swaps double vars z1, z2
```

- The compiler will generate instances of the swap function for double and int.

C++ Input/Output Library

- Three tyes of I/O
  - Interactive I/O
  - File I/O
  - Stream I/O
- Use istream for input and ostream for output.
- We have the following predefined streams:

```
istream cin;     // Standard input
ostream cout;  // Standard output
```

Output Operations

- To output a variable use the << operator.
  cout << 7; // Prints a 7 in standard output.

- Also you can use modifiers like:

flush – Flush output buffer
endl - Sends end-of-line and the flush.
dec - Display ints in base 10
hex - Displays ints in hex format
oct - Displays ints in octal format.
setw(i) – Specify field width.
left - Specify left justification
right – Specify right justification
setprecision(i) – Set total number of digits in a real value.

Examples

```
cout << setw(6) << left << 45;
  // Prints “45bbbb” where b is space char
cout << setw(6) << right << 45;
  // Print “bbbb45”
cout << setprecision(3) << 45.68;
  // Prints 46.7
```

Input Operations

- By default input operator >> skips whitespace chars.

```
int i;
double f;
cin >> i; // Read an int value i
cin >> f; // Read double f
```

- To check error conditions

```
If (!cin) {
   // operation failed
}
```

- You can also read a whole line

```
char line[200];
cin.getline(line, 200);
```

Input state

- To check the state of the input you can use:

  cin.eof() – EOF

  cin.bad() – In error state

  cin.fail() – Return true if last operation failed.

File Streams

- To write to a file you use <fstream>

  ifstream – input stream

  ofstream – output stream

For example:

```
ifstream myinput("file.txt"); // Open file for reading.
int i;
myinput >> i;  // Read integer i.
if (! myinput) {
  // Error
}
myinput.close(); // Close file.
```

File Streams

```
#include <fstream>
...
// Open file for writing.
ofstream myoutput("file.txt");
int i;
myoutput << i;  // Write integer i.
if (! myoutput) {
   // Error
}
myoutput.close(); // Close file.
```

Standard Template Library (STL)

- Created by Alex Stepanov in 1997.
- Intended to provide standard tool for programmers for vectors, lists, sorting, strings, numerical algorithms etc.
- It is part of the ANSI/ISO C++ standard.
- It has three components:
  - Containers: Contain collections of values.
  - Algorithms: Perform operations on the containers.
  - Iterators: Iterate over the containers.

Containers

- Containers store elements of the same type.
- There exist two kind of containers: Sequential and associative.
  - Sequential Containers:
    - used to represent sequences

    vector<ElementType> - vectors

    deque<ElementType> - queues with operations at either end.

    list<ElementType> - Lists

Containers

- Associative Containers:
  - Used to represent sorting collections.
  - They associate a key to a data value.

set<KeyType> - Sets with unique keys.

map<KeyType,ElementType> - Maps with unique keys.

multiset<KeyType> - Sets with duplicate keys.

multimaps<KeyType, ElementType> - Maps with duplicate keys.

Iterators

- Iterators are used to refer to a value in a container.
- Every container provides its own iterator. Example:

```
vector<int> v;

. . .

for (vector<int>::iterator it = v.begin();
                                it!=v.end(); ++it) {
  cout << *it << endl;
}
```

Iterators

- An iterator supports at least the following operations:

  ***iter** refers to the value the iterator points to.

  ++iter increments iter to point to the next value in the container.

  iter1 == iter2 Used to compare two iterators.

- Also a container like vector<int> v; provides the functions:

  v.begin() – Returns an iterator that points to the beginning of the container.

  v.end() – Returns an iterator that points to the end of the container.

Vectors

- Use

  #include <vector>
- Represents a resizable array.

  vector<int> v; // Vector of ints.

  vector<Figure *> figures; // Vector of pointer to Figures.
- ***v.capacity()*** represents the maximum number of elements before resizing.
- ***v.size()*** represents the number of elements used.
- ***v.push_back(e)*** adds an element to the end.
- ***e=v.pop_back()*** Remove last element.
- ***e=v[i] or e=v.at(i)*** Get a reference to the ith element

Vectors and Iterators

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;
main()
 {
      vector<string> table;
      table.push_back("Hello");
      table.push_back("World");
      table.push_back("cs390cpp");

      // Iterating with i
      for (int i = 0; i < table.size(); i++) {
        cout << table[i] << endl;
      }
```

Vectors and Iterators

```
    // Iterating with an iterator
    vector<string>::iterator it;

    for (it=table.begin(); it<table.end(); it++) {
      cout << *it << endl;
    }

    // Reverse iterator
    vector<string>::reverse_iterator rit;
    for (rit=table.rbegin(); rit<table.rend(); rit++)
    {
      cout << *rit << endl;
    }
}
```

Lists

- Use

  #include <list>
- Represents a list.

  list<int> l; // List of ints.

  list<Figure *> figures; // List of pointer to Figures.
- Optimized for insertions and deletions. No random access operators such as [] or at() are provided.
- ***l.size()*** represents the number of elements used.
- ***l.push_back(e)*** adds an element to the end.
- ***e=v.pop_back()*** Remove last element.
- ***l.sort()*** sorts list.

Maps

- They are templates that represent a table that relate a key to its data.
- Std::map <*key_type*, *data_type*, [*comparison_function*]>
- Example:

```
map <string, int> grades;
grades[“Peter”] = 99;
grades[“Mary”] = 100;
grades[“John”] = 98;
cout<<“Mary’s grade is "<<grades[“Mary”] << endl;
grades[“Peter”] = 100; // Change grade
```

Maps

- Checking if an element is not in the map

```
if(grades.find("Luke") == grades.end()) {
   cout<<"Luke is not in the grades list." << endl;
}
else {
   cout<<"Luke's grade is "<< grades["Luke"] << endl;
}
```

- Iterating over a map

```
for (map<string,int>::iterator it = grades.begin();
     it != grades.end(); ++it) {
    cout << "Name: " << it.first;
    cout << "Grade: " << it.second << endl;

}
```

- Erasing a key

grades.erase("Peter");

STL Strings

- The STL strings provide Java like string manipulation

```
#include<string>
…
string str1 = "Hello";
string str2 = "world."
string str3 = str1 + " " + str2;
cout << str3 << endl;
```

- You do not need to allocate or deallocate memory. Everything is done by the methods themselves.
- You can create a STL string from a C string: `string s("c string");`

Some STL Strings Functions

- str.length()
  - Length of string
- str[i] or str.at(i)
  - Get ith character in the string.
- size_t str1.find (const string& str2, size_t pos = 0)
  - Find the position of a str2 in str1. -1 if not found.
- str.substr (size_t pos = 0, size_t n = npos)
  - Returns a substring starting at pos with up to n chars.
- You can use str.c_str() to get the corresponding const char * string.

Copy Constructor

- A Copy constructor is a constructor that has as argument a reference to an object of the same type:

```
Stack::Stack(Stack &s)
```

- Copy constructors are called when:

1. Initializing an object from another

```
Stack s1(50);
s1.push(78);
Stack s2 = s1;
// Now s1 and s2 are two
// different objects where s1 and
// s2 have the same contents.
```

Copy Constructor (cont)

2. A parameter of a class type is passed as a value

```
Stack s1(50);
s1.push(78);
foo(s1);
…
void foo(Stack s) {
   // s and s1 are two different
   // objects with the same contents.
```

Copy Constructor (cont)

3. When a value of a class type is returned in a function.

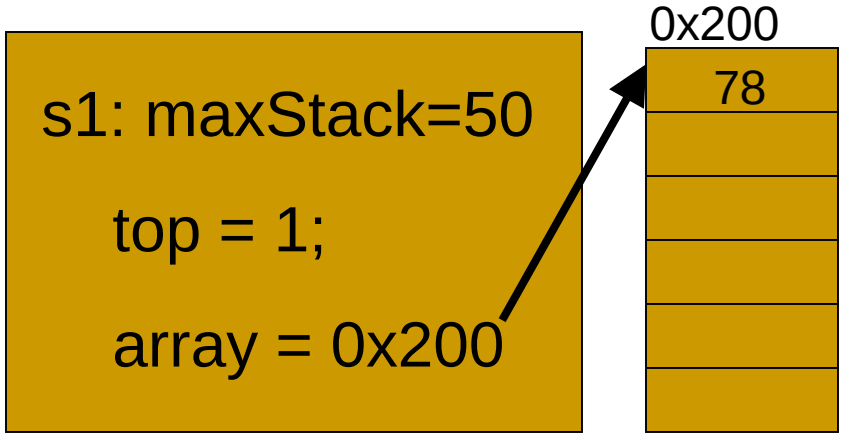
```
Stack s2 = getStack();
…
Stack getStack() {
  Stack s1(50);
  s1.push(78);
  return s1;
} // The content of s1 is copied into s2
```

Copy Constructor (cont.)

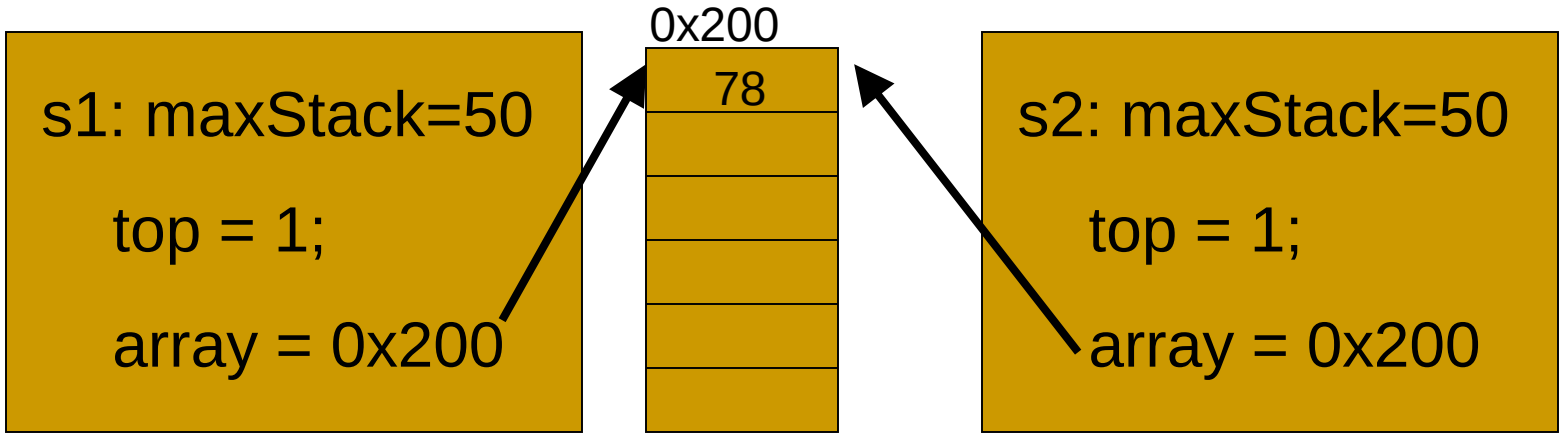
- By default the compiler will generate a copy constructor that will allocate a new instance and copy element member from the old to the new object (shallow copy).
- This is OK for simple classes but not for the classes that manage their own resources.
- Assume:

```
Stack s1(50);
 s1.push(78);
 Stack s2 = s1;
```

Copy Constructor (cont.)



Stack s2 = s1;



Copy Constructor (cont.)

- This is wrong because s2 should have its own copy of the stack array.
- This is solved by defining in the Stack class a “copy constructor” that will make a “deep copy” of the old object.

Copy Constructor (cont.)

Stack.h

```
class Stack {
 …
 public:
    Stack();
    Stack(Stack &s); // Copy constructor
   …
};
```

Copy Constructor (cont.)

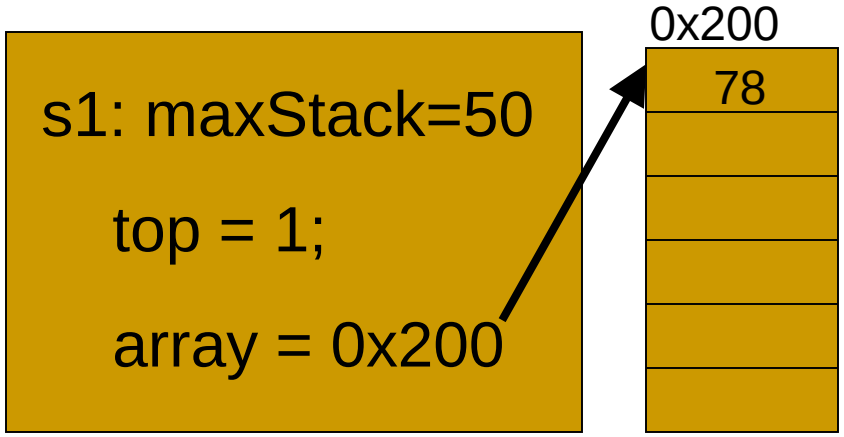
Stack.cpp

```
Stack::Stack(Stack &s) {
  top = s.top;
  maxStack = s.maxStack;

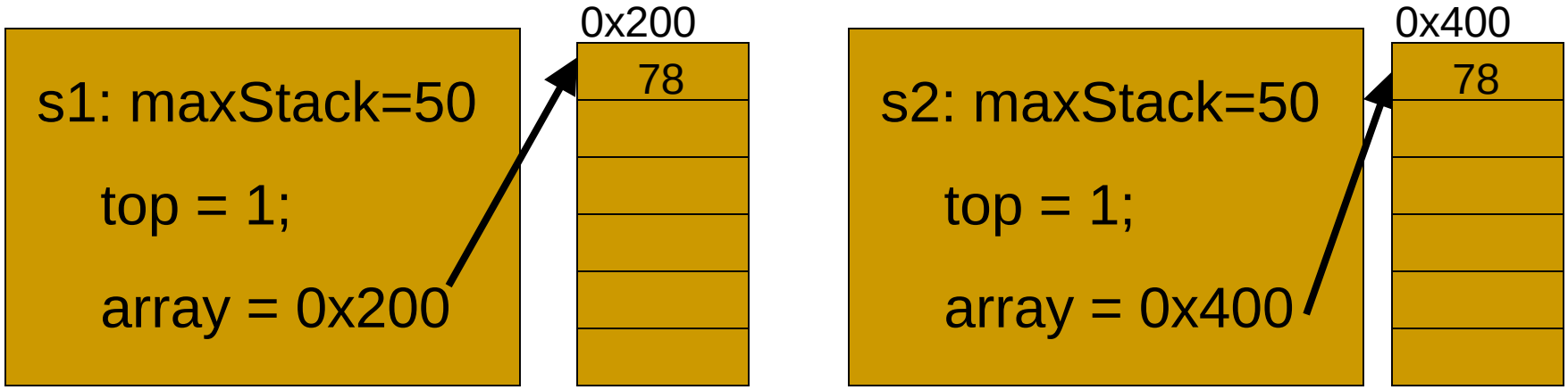
  // Create a new stack array
  stack = new double[maxStack];

  // Copy the entries in the array.
  for (int i = 0; i < top; i++) {
    stack[i] = s.stack[i];
  }
}
```

Copy Constructor (cont.)



Stack s2; s2 = s1;



A a; vs A a( );

- Assume the constructor of a class A() does not have arguments then
- In the defintions:

A a(); // a is the declaration of a

// function that returns A

and

A a; // Is an object variable of type A.

Assignment Operator with Classes

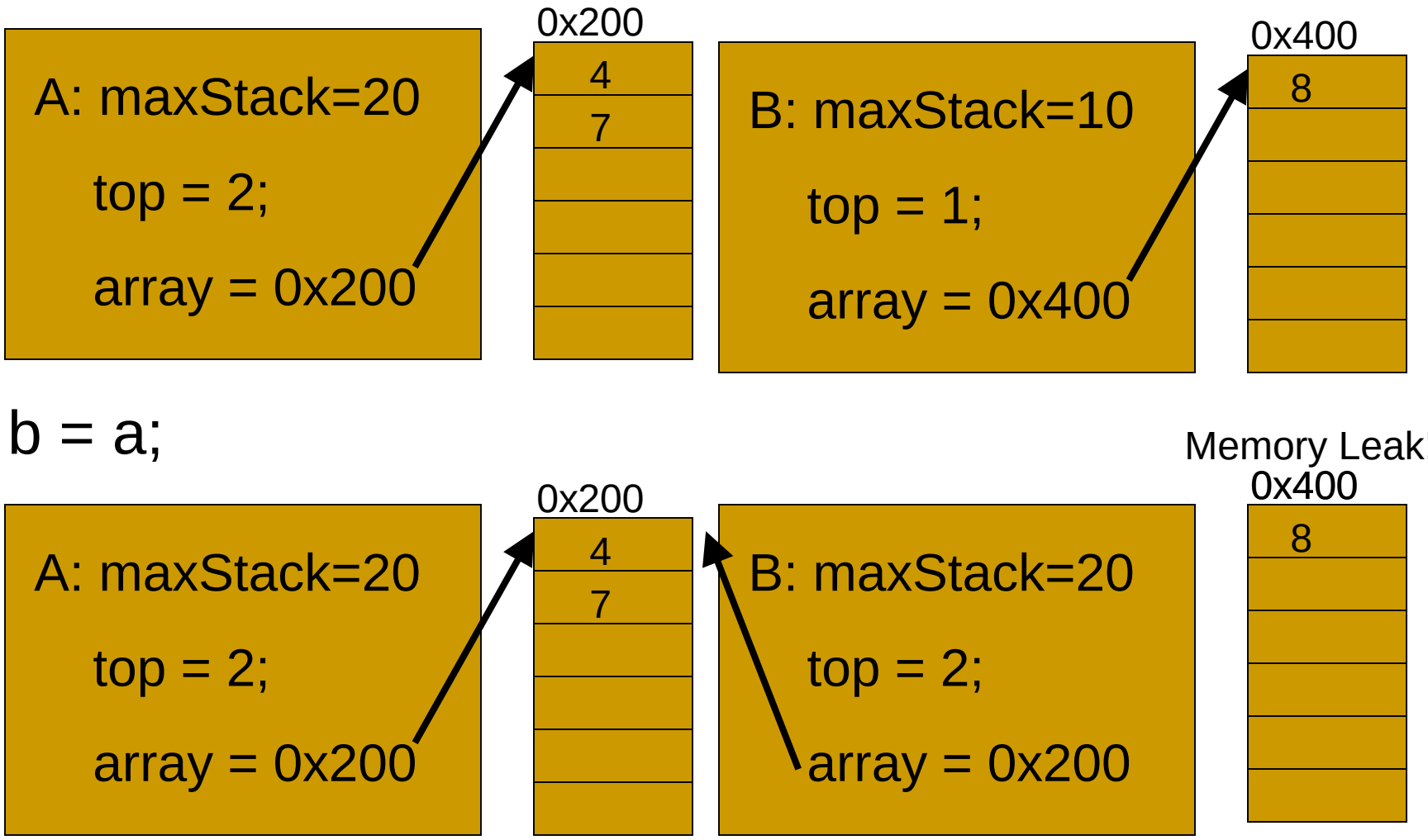
- By default assigning an object to another will copy the elements of one object to another (shallow copy)

```
Stack a(20); // Create stack a as a local var
a.push(4);
a.push(7);
Stack b(10); // Create stack b as a local var
b.push(8);
b = a; // Assign content of a to b
```

Assignment Operator with Classes

- This is wrong because the content of ***b*** will be exactly the same as ***a.***
- The member variable “stack” will point to the same array in both a and b.
- The array pointed by **b** will be overwritten and leaked.
- To fix this problem if you need to assign objects to objects you have to define your own assignment operator.

Assignment Operator with Classes



Assignment Operator with Classes

- The default assignment operator that implements a shallow copy is fine for most classes.
- However, it is not fine for classes that manage their own resources.
- For those classes we need to define an assignment operator.

Assignment Operator with Classes

Stack.h

```
class Stack {
 …
 public:
    Stack();

    // Copy constructor
    Stack(Stack &s);

    // Assignment Operator
    Stack & operator=(const Stack & s);
  …
};
```

Assignment Operator with Classes

Stack.cpp

```
// Assignment Operator
Stack &
Stack::operator=(const Stack & s){
 // Check for self assignment. E.g. a=a;
 if (this == &s) {
  return *this;
 }
 // deallocate old array
 delete [] stack;
```

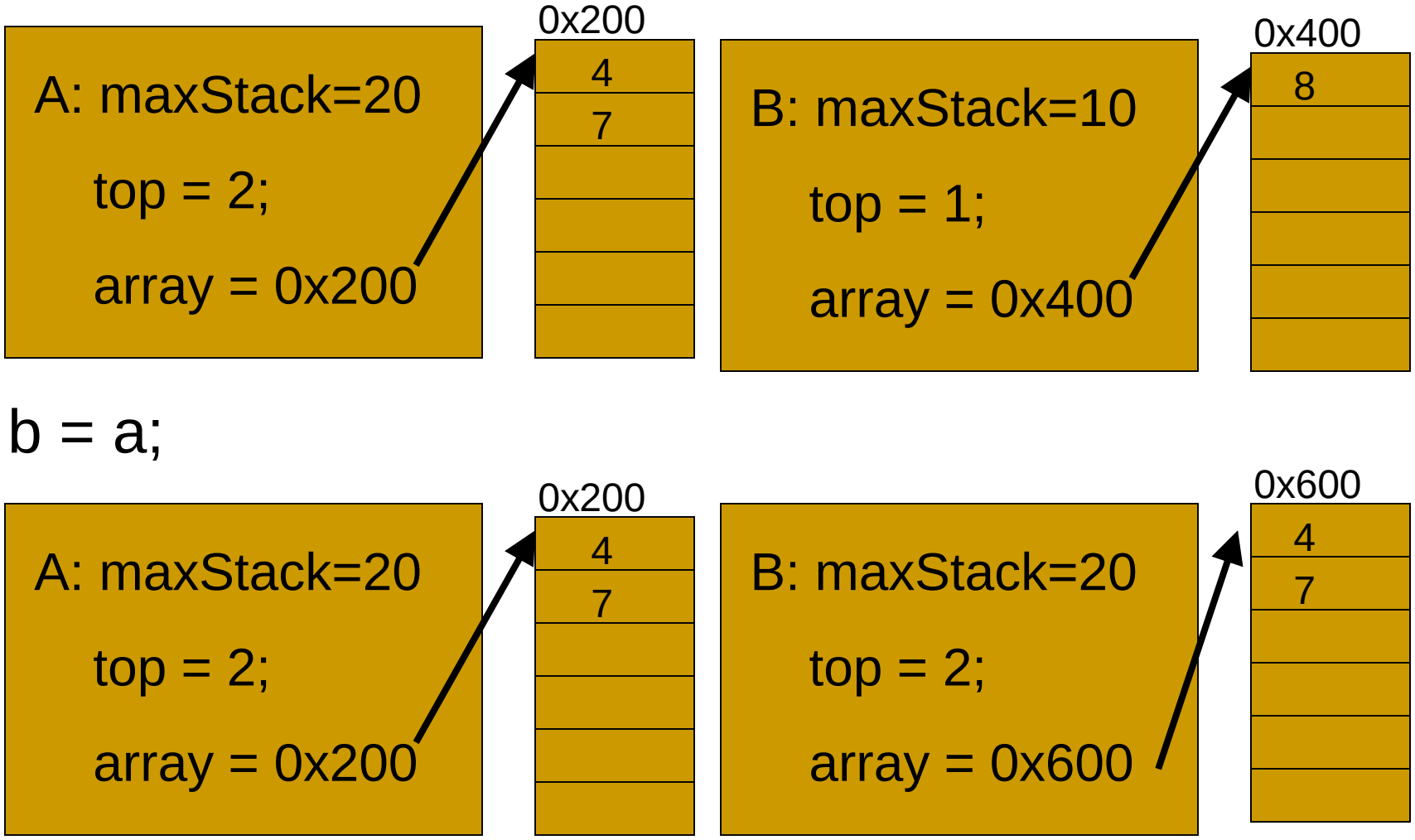
Assignment Operator with Classes

```
    // Copy members
    top = s.top;
     maxStack = s.maxStack;

     // Create a new stack array
     stack = new double[maxStack];

     // Copy the entries in the array.
     for (int i = 0; i < top; i++) {
       stack[i] = s.stack[i];
     }
     return *this;
}
```

Assignment Operator with Classes



Exception Handling

- In C++ there are exceptions like in Java
- The exceptions thrown derive from the class “exception”.
- Other standard exceptions are defined in the include <stdexcept>.
- Exception types are:
  - exception, logic_error, invalid_argument, runtime_error, range_error, overflow_error, underflow_error, etc.

Exception Handling

```
Stack.h:

#include <stdexcept>

class Stack {

    double pop()
  throw(logic_error);

    ….

}
```

Exception Handling

```
Stack.cpp:
#include "Stack.h"
double Stack::pop() throw(logic_error)
{
  if (top==0) {
      throw logic_error("Stack Empty");
  }
  top--;
  return stack[top];
}
```

Exception Handling

```
int main() {
  Stack * stack = new Stack();
  try {
    stack->push();
  }
  catch (logic_error &e) {
    cout << e.what() << endl;
  }
}
```

Namespaces

- Namespaces are used to help encapsulation and modularity.
- They also are used to avoid name conflicts.
- In Java namespaces are called packages.
- The syntax is similar to the classes but without the public/protected/private.

Namespace Definition

Figure.h

```
namespace MyDraw {
  class Figure {
    …..
  };
}
```

Line.h

```
namespace MyDraw {
  class Line {
    …..
  };
}
```

Namespace Usage

- To refer to the classes in the namespace from outside the namespace you need to pre-append the namespace.
- Example:

MyClass.h

```
#include  "Figure.h"
class MyClass {
  MyDraw::Figure * figure;// Need to
                          //pre-append MyDraw
…
};
```

Namespace Usage

- If you want to avoid in your code pre-pending the namespace every time it is used, you can use “using namespace”.

```
MyClass.h
#include  “Figure.h”
using namespace MyDraw;
class MyClass {
   Figure * figure; // No need to
                    // pre-append MyDraw
…
};
```

Standard Namespaces

- The standard library that defines strings, streams etc. are in the name space “std”.
- To refer to the elements of “std” you need to pre-append std::.

```
std::cout << “Hello world”;
```

- If you don’t want to pre-append “std” every time you use the standard library use the “using” statement.

```
using namespace std;
…
cout << “Hello world”;
```

Public Inheritance

- It is used when you want a derived class to inherit all the public interface of the parent class.
- Assume the following Figure parent class.

Figure.h

```
class Figure {
 public:
    enum FigureType { Line, Rectangle,
                              Circle, Text };

    Figure(FigureType figureType);

    FigureType getFigureType();
    void select(bool selected);
     bool isSelected();
    …
};
```

Public Inheritance

- Now assume the following Line class that is subclass of Figure. Notice the keywords “public Figure”.

  Line.h

```
class Line : public Figure {
    int x0, y0, x1, y1;
  public:
    Line(int x0, int y0,
                int x1, int y1);
    int getX0();
    …
}
```

- Line will inherit all the public methods of Figure, such as the select() and isSelected() methods.

Creating Objects of a Subclass

- The constructor of a subclass needs to take care first of constructing the attributes of the base class.
- In C++ there is no "super" like in Java, so the initialization of the base class is the same as the initialization of members.

```
 // Constructor/destructor for a line
Line::Line(int x0, int y0, int x1, int y1)
:Figure(Figure::FigureType::Line)
{
  controlPoints.push_back(
        new ControlPoint(this, x0,y0));
  controlPoints.push_back(
        new ControlPoint(this, x1,y1));
}
```

Dynamic Cast

- You can assign a pointer to Line to a variable that is a pointer to Figure.

```
Figure * figure = new Line(x0, y0, x1, y1);
```

- You can use ***figure*** to call any public method of Figure in the instance of the Line object.

```
bool selected = figure->isSelected();
```

- However you cannot call a method that only belongs to Line.

```
int x0 = figure->getX0();
       // Compiler error!! getX0() is
       // not a method of Figure.
```

Dynamic Cast

- You can use a dynamic cast to go from a pointer to a base class to a pointer to a subclass.

```
Line * line =
    dynamic_cast<Line>(figure);
 if (line != NULL) {
    // Yeah! figure was a line.
    int x0 = line->getX0();
 …
 }
```

- The dynamic cast will return NULL if the object ***figure*** points to is not a ***Line***.

Virtual Methods

- A virtual method is a method that can be overwritten by a subclass.

Figure.h

```
class Figure {
 public:
    …
    Figure(FigureType figureType);

    FigureType getFigureType();

    virtual void draw(CDC * pDC);
    …
};
```

Virtual Methods

- The subclass may overwrite the virtual method or may keep the definition of the parent class.

Line.h

```
class Line : public Figure {
  public:
    Line(int x0, int y0,
                 int x1, int y1);
    void draw(CDC * pDC);
    …
};
```

- In this case Line overwrites the definition of the draw class.

Virtual Methods

- Other subclasses may also overwrite the draw method for other geometric shapes.

Rectangle.h

```
class Rectangle : public Figure {
  public:
    Rectangle(int x0, int y0,
              int x1, int y1);
    void draw(CDC * pDC);
    …
};
```

Calling Virtual Functions

- Another class like ***Drawing*** may have a vector of pointer to ***Figure*** to represent a collection of shapes.
- To draw all the figures in the vector it just needs to iterate over all the pointers and call draw().

```
void
Drawing::draw(CDC* pDC)
{
   // For each figure in vector "figures" draw the figure with
   control points.
   for (unsigned i = 0; i < this->figures.size(); i++) {
      Figure * f = figures.at(i);
      f->draw(pDC);
  }
}
```

- The draw() method called will be the one redefined by the subclasses and not the one defined by Figure.

Abstract Classes

- An abstract class is one that can serve as a base class but that cannot be directly instantiated.
- It can be instantiated only through a subclass.
- An abstract class may declare methods without implementing them.

Abstract Classes

- Example

Figure.h

```
class Figure {
 public:
    …
    Figure(FigureType figureType);

    FigureType getFigureType();

    virtual void draw(CDC * pDC) = 0;
    …
};
```

- The “=0” means that the base class Figure does not implement this method but subclasses should implement it.
- Trying to create an object of type Figure will give a compiler error.

```
Figure * f = new Figure(Figure::FigureType::Line);
                                        // Compiler error
Figure * l = new Line(x0, y0, x1, y1); // OK
```

Virtual Destructors

- When calling delete on a subclass, the destructor of the subclass is called before the destructor of the base class.

A.h

```
class A {
  X * x;
public:
  A();
  ~A();
};
```

A.cpp

```
  A() {
    x = new X;
  }
  ~A() {
    delete x;
  }
```

Virtual Destructors

B.h

```
class B : public A {
  Y * y;
public:
  B();
  ~B();
};
```

B.cpp

```
B::B() {
  y = new Y;
}
B::~B() {
  delete y;
}
```

Virtual Destructors

```
B * b = new B();
…
delete b; // It will call ~B() and then ~A()

A * a = new A();
..
delete a; // It will call ~A()
```

However,

```
A* a = new B();
..
delete a; // It will call ~A() only!!!
```

Virtual Destructors

- To make sure that ~B destructor is called, you need to define the destructor in A as virtual.

  A.h

```
class A {
  X * x;
public:
  A();
  virtual ~A();
};
```

- Now

```
A* a = new B();
..
delete a; // It will call ~B() and ~A() that is what we want.
```

Private Inheritance

- We have seen public inheritance where all the public methods of the parent class are public in the subclass.
- In private inheritance, the public methods of the parent class are private in the subclass.

```
class A {
 public:
    void xx();
    void yy();
};
class B : private A {
  public:
    using A::xx(); // Makes xx() public.
                   // Only xx() is accessible in B.
                   // yy() is private.
};
```

Protected Inheritance

- In protected inheritance, the public methods of the parent class are protected in the subclass.

```
class A {
 public:
    void xx();
    void yy();
};
class B : protected A {
  public:
    using A::xx(); // Makes xx() made public.
                   // Only xx() is public in B.
                  // yy() is protected so it
                  //can be inherited in a subclass.
};
```

Multiple Inheritance

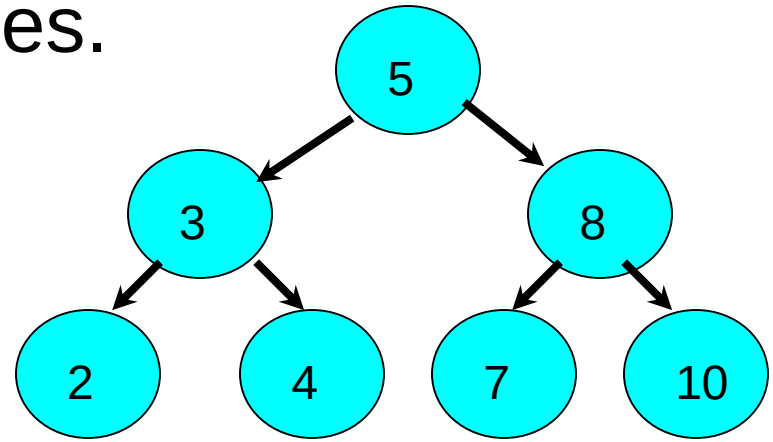
- In C++ you have multiple inheritance, that is a subclass can inherit from two or more parent classes.

```
class A {
 public:
   void xx();
 }
 class B {
 public:
   void yy();
 }
 class C: public A, public B {
  }; // C inherits from both A and B so xx() and yy() are
 public.
```

- Multiple inheritance is discouraged since adds extra complexity that is not needed.
- Java uses single inheritance but a class may implement multiple interfaces.

Binary Trees

- A Tree is a data structure that generalizes a List
- A node may have more than one child node
- A Binary Tree has at most two children nodes.



Binary Tree Implementation

```
#include <stdio.h>

class TreeNode {
public:
    int _value;
    TreeNode * _left;
    TreeNode * _right;
};

class Tree {
    TreeNode * _root;
public:
    Tree();

    // Insert number in tree
    bool insert(int val);

    // Return true if val is in tree
    bool find(int val);

    // Print tree
    void print();

    // Auxilia method to print tree
    void printAuxiliar(TreeNode * node);
};
```

Binary Tree Implementation

```
Tree::Tree()
{
    _root = NULL;
}
```

Binary Tree Implementation

```
// Insert number in tree. Use prev.
bool
Tree::insert(int val)
{
        TreeNode * n = _root;
        TreeNode * prev = NULL;
        while (n != NULL) {
                if (val < n->_value) {
                        prev = n;
                        n = n->_left;
                }
                else if (val > n->_value) {
                        prev = n;
                        n = n->_right;
                }
                else {
                  // value already exists. Return
                  return false;
                }
        }
```

Binary Tree Implementation

```
        // Create new node
        n = new TreeNode();
        n->_value = val;
        n->_left = NULL;
        n->_right = NULL;

        // Attach new node
        if (prev==NULL) {
                // Tree was empty
                _root = n;
        }
        else {
                // tree not empty. Check what child
                // to insert it to.
                if (val < prev->_value) {
                        prev->_left = n;
                }
                else {
                        prev->_right = n;
                }
        }
        return true;
}
```

Binary Tree Implementation

```
// Same Insert number in tree. Double pointer version. More compact.
bool
Tree::insert(int val)
{
        TreeNode ** n = &_root;
        while (*n != NULL) {
                if (val < (*n)->_value) {
                        n = &(*n)->_left;
                }
                else if (val > (*n)->_value) {
                        n = &(*n)->_right;
                }
                        else {
                          return false;
                        }
        }
        *n = new TreeNode();
        (*n)->_value = val;
        (*n)->_left = NULL;
        (*n)->_right = NULL;
        return true;
}
```

Binary Tree Implementation

```
bool
Tree::find(int val)
{
        TreeNode * n = _root;
        while (n != NULL) {
                if (val < n->_value) {
                        n = n->_left;
                }
                else if (val > n->_value) {
                        n = n->_right;
                }
                 else {
                   return true;
                 }
        }

       return false;
}
```

Binary Tree Implementation

```
// Print tree
void
Tree::print()
{
    printAuxiliar(_root);
}

void
Tree::printAuxiliar(TreeNode * node)
{
    if (node==NULL) return;
    printAuxiliar(node->_left);
    printf("%d\n", node->_value);
    printAuxiliar(node->_right);
}
```

Binary Tree Implementation

```
int
main(int argc, char **argv)
{
        Tree tree;
        tree.insert(5);
        tree.insert(9);
        tree.insert(3);
        tree.insert(10);
        tree.insert(11);

        tree.print();

        bool found = tree.find(3);
        if (found) {
                printf("found 3\n");
        }
        else {
                printf("did not find 3\n");
        }

        found = tree.find(7);
        if (found) {
                printf("found 7\n");
        }
        else {
                printf("did not find 7\n");
        }
}
```

Binary Tree Implementation

Output:

```
cs240@data ~/2014Fall/lab8-tree/test $ ./Tree
3
5
9
10
11
found 3
did not find 7
```

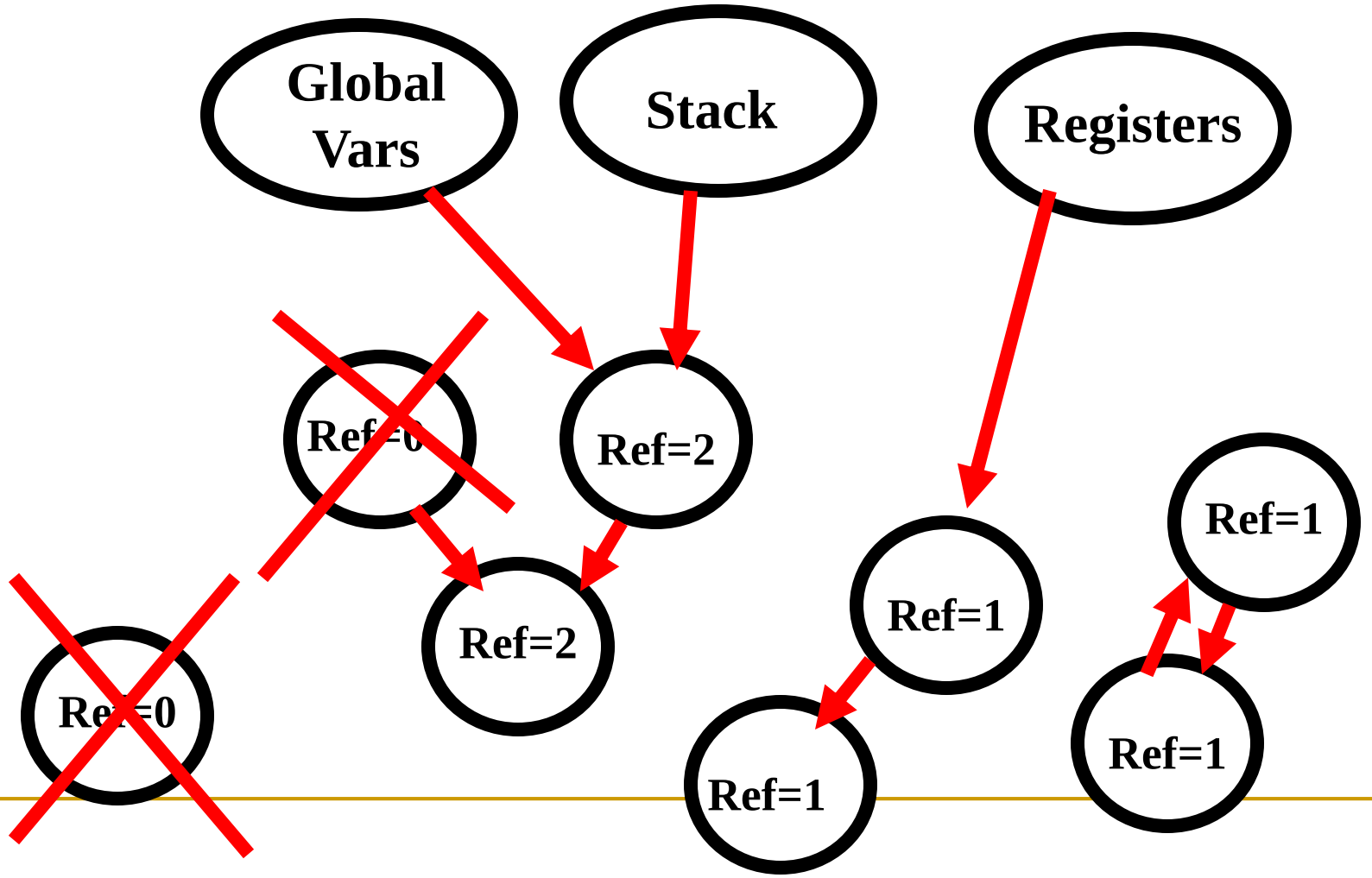
Reference Counting

- Reference counting is a garbage collection technique where an object has a reference counter that keeps track of the number of references to the object.
- Every time a new pointer points to the object, the reference counter is increased.
- When a pointer no longer points to the object, the reference is decreased.
- When the reference counter reaches 0, the object is removed.

Reference Counting

- In reference counting every object has a counter with the number of reference pointing to the object,
- When the reference count reaches 0 the object is removed.
- This is the approach used by languages such as Perl and Phyton or C++ with “Smart Pointers”.
- Advantage:
  - Easy to implement. Incremental (objects are freed while program is running)
- Disadvantage:
  - Slow. Every pointer assignment needs to increase/decreased a reference count.
  - Unable to free cycles
- In Multithreaded environments a mutex lock is necessary when incrementing/decrementing reference counter.

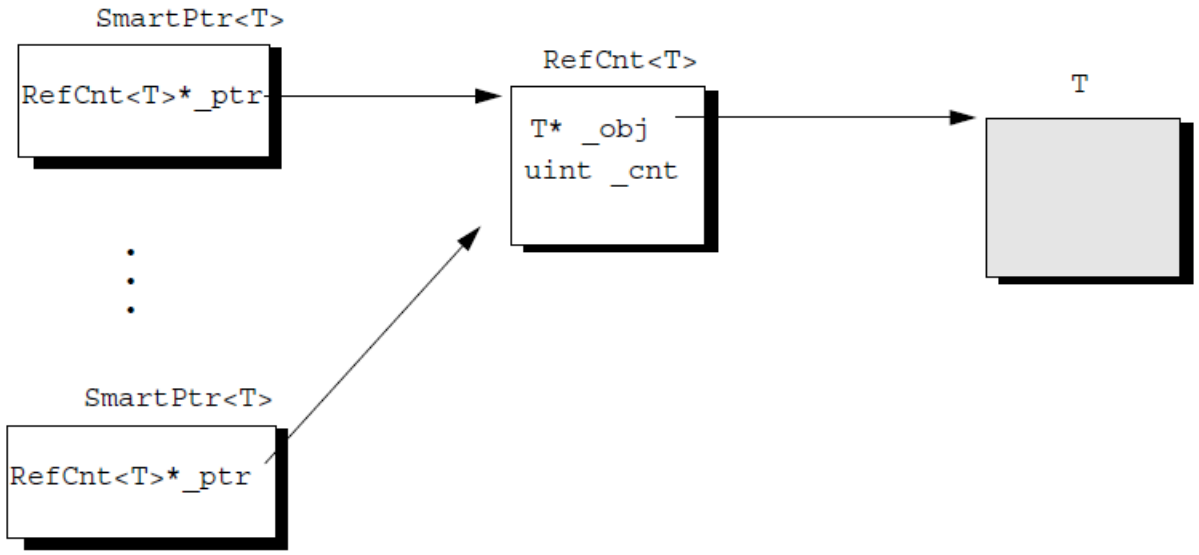
Reference Counting



Smart Pointers

- Based on Dr. Lavender Class notes at http://www.cs.utexas.edu/~lavender/courses/cs371/lectures/lecture-15.pdf
- Using operator overloading in C++ we can wrap the pointer operations such as “*” and “&” as well as “=“ to increment/decrement a reference counter.
- Also we can wrap an object around a Reference Count wrapper.
- Multiple SmartPtr<T> template instances point to a single RefCnt<T> instance, which holds a pointer and a reference count to an instance of a class T (e.g., string).
- The SmartPtr<T> template is a *handle* that is used to access a reference counted object and which updates the reference count.
  - On construction of a SmartPtr<T> instance, the count in incremented
  - On destruction the count is decremented.
  - When the last SmartPtr<T> object is destructed, the RefCnt<T> is deleted, which in turn deletes the object of type T.

Smart Pointers



Smart Pointers

SmartPtr.h

```
#include <iostream>
using namespace std;
template<class T>
class RefCnt {
 private:
      T* _obj;
      unsigned long _cnt;
 public:
      RefCnt(T* p) : _obj(p), _cnt(0) {
                                          cout << "New object " << (void *) _obj << endl;
      }
      ~RefCnt() {
                cout << "Delete object " << (void *) _obj << endl;
                assert(_cnt == 0); delete _obj;
      }
      T* object() const { return _obj; }
      int inc() {
                _cnt++;
                cout <<" increment:" << _obj << " _cnt=" << _cnt << "\n";
                return _cnt;
      }
      int dec() {
                _cnt--;
                cout <<" decrement:" << _obj << " _cnt=" << _cnt << "\n";
                return _cnt;
      }
};
```

Smart Pointers

```
/* A ``smart pointer to a reference counted object */
template<class T> class SmartPtr {
        RefCnt<T>* _ptr; // pointer to a reference counted object of type T

        /* hide new/delete to disallow allocating a SmartPtr<T> from the heap */
        static void* operator new(size_t) {}
        static void operator delete(void*) {}
 public:
        SmartPtr(T* p) {
                _ptr = new RefCnt<T>(p);
                _ptr->inc();
        }
        SmartPtr(const SmartPtr<T>& p) {
                _ptr = p._ptr;
                _ptr->inc();
        }

        ~SmartPtr() { if (_ptr->dec() == 0) { delete _ptr; } }

        void operator=(const SmartPtr<T>& p) {
                if (this != &p) {
                                if (_ptr->dec() == 0) delete _ptr;
                                _ptr = p._ptr;
                                _ptr->inc();
                }
        }

        T* operator->() const { return _ptr->object(); }
        T& operator*() const { return *(_ptr->object()); }
};
```

Smart Pointers

testSmartPtr.cpp:

```
#include <string>
#include "SmartPtr.h"

typedef SmartPtr<string> StringPtr;

int main()
{
	cout << "StringPtr p = new string(\"Hello, world\")" << endl;
	StringPtr p = new string("Hello, world");

	cout << "StringPtr s = p" << endl;
	StringPtr s = p; // invokes copy constructor incrementing reference counter

	cout << "Invoke a String method on a StringPtr " << endl;
	cout << "Length of " << *s << " is "  << s->length() << endl;

	cout << "s = new string(\"s\");" << endl;
	s = new string("s");

	cout << "p = new string(\"p\");" << endl;
	p = new string("p");

	cout << "Before Return"<< endl;

	return 0;
};
```

Smart Pointers

Makefile:

```
goal: TestSmartPtr

TestSmartPtr: TestSmartPtr.cpp SmartPtr.h
    g++ -o TestSmartPtr TestSmartPtr.cpp

clean:
    rm -f TestSmartPtr core
```

Smart Pointers

Output:

```
lore 228 $ ./TestSmartPtr
StringPtr p = new string("Hello, world")
New object 0x22878
 increment:0x22878 _cnt=1
StringPtr s = p
 increment:0x22878 _cnt=2
Invoke a String method on a StringPtr
Length of Hello, world is 12
s = new string("s");
New object 0x22898
 increment:0x22898 _cnt=1
 decrement:0x22878 _cnt=1
 increment:0x22898 _cnt=2
 decrement:0x22898 _cnt=1
p = new string("p");
New object 0x228b8
 increment:0x228b8 _cnt=1
 decrement:0x22878 _cnt=0
Delete object 0x22878
 increment:0x228b8 _cnt=2
 decrement:0x228b8 _cnt=1
Before Return
 decrement:0x22898 _cnt=0
Delete object 0x22898
 decrement:0x228b8 _cnt=0
Delete object 0x228b8
```

STL Smart Pointers

- The STL has already a standard implementation of Smart Pointers similar to the ones described.

```
#include <iostream>
#include <memory>
#include <thread>
#include <chrono>
#include <mutex>

struct Base
{
   Base() { std::cout << "  Base::Base()\n"; }
   // Note: non-virtual destructor is OK here
   ~Base() { std::cout << "  Base::~Base()\n"; }
};
struct Derived: public Base
{
   Derived() { std::cout << "  Derived::Derived()\n"; }
   ~Derived() { std::cout << "  Derived::~Derived()\n"; }
};
```

STL Smart Pointers

```
void thr(std::shared_ptr<Base> p)
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::shared_ptr<Base> lp = p; // thread-safe, even though the
                                  // shared use_count is incremented
    {
      static std::mutex io_mutex;
      std::lock_guard<std::mutex> lk(io_mutex);
      std::cout << "local pointer in a thread:\n"
                << "  lp.get() = " << lp.get()
                << ", lp.use_count() = " << lp.use_count() << '\n';
    }
}
```

STL Smart Pointers

```
int main()
{
    std::shared_ptr<Base> p = std::make_shared<Derived>();

    std::cout << "Created a shared Derived (as a pointer to Base)\n"
              << "  p.get() = " << p.get()
              << ", p.use_count() = " << p.use_count() << '\n';
    std::thread t1(thr, p), t2(thr, p), t3(thr, p);
    p.reset(); // release ownership from main
    std::cout << "Shared ownership between 3 threads and released\n"
              << "ownership from main:\n"
              << "  p.get() = " << p.get()
              << ", p.use_count() = " << p.use_count() << '\n';
    t1.join(); t2.join(); t3.join();
    std::cout << "All threads completed, the last one deleted Derived\n";
}
```

Lock Guards

- Lock Guards is a wrapper that automatically locks a mutex lock when entering a function and unlock the mutex when exiting the function.
- Assume the following code:

```
pthread_mutex_t mutex;
void mt_function() {
   mutex_lock( &mutex);
   // Synchronized code
   ...
   if (error) {
      return;  // Oops. Forgot to unlock mutex
   }
   mutex_unlock(&mutex);
}
```

Lock Guards

- Mutex locks may be prone to errors. The user may forget to unlock the mutex before returning, or an exception may be thrown while holding the mutex.
- Lock Guards are objects that wrap a mutex lock to lock it during construction at the beginning of the method and unlock it during desruction at the end of the method.

Lock Guards

```
class LockGuard {
    pthread_mutex_t & mutex;
public:
    LockGuard( pthread_mutex_t & mutex) {
      this->mutex = mutex;
      pthread_mutex_lock(&mutex);
    }
    ~LockGuard() {
        pthread_mutex_unlock(&mutex);
    }
};
```

Lock Guards

```
void mt_function() {
    LockGuard myLock( mutex)
    // Synchronized code
     ...
     if (error) {
        return;
        // Oops. Forgot to unlock mutex
        // No problem!
        // Destructor of LockGuard will unlock it at return.
     }
     // No need to call pthread_mutex_unlock() at return.
     // Destructor of LockGuard will unlock it at return.
  }
```

Final Review

- C development cycle:
- C Programming Structure
- Control Flow
- While/for/do, etc
- Functions
- Arrays & Strings
- Pointer
- Dynamic memory
- Data Types: structures, unions, strings
- Streams (Files)
- File calls/ directories

Final Review

- Java and C++
- Example of a C++ program. Stack.h and Stack.cpp
- Reference Data Types
- Passing by Reference and by Value
- Constant Parameters
- Default Parameters.
- Function Overloading
- Operator Overloading

Final Review

- Classes
- private:, protected: public:
- friends
- Inline functions
- new and delete
- Objects as local variables
- Constructors and Destructors
- Copy constructor
- Assignment operator in classes
- Exception Handling

Final Review

- Namespaces “using namespace std”
- Standard namespaces. “using namespace std;”.
- Public inheritance
- Constructor in a subclass
- Dynamic cast
- Virtual Methods
- Abstract classes

Final Review

- Virtual Destructors
- Private and Protected Inheritance
- Multiple Inheritance
- Parameterized Types and Templates
- Writing a Template: A ListInt class and a ListGeneric template.
- Using a Template
- Iterator Templates
- Default Template Parameters
- Function Templates

Final Review

- C++ Input/Output Library
- File Streams
- Standard Template Library (STL)
- Containers and Iterators
- Vectors
- Lists
- Maps
- STL Strings
- Memory Allocation Errors
- Smart Pointers
- Lock Guards

To Study

- Class slides
- Practice Exam
- Study Guide. See web page.

Additional Slides

C++ Quizz 1

1. In the following code

```
class A {
  int x;
  void m();
};
```

the definition of the variable x is:

a) private

b) public

c) protected

d) a syntax error

C++ Quizz 1

2. In the following code:

```
void foo(int & a) {
  a = 2;
}

main() {
  int y = 5;
  foo(y);
  printf("y=%d\n",y);
}
```

The output is:

a) y=5
b) y=2
c) Syntax error
d) Undetermined

C++ Quizz 2

1. In the following code

```
test_desc.cpp
class A {
      int x;
 public:
    A();
    ~A()
  };
```

```
A::A() {
   x = 5;
}
A:~A() {
printf("Done!");
}
```

```
A g;
main() {
    A a;
    A * b = new A;
    delete b;
}
```

The output is:

a) No output
b) Done!
c) Done! Done!
d) Done!Done!Done!

C++ Quizz 2

- The solution is c)
- The destructor for object b is called in delete b. → Prints “Done!”
- The destructor for a is called when main returns. -> Prints “Done!”
- The destructor for g is called when the program exits. . -> Prints “Done!”

C++ Quiz 3

- The solution is c)
- The destructor for object b is called in delete b. → Prints “Done!”
- The destructor for a is called when main returns. -> Prints “Done!”
- The destructor for g is called when the program exits. . -> Prints “Done!”

Lab3: Implementing resizable table

resizable_table.h

```
#if !defined RESIZABLE_ARRAY_H
#define RESIZABLE_ARRAY_H

#define INITIAL_SIZE_RESIZABLE_TABLE 10

typedef struct RESIZABLE_TABLE_ENTRY {
        char * name;
        void * value;
} RESIZABLE_TABLE_ENTRY;

typedef struct RESIZABLE_TABLE {
        int maxElements;
        int currentElements;
        struct RESIZABLE_TABLE_ENTRY * array;
} RESIZABLE_TABLE;

RESIZABLE_TABLE * rtable_create();
int rtable_add(RESIZABLE_TABLE * table, char * name, void * value);
…
#endif
```

Lab3: Implementing a resizable table

resizable_table.cpp:

```
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include "resizable_table.h"

//
// It returns a new RESIZABLE_TABLE. It allocates it dynamically,
// and initializaes the values. The initial maximum size of the array is 10.
//
RESIZABLE_TABLE * rtable_create() {

        // Allocate a RESIZABLE_TABLE
        RESIZABLE_TABLE * table = (RESIZABLE_TABLE *) malloc(sizeof(RESIZABLE_TABLE));
        if (table == NULL) {
                return NULL;
        }
        // Initialze the members of RESIZABLE_TABLE
        table->maxElements = INITIAL_SIZE_RESIZABLE_TABLE;
        table->currentElements = 0;
        table->array = (struct RESIZABLE_TABLE_ENTRY *)
                malloc(table->maxElements*sizeof(RESIZABLE_TABLE_ENTRY));
        if (table->array==NULL) {
                return NULL;
        }

        return table;
}
```

Lab3: Implementing a resizable table

```
//
// Adds one pair name/value to the table. If the name already exists it will
// Substitute its value. Otherwise, it will store name/value in a new entry.
// If the new entry does not fit, it will double the size of the array.
// The name string is duplicated with strdup() before assigning it to the
// table.
//
int rtable_add(RESIZABLE_TABLE * table, char * name, void * value) {

        // Find if it is already there and substitute value

        // If we are here is because the entry was not found.

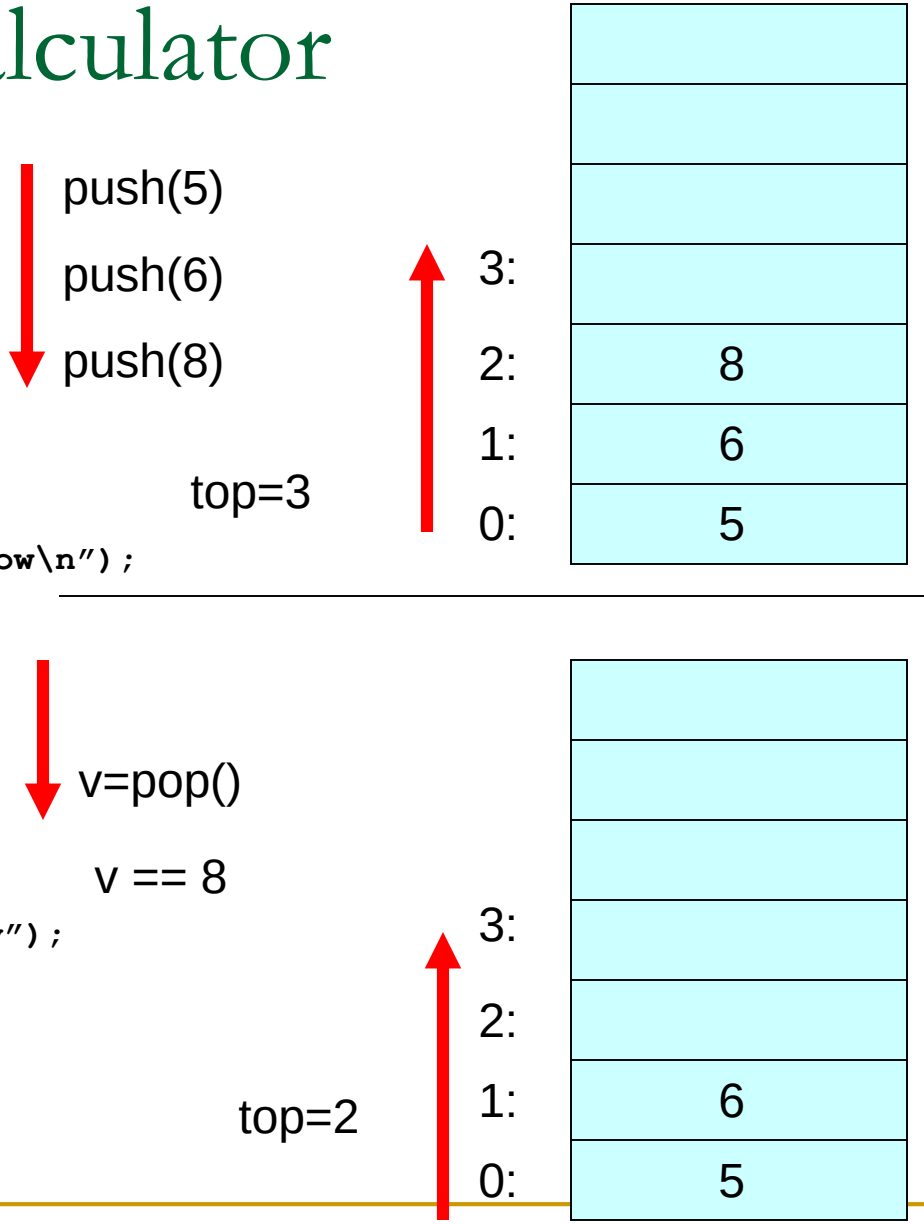
        // Make sure that there is enough space

        //
        // Add name and value to a new entry.
        // We need to use strdup to create a copy of the name but not value.
        //

        return 0;
}
```

Lab 4: RPN Calculator

```
// Implementation of a stack
#define MAXSTACK 100
double stack[MAXSTACK];
int top = 0;
void push(double val)
{
    if ( top == MAXSTACK) {
      printf("push: Stack overflow\n");
         exit(1);
    }
   stack[top] = val;
   top++;
}
double pop()
{
  if (top ==0) {
         printf("pop: Stack empty");
         exit(1);
  }
  top--;
  return stack[top];
}
```



Lab 4: RPN Calculator

```
int main( int argc, char ** argv) {
  // argv is the array of arguments
  int i;
  // Printing the arguments
  for (i=0; i < argc; i++) {
    printf("%d: %s\n", i, argv[i]);
  }

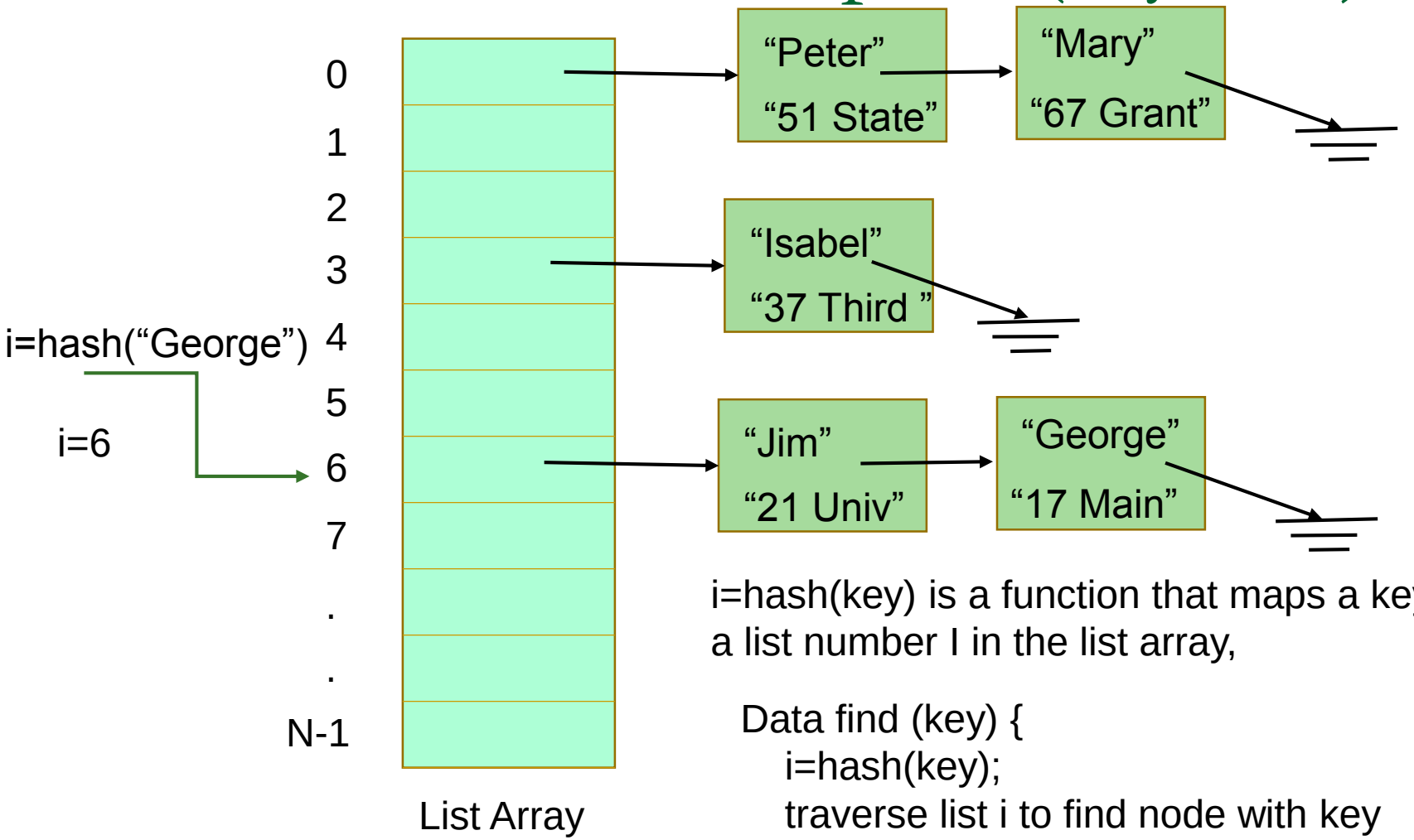
  // For all args
  //   If argv[i] is a number,
  //     convert it to double and push it to the stack.
  //   If argv[i] is "+"
  //     v1=pop(), v2=pop(), push(v1+v2)
  //   similar for other operands
  // end
  // result will be in stack[0]
}
```

A Hash Table Class

- A Hash Table is a data structure that can store (key,data) pairs like linked lists or trees.
- Hash Tables have the advantage that they insert, find, or remove items in constant time in average.
- Hash table consist of an array of linked lists
- The key is translated to an index in the arrays using a hash funtion. Usually it is a mod operator

  index = key % ArrayList;

Hash Table – Stores pairs (key,data)



i=hash(key) is a function that maps a key to a list number I in the list array,

```
Data find (key) {
    i=hash(key);
    traverse list i to find node with key
    if node found return data else return null
}
```

Hash Function

- The hash function takes a key and returns a number between 0 and N-1 where N is the number of lists in the hash table.
- If the key is int, then a typical hash function is:

```
int hash(int key) {
  return key % N;
}
```

Hash Function

- If the key is string, then a typical hash function adds all the ascii chars module N:

```
int hash(char * key) {
        int sum = 0;
        while (*key) {
          sum += *key;
          key++;
        }
        return sum % N;
}
```

Hash Table - Find

```
Data find(key) {
    i = hash(key)
    list = listArray[i];
    traverse list until node with key is found.
    if node is found
       return data in node
    else
       return null
}
```

Hash Table - Insert

```
void insert(key,data) {
  i = hash(key)
  list = listArray[i]
  traverse list until it finds a node that contains key
  if node found {
     substitute data
     return
  }
  else {
     create node and store data and key
     insert node in list
  }
}
```

HashTableVoid.h

```
#include <assert.h>
#include <stdlib.h>
#include <string.h>

// Each hash entry stores a key, object pair
struct HashTableVoidEntry {
  const char * _key;
  void * _data;
  HashTableVoidEntry * _next;
};

// This is a Hash table that maps string keys to objects of type Data
class HashTableVoid {
 public:
  // Number of buckets
  enum { TableSize = 2039};

  // Array of the hash buckets.
  HashTableVoidEntry **_buckets;

  // Obtain the hash code of a key
  int hash(const char * key);
```

HashTableVoid.h

```
public:
  HashTableVoid();

  // Add a record to the hash table. Returns true if key already exists.
  // Substitute content if key already exists.
  bool insertItem( const char * key, void * data);

  // Find a key in the dictionary and place in "data" the corresponding record
  // Returns false if key is does not exist
  bool find( const char * key, void ** data);

  // Removes an element in the hash table. Return false if key does not exist.
  bool removeElement(const char * key);
};
```

HashTableVoid.h

```
class HashTableVoidIterator {
  int _currentBucket;
  HashTableVoidEntry *_currentEntry;
  HashTableVoid * _hashTable;
 public:
  HashTableVoidIterator(HashTableVoid *
hashTable);
  bool next(const char * & key, void * & data);
};
```

Hash Table: Hash Function and Constructor (HashTableVoid.cpp)

```
#include "HashTableVoid.h"

int HashTableVoid::hash(const char * key)
{
  int h = 0;
  const char * p = key;
  while (*p) {
    h += *p;
    p++;
  }
  return h % TableSize;
}

HashTableVoid::HashTableVoid()
{
  _buckets = (HashTableVoidEntry **) malloc(TableSize*sizeof(HashTableVoidEntry*));
  for ( int i = 0; i < TableSize; i++) {
    _buckets[i] = NULL;
  }
}
```

Hash Table: insertItem()

```
bool HashTableVoid::insertItem( const char * key, void * data)
{
  // Get hash bucket
  int h = hash(key);

  HashTableVoidEntry * e = _buckets[h];
  while (e!=NULL) {
    if (!strcmp(e->_key, key)) {
      // Entry found
      e->_data = data;
      return true;
    }
    e = e->_next;
  }

  // Entry not found. Add it.
  e = new HashTableVoidEntry;
  e->_key = strdup(key);
  e->_data = data;
  e->_next = _buckets[h];
  _buckets[h] = e;
  return false;
}
```

Hash Table: find()

```
bool HashTableVoid:: find( const char * key, void ** data);
{
  // Get hash bucket
  int h = hash(key);

  HashTableVoidEntry * e = _buckets[h];
  while (e!=NULL) {
    if (!strcmp(e->_key, key)) {
      // Entry found
      *data = e->_data;
      return true;
    }
    e = e->_next;
  }
  return false;
}
```

Hash Table: RemoveElement()

```
bool HashTableVoid::removeElement(const char * key)
{
  // Get hash bucket
  int h = hash(key);

  HashTableVoidEntry * e = _buckets[h];
  HashTableVoidEntry * prev = NULL;
  while (e!=NULL) {
    if (!strcmp(e->_key, key)) {
      // Entry found
      if (prev != NULL) {
        prev->_next = e->_next;
      }
      else {
        _buckets[h] = e->_next;
      }
      free(e->_key);
      delete e;
      return true;
    }
    prev = e;
    e = e->_next;
  }
  return false;
}
```

Hash Table Test

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


struct Student {
  const char * name;
  int grade;
};

Student students[] = {
  {"Rachael", 8 },
  {"Monica", 9},
  {"Phoebe", 10},
  {"Joey", 6},
  {"Ross", 8},
  {"Chandler", 7}
};

struct Vars {
  const char * varName;
  const char * value;
};

Vars vars[] = {
  {"a", "abcd"},
  {"b", "efgh"},
  {"c", "defg"}
};
```

Hash Table Test

```
Int main()
{
  HashTableVoid h;

  for (int i=0; i<sizeof(students)/sizeof(Student);i++) {
    bool e;
    e = h.insertItem(students[i].name, (void*) students[i].grade);
    assert(!e);
  }

  for (int i=0; i<sizeof(students)/sizeof(Student);i++) {
    bool e;
    int grade;
    void * gradev;
    e = h.find(students[i].name, &gradev);
    grade = (int)gradev;
    assert(e);
    assert(grade==students[i].grade);
  }
```

Hash Table Test

```
  int grade;
  void * gradev;
  bool e = h.find("John",&gradev);
  grade = (int)gradev;
  assert(!e);

  e = h.removeElement("John");
  assert(!e);

  e = h.removeElement("Rachael");
  assert(e);
}
```

Hash Table Properties

- A hash table has operations insert, find, remove operations take O(1) expected constant time.
- The worst case is when the hash table lists become very long and then the operations become O(n).

Hash Table Properties

- To make sure that the operations take O(1) expected time, the hash table should satisfy the following properties:
  - The hash function should be evenly distributed, the lists have more less the same length.
  - The number of lists in the hash table should be about the same as the number of entries stored in the table.
- This will make sure that each list will have very few (0,1,2) entries in each list.

Lab7 Sound Class

```
Sound::read(const char * fileName)
    {
    FILE * f = fopen(fileName, "r");

    if (f == NULL) {
    return false;
    }

    // Get size of file
    struct stat buf;
    stat(fileName, &buf);
    int fsize = buf.st_size;

    // Allocate memory for wave file
    WaveHeader * hdr = (WaveHeader *) malloc(fsize+1);

    // Read file
    fread( hdr, fsize, 1, f);

    fclose(f);

    // Fill it up
    this->numChannels = hdr->numChannels[0];
    this->sampleRate = getLittleEndian4(hdr->sampleRate);
    this->bitsPerSample = hdr->bitsPerSample[0]+(hdr->bitsPerSample[1]<<8);
    this->bytesPerSample = (this->bitsPerSample/8)*this->numChannels;
    this->hdr = (char *) hdr;
    this->lastSample = getLittleEndian4(hdr->subchunk2Size)/this->bytesPerSample;
    this->maxSamples = this->lastSample;

    return true;
}
```