C Programming Language Review
C: A High-Level Language

Gives symbolic names to values
  – don’t need to know which register or memory location

Provides abstraction of underlying hardware
  – operations do not depend on instruction set
  – example: can write “a = b * c”, even if CPU doesn’t have a multiply instruction

Provides expressiveness
  – use meaningful symbols that convey meaning
  – simple expressions for common control patterns (if-then-else)

Enhances code readability

Safeguards against bugs
  – can enforce rules or conditions at compile-time or run-time
A C Code “Project”

- You will use an “Integrated Development Environment” (IDE) to develop, compile, load, and debug your code.
- Your entire code package is called a *project*. Often you create several files to split the functionality:
  - Several C files
  - Several include (.h) files
  - Maybe some assembly language (.a30) files
  - Maybe some assembly language include (.inc) files

- A lab, like “Lab7”, will be your project. You may have three .c, three .h, one .a30, and one .inc files.
- More will be discussed in a later set of notes.
Compiling a C Program

Entire mechanism is usually called the “compiler”

Preprocessor
- macro substitution
- conditional compilation
- “source-level” transformations
  - output is still C

Compiler
- generates object file
  - machine instructions

Linker
- combine object files (including libraries) into executable image
Compiler

Source Code Analysis
- “front end”
  - parses programs to identify its pieces
    - variables, expressions, statements, functions, etc.
  - depends on language (not on target machine)

Code Generation
- “back end”
  - generates machine code from analyzed source
  - may optimize machine code to make it run more efficiently
  - very dependent on target machine

Symbol Table
- map between symbolic names and items
- like assembler, but more kinds of information
## Memory Map for Our MCU

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000h</td>
<td>On-chip RAM</td>
</tr>
</tbody>
</table>
| 0001 8000h| Reserved area
d| |
| 0008 0000h| Peripheral I/O registers           |
| 0010 0000h| On-chip ROM (data flash)           |
| 0010 8000h| Reserved area
d| |
| 007F 8000h| FCU-ROM                           |
| 007F A000h| Reserved area
d| |
| 007F C000h| Peripheral I/O registers           |
| 007F C500h| Reserved area
d| |
| 007F FC00h| Peripheral I/O registers           |
| 0080 0000h| Reserved area
d| |
| 00F8 0000h| On-chip ROM (program ROM) (write only) |
| 0100 0000h| Reserved area
d| |
| FF00 0000h| On-chip ROM (FCU firmware) (read only) |
| FF00 C000h| On-chip ROM (user boot) (read only) |
| FF80 0000h| Reserved area
d| |
| FFF8 0000h| On-chip ROM (program ROM) (read only) |
| FFFF FFFFh| Reserved area
d| |
Classifying Data

Variable data
- Automatic variable
  - With initial value
  - Without initial value
- Static variable
  - With initial value
  - Without initial value

Fixed data
- Constant, character string
- Program

To stack area
To RAM and ROM areas
To RAM area
To ROM area
To ROM area

Figure 2.1.1 Types of data and code generated by NC30 and their mapped areas
Storage of Local and Global Variables

```c
int inGlobal;

void chapter12() {
    int inLocal;
    int outLocalA;
    int outLocalB;

    /* initialize */
    inLocal = 5;
    inGlobal = 3;

    /* perform calculations */
    outLocalA = inLocal++ & ~inGlobal;
    outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);
}
```
Another Example Program with Function Calls

const int globalD=6;
int compute(int x, int y);
int squared(int r);

void main() {
    int a, b, c;  // These are main’s automatic
    // variables, and will be
    a = 10;    // stored in main’s frame
    b = 16;
    c = compute(a,b);
}

int compute(int x, int y) {
    int z;
    z = squared(x);
    z = z + squared(y) + globalD;
    return(z);
}

int squared(int r) {
    return (r*r);
}
Control Structures

• if – else
• switch
• while loop
• for loop
If-else

If (condition)
  action_if;
else
  action_else;

Else allows choice between two mutually exclusive actions without re-testing condition.
switch (expression) {
    case const1:
        action1; break;
    case const2:
        action2; break;
    default:
        action3;
}

*Alternative to long if-else chain. If break is not used, then case "falls through" to the next.*
While

while (test)
  loop_body;

Executes loop body as long as test evaluates to TRUE (non-zero).

Note: Test is evaluated before executing loop body.
For

for (init; end-test; re-init) statement

Executes loop body as long as test evaluates to TRUE (non-zero). Initialization and re-initialization code included in loop statement.

Note: Test is evaluated before executing loop body.
<table>
<thead>
<tr>
<th>Decimal</th>
<th>Character</th>
<th>00</th>
<th>nul</th>
<th>10</th>
<th>dle</th>
<th>20</th>
<th>sp</th>
<th>30</th>
<th>0</th>
<th>40</th>
<th>@</th>
<th>50</th>
<th>P</th>
<th>60</th>
<th>`</th>
<th>70</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>soh</td>
<td>11</td>
<td>dc1</td>
<td>21</td>
<td>!</td>
<td>31</td>
<td>1</td>
<td>41</td>
<td>A</td>
<td>51</td>
<td>Q</td>
<td>61</td>
<td>a</td>
<td>71</td>
<td>q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>stx</td>
<td>12</td>
<td>dc2</td>
<td>22</td>
<td>&quot;</td>
<td>32</td>
<td>2</td>
<td>42</td>
<td>B</td>
<td>52</td>
<td>R</td>
<td>62</td>
<td>b</td>
<td>72</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>etx</td>
<td>13</td>
<td>dc3</td>
<td>23</td>
<td>#</td>
<td>33</td>
<td>3</td>
<td>43</td>
<td>C</td>
<td>53</td>
<td>S</td>
<td>63</td>
<td>c</td>
<td>73</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>eot</td>
<td>14</td>
<td>dc4</td>
<td>24</td>
<td>$</td>
<td>34</td>
<td>4</td>
<td>44</td>
<td>D</td>
<td>54</td>
<td>T</td>
<td>64</td>
<td>d</td>
<td>74</td>
<td>t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>enq</td>
<td>15</td>
<td>nak</td>
<td>25</td>
<td>%</td>
<td>35</td>
<td>5</td>
<td>45</td>
<td>E</td>
<td>55</td>
<td>U</td>
<td>65</td>
<td>e</td>
<td>75</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>ack</td>
<td>16</td>
<td>syn</td>
<td>26</td>
<td>&amp;</td>
<td>36</td>
<td>6</td>
<td>46</td>
<td>F</td>
<td>56</td>
<td>V</td>
<td>66</td>
<td>f</td>
<td>76</td>
<td>v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>bel</td>
<td>17</td>
<td>etb</td>
<td>27</td>
<td>'</td>
<td>37</td>
<td>7</td>
<td>47</td>
<td>G</td>
<td>57</td>
<td>W</td>
<td>67</td>
<td>g</td>
<td>77</td>
<td>w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>bs</td>
<td>18</td>
<td>can</td>
<td>28</td>
<td>(</td>
<td>38</td>
<td>8</td>
<td>48</td>
<td>H</td>
<td>58</td>
<td>X</td>
<td>68</td>
<td>h</td>
<td>78</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>ht</td>
<td>19</td>
<td>em</td>
<td>29</td>
<td>)</td>
<td>39</td>
<td>9</td>
<td>49</td>
<td>I</td>
<td>59</td>
<td>Y</td>
<td>69</td>
<td>i</td>
<td>79</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0a</td>
<td>nl</td>
<td>1a</td>
<td>sub</td>
<td>2a</td>
<td>*</td>
<td>3a</td>
<td>:</td>
<td>4a</td>
<td>J</td>
<td>5a</td>
<td>Z</td>
<td>6a</td>
<td>j</td>
<td>7a</td>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b</td>
<td>vt</td>
<td>1b</td>
<td>esc</td>
<td>2b</td>
<td>+</td>
<td>3b</td>
<td>;</td>
<td>4b</td>
<td>K</td>
<td>5b</td>
<td>[</td>
<td>6b</td>
<td>k</td>
<td>7b</td>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0c</td>
<td>np</td>
<td>1c</td>
<td>fs</td>
<td>2c</td>
<td>,</td>
<td>3c</td>
<td>&lt;</td>
<td>4c</td>
<td>L</td>
<td>5c</td>
<td>\</td>
<td>6c</td>
<td>l</td>
<td>7c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0d</td>
<td>cr</td>
<td>1d</td>
<td>gs</td>
<td>2d</td>
<td>-</td>
<td>3d</td>
<td>=</td>
<td>4d</td>
<td>M</td>
<td>5d</td>
<td>]</td>
<td>6d</td>
<td>m</td>
<td>7d</td>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0e</td>
<td>so</td>
<td>1e</td>
<td>rs</td>
<td>2e</td>
<td>.</td>
<td>3e</td>
<td>&gt;</td>
<td>4e</td>
<td>N</td>
<td>5e</td>
<td>^</td>
<td>6e</td>
<td>n</td>
<td>7e</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0f</td>
<td>si</td>
<td>1f</td>
<td>us</td>
<td>2f</td>
<td>/</td>
<td>3f</td>
<td>?</td>
<td>4f</td>
<td>O</td>
<td>5f</td>
<td>_</td>
<td>6f</td>
<td>o</td>
<td>7f</td>
<td>del</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Masking

One of the most common uses of logical operations is “masking.”

Masking is where you want to examine only a few bits at a time, or modify certain bits.

For example, if I want to know if a certain number is odd or even, I can use an “and” operator.

\[
\begin{array}{cccccccc}
0101 & 0101 & 0101 & 0101 \\
\text{AND} & 0000 & 0000 & 0000 & 0001 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0000 & 0000 & 0000 & 0001 \\
\end{array}
\]

Or, lets say you want to look at bits 7 to 2:

\[
\begin{array}{cccccccc}
0101 & 0101 & 0101 & 0101 \\
\text{AND} & 0000 & 0000 & 1111 & 1100 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0000 & 0000 & 0101 & 0100 \\
\end{array}
\]

Code? Bitwise and is &, bitwise or is |
Let's assume three switches connected to port 1 like the following:

How do you read the three switches?
After you set the direction:

```c
int data;
data = (int) PORT1.PORT.BIT.B0;
```

All at the same time?
```c
data = (int) PORT1.PORT.BYTE;
```
Now, write the C code to interrogate the switches and print
  “Switch n pressed” if it is being pressed. Print
  “No switches printed” if none are being pressed.

    //perform bitwise AND for bit0, then 1, then 2
    if (!(data & 1)) printf("Switch 1 pressed/n");
    if
    if
    // if no switches pressed, say so
    if
Example - upper/lower case ASCII

Masking also lets you convert between ASCII upper and lower case letters:

- “A” = 0x41 (0100 0001)
- “a” = 0x61 (0110 0001)

To convert from capitals to lower case:
- Add 32 (0x20)
- OR with 0x20

To convert from lower case to capitals
- Subtract 32 (0x20)
- AND 0xDF

The logical operations are the only way to ensure the conversion will always work
1D Arrays

- Declaration of one-dimensional array
  ```
  char buff1[3];
  int buff2[3];
  ```

- Declaration and initialization of one-dimensional array
  ```
  char buff1[] = {'a', 'b', 'c'};

  int buff2[] = {10, 20, 30};
  ```

Figure 1.7.2 Declaration of one-dimensional array and memory mapping
2D Arrays

C arrays are stored in a \textit{row-major} form (a row at a time)
Pointers

A pointer variable holds the address of the data, rather than the data itself.

To make a pointer point to variable `a`, we can specify the address of `a`:

- address operator `&`

The data is accessed by dereferencing (following) the pointer:

- indirection operator `*` works for reads and writes

Assigning a new value to a pointer variable changes where the variable points, not the data.

```c
void main () {
    int i, j;
    int *p1, *p2;
    i = 4;
    j = 3;
    p1 = &i;
    p2 = &j;
    *p1 = *p1+*p2;
    p2 = p1;
}
```
More about Pointers

Incrementing and decrementing pointers to array elements
- Increment operator ++ makes pointer advance to next element (next larger address)
- Decrement operator -- makes pointer move to previous element (next smaller address)
- These use the size of the variable’s base type (e.g. int, char, float) to determine what to add
  - \texttt{p1++} corresponds to \texttt{p1 = p1 + sizeof(int)};
  - \texttt{sizeof} is C macro which returns size of type in bytes

Pre and post
- Putting the ++/-- \textbf{before} the pointer causes inc/dec \textbf{before} pointer is used
  - \texttt{int *p=100, *p2;}
  - \texttt{p2 = ++p;} assigns 102 to integer pointer \texttt{p2}, and \texttt{p} is 102 afterwards
- Putting the ++/-- \textbf{after} the pointer causes inc/dec \textbf{after} pointer is used
  - \texttt{char *q=200, *q2;}
  - \texttt{q2 = q--;} assigns 200 to character pointer \texttt{q2}, and \texttt{q} is 199 afterwards

```c
int a[18];
int * p;
p = &a[5];
*p = 5; /* a[5]=5 */
p++;
*p = 7; /* a[6]=7 */
p--;
*p = 3; /* a[5]=3 */
```
What else are pointers used for?

Data structures which reference each other
  – lists
  – trees
  – etc.

Exchanging information between procedures
  – Passing arguments (e.g. a structure) quickly – just pass a pointer
  – Returning a structure

Accessing elements within arrays (e.g. string)
Strings

See Section 16.3.4 of Patt & Patel.
There is no “string” type in C.
Instead an array of characters is used - `char a[44]`
The string is terminated by a NULL character (value of 0, represented in C by \0).
  – Need an extra array element to store this null

Example
  – `char str[10] = “testing”;

```
  testing \0

str
str[0]
str[1]
str[2]
```
Formatted String Creation

Common family of functions defined in stdio.h
– printf: print to standard output
– sprintf: print to a string
– fprintf: print to a file

Syntax: sprintf(char *str, char * fmt, arg1, arg2, arg3 .. );
– str: destination
– fmt: format specifying what to print and how to interpret arguments
  • %d: signed decimal integer
  • %f: floating point
  • %x: unsigned hexadecimal integer
  • %c: one character
  • %s: null-terminated string
– arg1, etc: arguments to be converted according to format string
sprintf Examples – strings and integers

```c
char s1[30], s2[30];
int a=5, b=10, c=-30;
char ch='$';
sprintf(s1, “Testing”);

sprintf(s2, “a=%d, b=%d”, a, b);

sprintf(s1, “b=%x, c=%d”, b, c);

sprintf(s1, “b=0x%x”, b);

sprintf(s2, “s1=%s”, s1);

sprintf(s1, “%c %c”, ch, s2);
```

Output:

- `s1` contains “Testing”.
- `s2` contains “a=5, b=10”.
- `s1` contains “b=a, c=-30”.
- `s1` contains “b=0xa”.
- `s2` contains “s1=b=0xa”.
- `s1` contains “$ s”.

---

Embedded Systems
sprintf Examples – floating-point

Variation on %f format specifier

- %-w.pf
  - - = left-justify. Optional
  - w = minimum field width (# of symbols)
  - p = precision (digits after decimal point)

Examples

```c
float f1=3.14, f2=9.991, f3=-19110.331;
char s1[30], s2[30];
sprintf(s1, "%f", f1);

sprintf(s1, "%f", f3);

sprintf(s1, "%4.1f", f2);
```

```
s1
3.140000

s1
-19110.3

s1
10.0
```
sprintf Examples – More Integers

Variation on %d format specifier for integers (d/i/o/x/u)

- %w.pd

  • - = left justify. Optional
  • w = minimum field width (# of symbols)
  • p = precision (digits). Zero pad as needed

Examples

```c
int a=442, b=1, c=-11;
char s1[30], s2[30];
sprintf(s1, "%5d", a);

sprintf(s1, "%-4d", b);

sprintf(s1, "%4d", b);

sprintf(s1, "%-5.4d", c);
```

```
s1  442
s1  1
s1  1
s1 -011
```
String Operations in string.h

Copy \textbf{ct} to \textbf{s} including terminating null character. Returns a pointer to \textbf{s}.
- \texttt{char* strcpy(char* s, const char* ct);}  
  s1 = “cheese”;
  s2 = “limburger”;
  strcpy(s1, s2); /* s1 = limburger */

Concatenate the characters of \textbf{ct} to \textbf{s}. Terminate \textbf{s} with the null character and return a pointer to it.
- \texttt{char* strcat(char* s, const char* ct);}  
  s1 = “cheese”;
  s2 = “ puffs”;
  strcat(s1, s2); /* s1 = cheese puffs */
More String Operations

Concatenate at most \( n \) characters of \( ct \) to \( s \). Terminate \( s \) with the null character and return a pointer to it.

- \( \text{char* strncat(char* s, const char* ct, int n);} \)
  
  \( s1 = \text{“cheese”}; \)
  
  \( s2 = \text{“puffs”}; \)
  
  \( \text{strncat(s1, s2, 4);} /* \text{cheese puf */} \)

Compares two strings. The comparison stops on reaching a \textbf{null} terminator. Returns a 0 if the two strings are identical, less than zero if \( s2 \) is greater than \( s1 \), and greater than zero if \( s1 \) is greater than \( s2 \). (Alphabetical sorting by ASCII codes)

- \( \text{int strcmp(const char* s1, const char* s2);} \)
  
  \( s1 = \text{“cheese”}; \)
  
  \( s2 = \text{“chases”}; \)
  
  \( \text{strcmp(s1, s2);} /* \text{returns non-zero number */} \)
  
  \( \text{strcmp(s1, “cheese”);} /* \text{returns zero */} \)
More String Operations

Return pointer to first occurrence of \texttt{c} in \texttt{s1}, or \texttt{NULL} if not found.
\begin{itemize}
\item \texttt{char* strchr(const char* s1, int c);}
\item \texttt{s1 = “Smeagol and Deagol”;}
\item \texttt{char a *;}
\item \texttt{a = strchr(s1, “g”); /* returns pointer to s1[4] */}
\end{itemize}

Return pointer to last occurrence of \texttt{c} in \texttt{s1}, or \texttt{NULL} if not found.
\begin{itemize}
\item \texttt{char* strrchr(const char* s1, int c);}
\item \texttt{s1 = “Smeagol and Deagol”;}
\item \texttt{char a *;}
\item \texttt{a = strchr(s1, “a”); /* returns pointer to s1[14] */}
\end{itemize}

Can use the returned pointer for other purposes
\begin{itemize}
\item \texttt{*a = ‘\0’; /* s1 = “Smeagol and De” */}
\item \texttt{strcat(s1, “spair”); /* s1 = “Smeagol and Despair” */}
\end{itemize}
Dynamic Memory Allocation in C

Why?

– Some systems have changing memory requirements, and stack variables (automatic) aren’t adequate
– Example: Voice recorder needs to store recordings of different lengths. Allocating the same size buffer for each is inefficient

How?

– Allocate nbytes of memory and return a start pointer
  • `void * malloc (size_t nbytes);`
– Allocate nelements*size bytes of memory and return a start pointer
  • `void * calloc (size_t nelements, size_t size);`
– Change the size of a block of already-allocated memory
  • `void * realloc (void * pointer, size_t size);`
– Free a block of allocated memory
  • `void free (void * pointer);`
Using Dynamic Memory Management

Request space for one or more new variables
- Request pointer to space for one element
  ```c
  int * j, *k;
  j = (int *) malloc (sizeof(int));
  *j = 37;
  ```
- Request pointer to space for array of elements and initialize to zero
  ```c
  k = (int *) calloc(num_elements, sizeof(int));
  k[0] = 55;
  k[1] = 31;
  ```
- These return NULL if there isn’t enough space
  - Program has to deal with failure -- embedded program probably shouldn’t just quit or reset....

Free up space when done using variables
```c
free(k);
```
Example Application: Voice Recorder

Recording
- While *record* switch is pressed
  - sample microphone
  - store in temporary RAM buffer
- When *record* switch is released
  - copy audio to a permanent buffer
  - add to end of list of recordings

Playback and skipping
- *forward* switch: skip forward over one recording, wrap around at end
- *play* switch: play the current recording
- *delete* switch: delete the current recording

Data Structure: linked list of recordings
Data Structure Detail: Linked List

Each list element is defined as a structure with fields

- AudioSize: Number of bytes
- AudioData: ...
- Next: Pointer to next list element

```c
typedef struct {
    unsigned AudioSize;
    char * AudioData;
    struct List_T * Next;
} List_T;
```
Code for Voice Recorder main

unsigned char buffer[MAX_BUFFER_SIZE];
struct List_T * recordings = NULL, * cur_recording = NULL;

void main(void) {
    while (1) {
        while (NO_SWITCHES_PRESSED)
            ;
        if (RECORD)
            handle_record();
        else if (PLAY)
            handle_play();
        else if (FORWARD)
            handle_forward();
        else if (DELETE)
            handle_delete();
    }
}
void handle_forward(void) {
    if (cur_recording)
        cur_recording = cur_recording->Next;
    if (!cur_recording)
        cur_recording = recordings;
}
void handle_record(void) {
    unsigned i, size;
    unsigned char * new_recording;
    struct List_T * new_list_entry;
    i = 0;
    while (RECORD)
        buffer[i++] = sample_audio();
    size = i;
    new_recording = (unsigned char *) malloc (size);
    for (i=0; i<size; i++) /* could also use memcpy() */
        new_recording[i] = buffer[i];
    new_list_entry = (List_T *) malloc ( sizeof(List_T) );
    new_list_entry->AudioData = new_recording;
    new_list_entry->AudioSize = size;
    new_list_entry->Next = NULL;
    recordings = Append(recordings, new_list_entry);
}
Code for handle_delete

```c
void handle_delete(void) {
    List_T * cur = recordings;
    if (cur == cur_recording)
        recordings = recordings->Next;
    else {
        while (cur->Next != cur_recording)
            cur = cur->Next;
        /* cur now points to previous list entry */
        cur->Next = cur_recording->Next;
    }
    free(cur_recording->AudioData);
    free(cur_recording);
}
```
Allocation Data Structures

Keep free memory in sorted list of free blocks

```c
typedef struct hdr {
    struct hdr * next;
    unsigned int size;
};
hdr * FreeList;
```

Assume hdr takes no space for examples

Allocation Operations

To allocate memory
- find first block of size \( \geq \) requested\_size
- modify list to indicate space isn’t free
  - if sizes match exactly, remove free block from list
  - else split memory
    - reduce size field by requested\_size, keeping first part of block in free space
    - allocate memory in second part of block
  - return pointer to newly allocated block

To free memory depends on block’s memory location
- If before first free block, prepend it at head of free list
- If between free list entries, insert in list
- If after last free block, append it at tail of free list

Freed memory block may be adjacent to other free blocks. If so, merge contiguous blocks
Dangers of Dynamic Memory Allocation

Memory leaks waste memory
  – Never freeing blocks which are no longer needed. User’s responsibility.

May accidentally use freed memory
  – User’s responsibility.

Allocation speed varies
  – Linked list must be searched for a block which is large enough
  – Bad for a real-time system, as worst case may be large.

Fragmentation
  – Over time free memory is likely to be broken into smaller and smaller fragments.
  – Eventually there won’t be a block large enough for an allocation request, even though there is enough total memory free
Heap and Fragmentation

Problem:
- `malloc/calloc/free` use a *heap* of memory; essentially a list of blocks of empty and used memory
- Repeated allocation/free cycles with differently sized allocation units leads to *fragmentation*
  - Although there may be enough memory free, it may be fragmented into pieces too small to meet request

Solutions (none optimal):
- Always allocate a fixed size memory element
- Use multiple heaps, each with a fixed element size
What is an Algorithm?


An algorithm is created in the design phase

How is an algorithm represented?
Typically represented as pseudo code
Historically represented as flowcharts

Do yourself a favor – write algorithms before code – always!
Pseudo Code

Pseudo code is written in English to describe the functionality of a particular software module (subroutine)
Include name of module/subroutine, author, date, description of functionality of module, and actual steps
Often you can take the pseudo code and use them lines in your program as comments!
Avoid a very fine level of detail (although this may sometimes be difficult to do)
Avoid writing code – use English, not assembly language (or higher-level language) instructions
An Example

Problem: Compare two numbers in \( x \) and \( y \), put the larger number in \( z \). If Equal, put 0 in \( z \).

Sample input/output:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( y )</td>
<td>( z )</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-4</td>
<td></td>
</tr>
<tr>
<td>-5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>-5</td>
<td>-4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Algorithm - Larger

Algorithm:
; Larger: Jim Conrad, 2011-09-13
; Purpose: Compare two numbers in x and y, put the larger number in z. If equal, put 0 in z.
Perform x-y
If result is positive ; x is bigger
    Put x in z, exit
If result is negative ; y is bigger
    Put y in z, exit
If zero,
    Put 0 in z, exit
An example

What do you think this does?

; ____________: Jim Conrad, 2011-09-13
; Purpose:
;
Set total to zero
Set grade counter to one
While grade counter is less than or equal to ten
   Input the next grade
   Add the grade into the total
Set the class average to the total divided by ten
Print the class average.

Borrowed from http://www.unf.edu/~broggio/cop2221/2221pseu.htm