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 spilt 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.

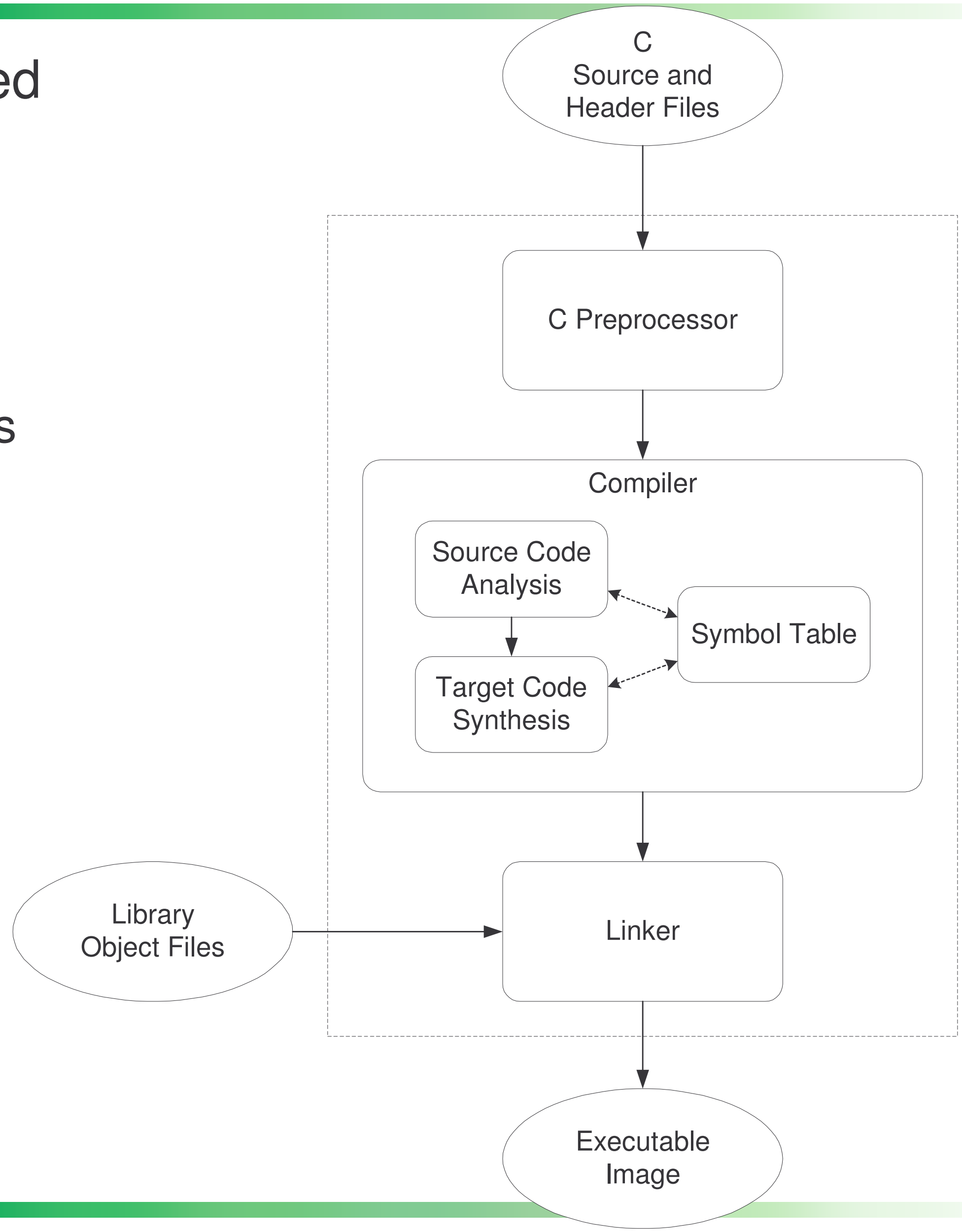
Compiling a C Program

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

Address	Area
0000 0000h	On-chip RAM
0001 8000h	Reserved area[*1]
0008 0000h	Peripheral I/O registers
0010 0000h	On-chip ROM (data flash)
0010 8000h	Reserved area[*1]
007F 8000h	FCU-RAM[3]
007F A000h	Reserved area[*1]
007F C000h	Peripheral I/O registers
007F C500h	Reserved area[*1]
007F FC00h	Peripheral I/O registers
0080 0000h	Reserved area[*1]
00F8 0000h	On-chip ROM (program ROM) (write only)
0100 0000h	Reserved area[*1]

Address	Area
	Reserved area[*1]
FEFF E000h	On-chip ROM (FCU firmware)[*3] (read only)
FF00 0000h	Reserved area[*1]
FF00 C000h	On-chip ROM (user boot) (read only)
FF80 0000h	Reserved area[*1]
FFF8 0000h	On-chip ROM (program ROM) (read only)
FFFF FFFFh	

Classifying Data

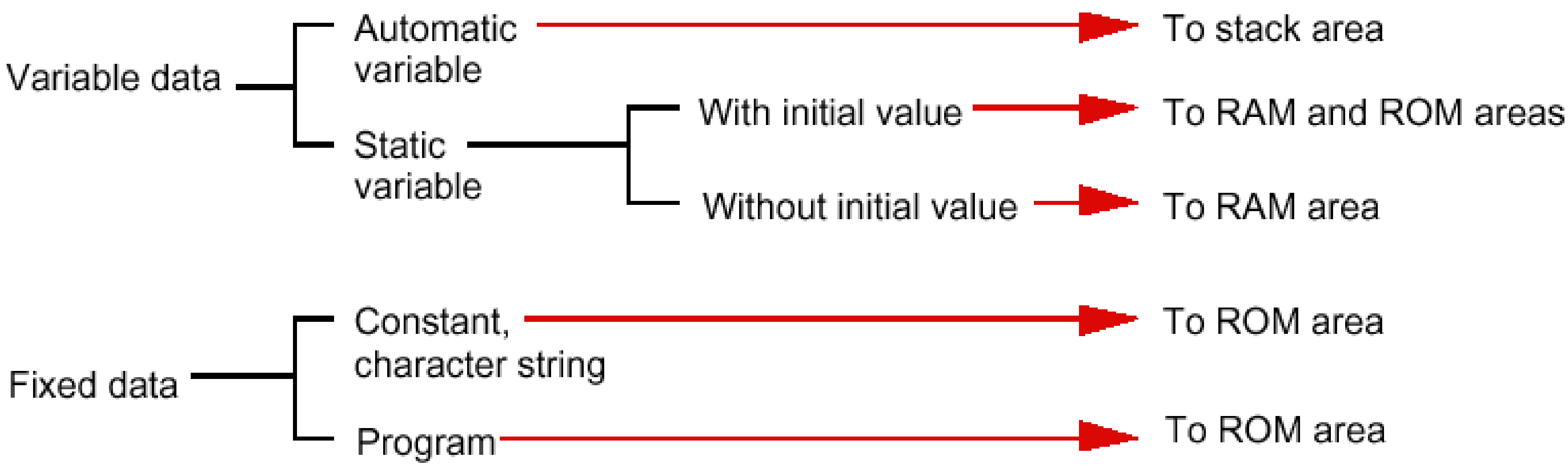


Figure 2.1.1 Types of data and code generated by NC30 and their mapped areas

Storage of Local and Global Variables

```
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

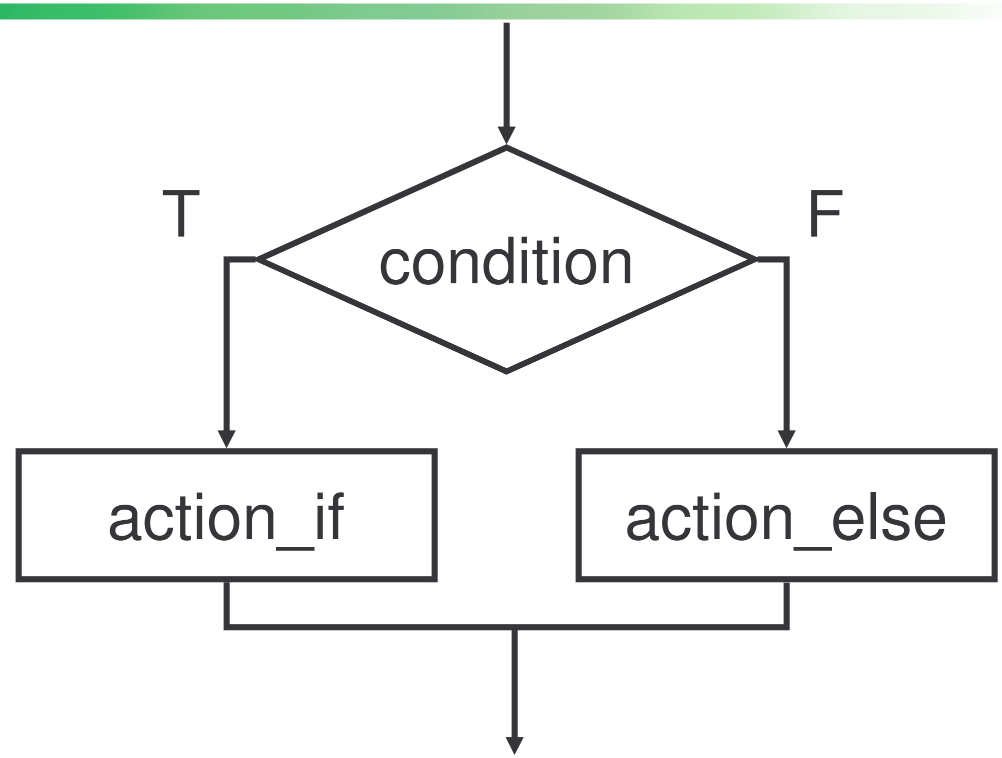
```
const int globalD=6;
int compute(int x, int y);
int squared(int r);

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

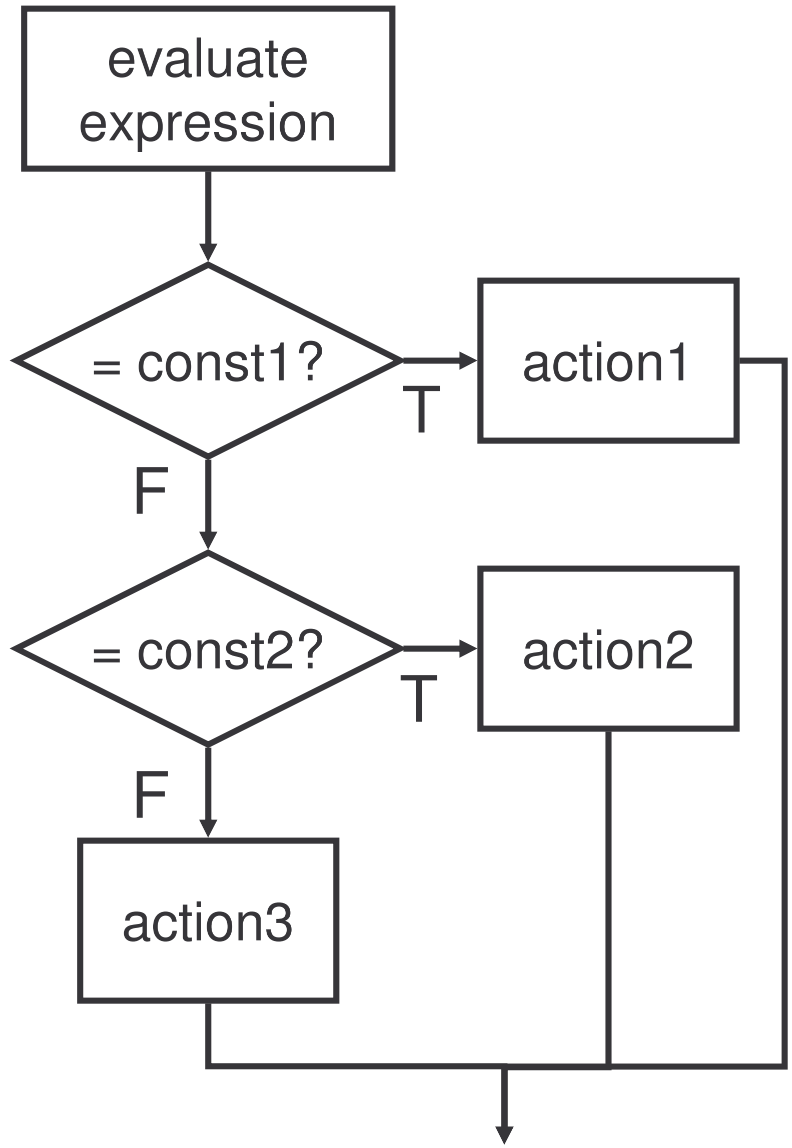
```
if (condition)
     action_if;
  else
     action_else;
```



Else allows choice between
two mutually exclusive actions without re-testing condition.

Switch

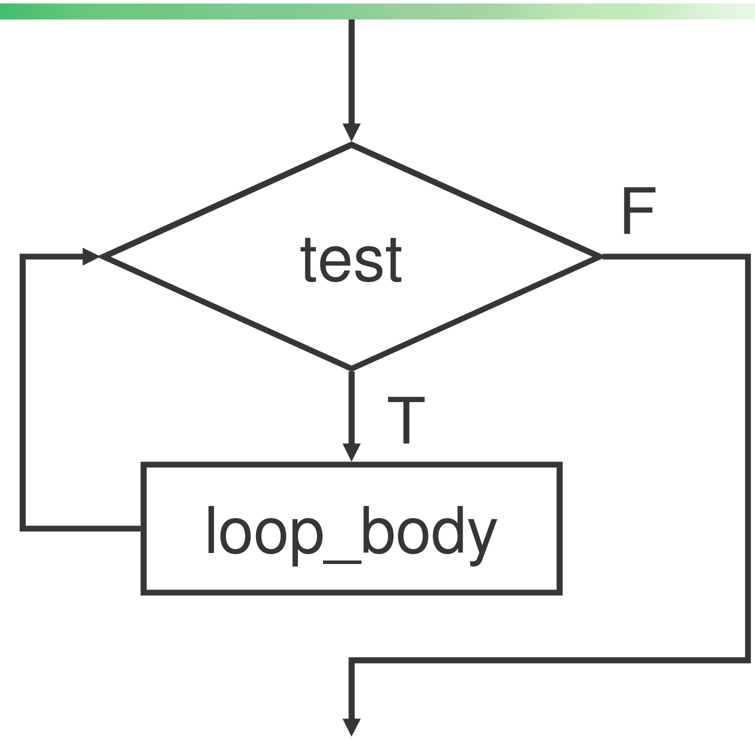
```
switch (expression) {
  case const1:
    action1; break;
  case const2:
    action2; break;
  default:
    action3;
}
```



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

While

```
while (test)
    loop_body;
```

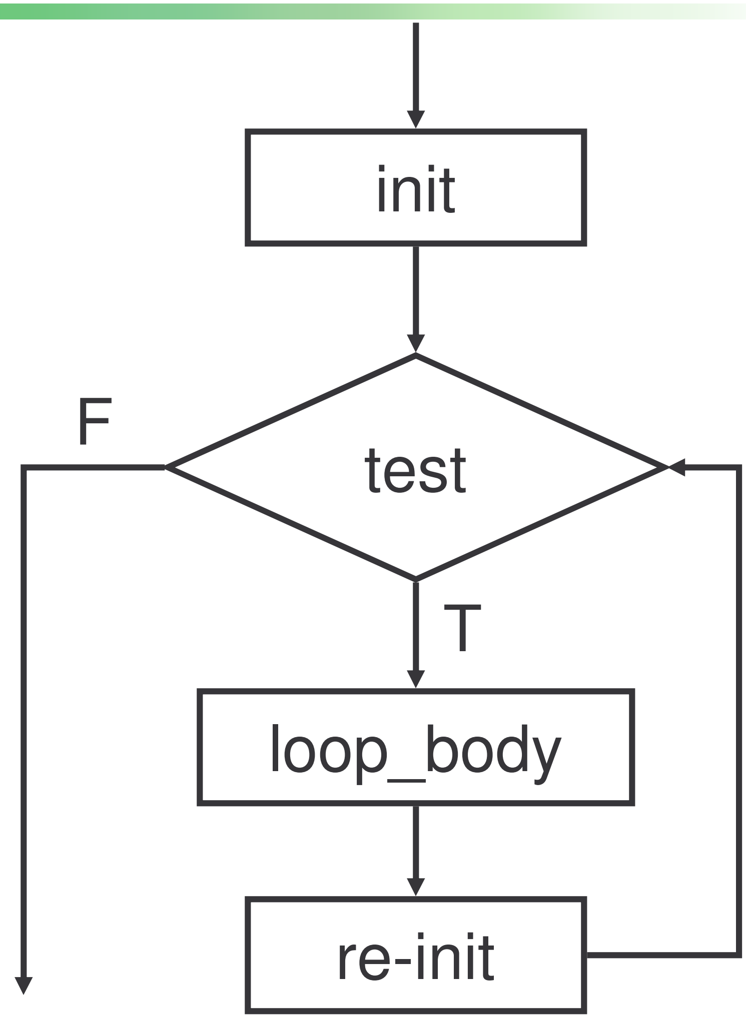


Executes loop body as long as test evaluates to TRUE (non-zero).

*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.*

ASCII Table

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

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.

```
        0101 0101 0101 0101
AND     0000 0000 0000 0001
        -------------------
        0000 0000 0000 0001
```

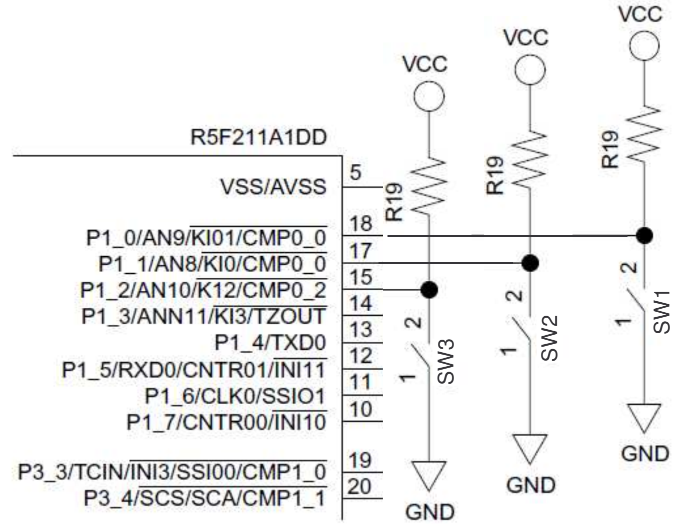
Or, lets say you want to look at bits 7 to 2:

```
        0101 0101 0101 0101
AND     0000 0000 1111 1100
        -------------------
        0000 0000 0101 0100
```

Code? Bitwise and is &, bitwise or is |

Code Example

Let's assume three switches connected to port 1 like the following:



How do you read the three switches?

After you set the direction:

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

All at the same time?

```
data = (int) PORT1.PORT.BYTE;
```

C examples

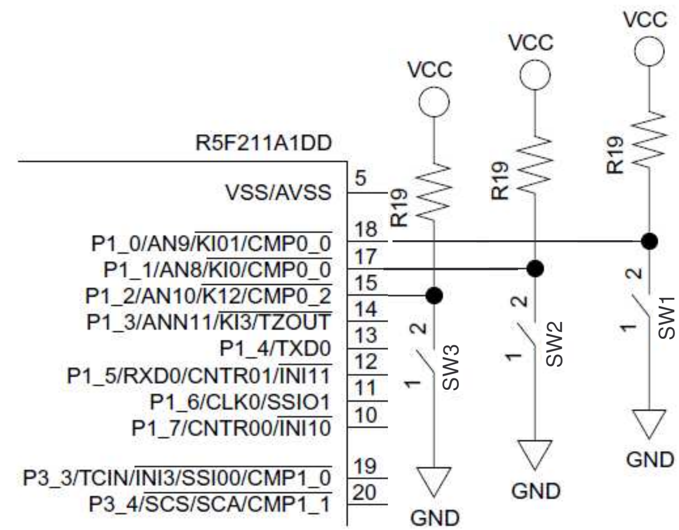
Now, write the C code to interrogate the switches and print

"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

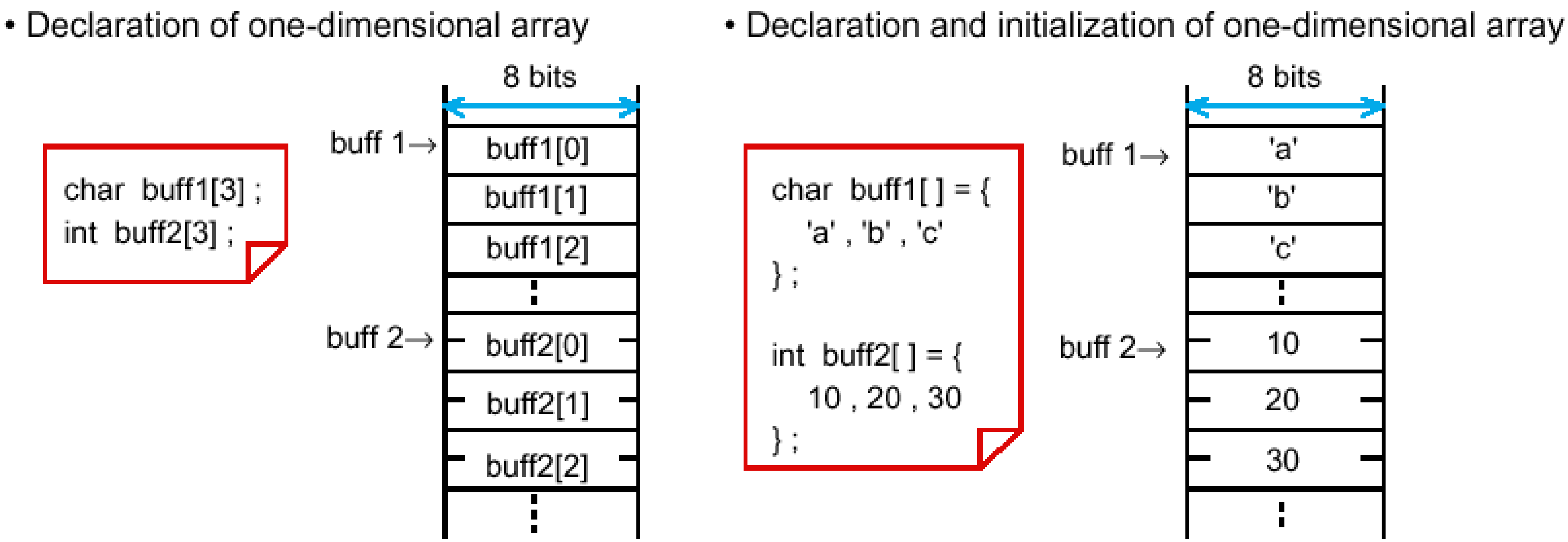


Figure 1.7.2 Declaration of one-dimensional array and memory mapping

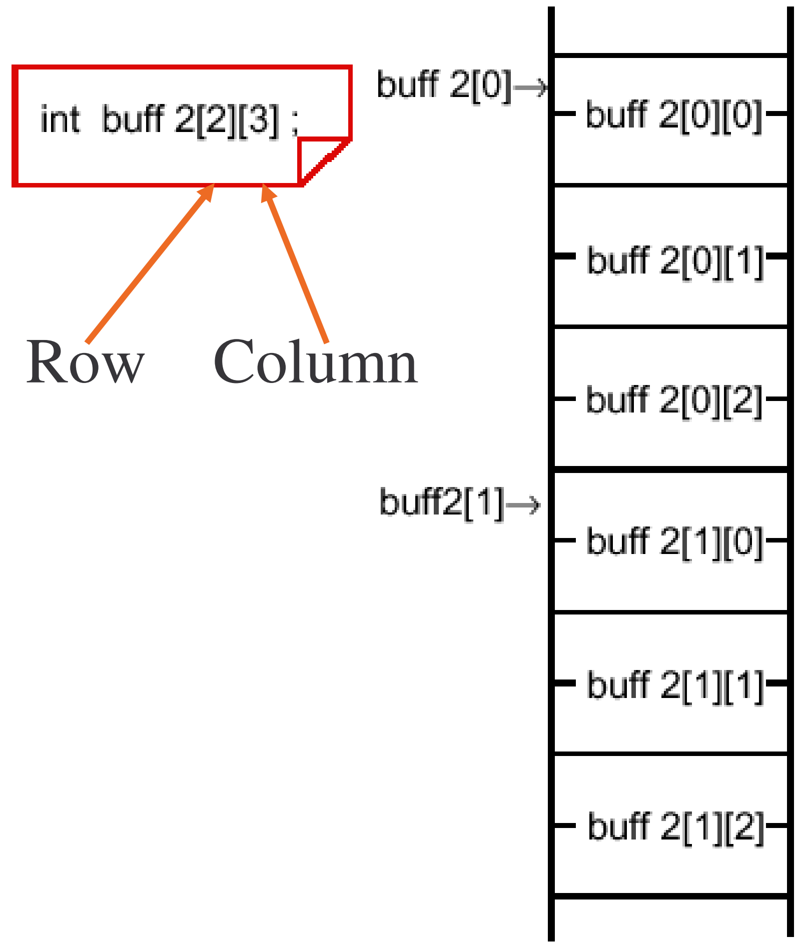
2D Arrays

Columns

Rows

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]

C arrays are stored in a *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**

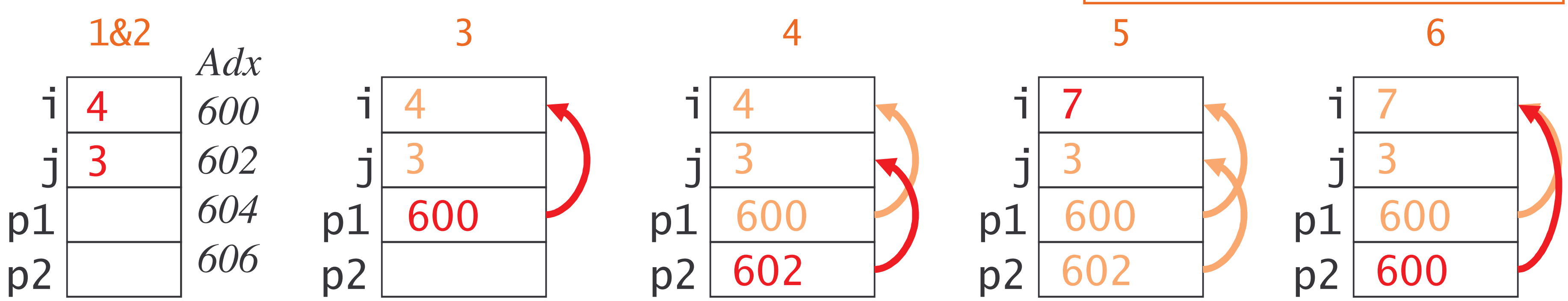
– address operator **&**

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*

```
  void main ( ) {
    int i, j;
    int *p1, *p2;
1   i = 4;
2   j = 3;
3   p1 = &i;
4   p2 = &j;
5   *p1 = *p1+*p2;
6   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
  - **p1++** corresponds to **p1 = p1 + sizeof(int);**
  - sizeof is C macro which returns size of type in bytes

```
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 */
```

Pre and post

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

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.

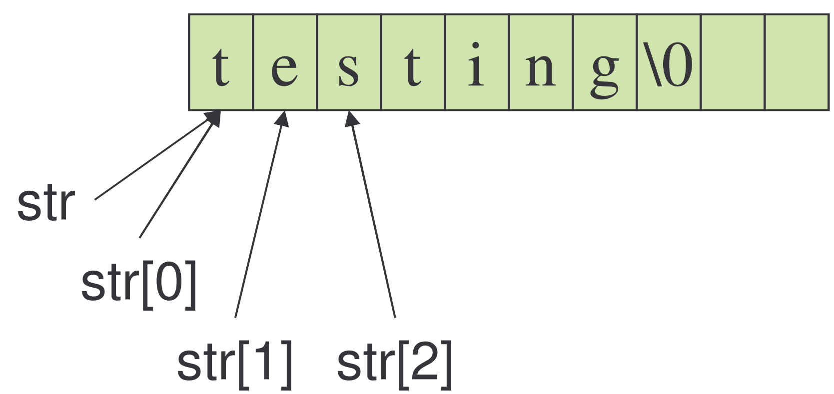
Instead an **array of characters** is used - *char a[44]*

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”;



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 * frmt, 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

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

s1: Testing

```
sprintf(s2, "a=%d, b=%d", a, b);
```

s2: a=5, b=10

```
sprintf(s1, "b=%x, c=%d", b, c);
```

s1: b=a, c=-30

```
sprintf(s1, "b=0x%x", b);
```

s1: b=0xa

```
sprintf(s2, "s1=%s", s1);
```

s2: s1=b=0xa

```
sprintf(s1, "%c %c", ch, s2);
```

s1: $ s

sprintf Examples – floating-point

Variation on %f format specifier

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

Examples

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

Examples

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

s1

442

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

s1

1

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

s1

1

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

s1

-011

String Operations in string.h

Copy **ct** to **s** including terminating null character. Returns a pointer to **s**.

- char* strcpy(char* s, const char* ct);

```
s1 = “cheese”;
s2 = “limburger”;
strcpy(s1, s2);  /* s1 = limburger */
```

Concatenate the characters of **ct** to **s**. Terminate **s** with the null character and return a pointer to it.

- 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.

- char* strncat(char* s, const char* ct, int n);

```
s1 = “cheese”;
s2 = “ puffs”;
strncat(s1, s2, 4);  /* cheese puf */
```

Compares two strings. The comparison stops on reaching a **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)

- int strcmp(const char* s1, const char* s2);

```
s1 = “cheese”;
s2 = “chases”;
strcmp(s1,s2); /* returns non-zero number */
strcmp(s1, “cheese”); /* returns zero */
```

More String Operations

Return pointer to first occurrence of **c** in **s1**, or **NULL** if not found.

- char* strchr(const char* s1, int c);

```
s1 = “Smeagol and Deagol”;
char a *;
a = strchr(s1, “g”); /* returns pointer to s1[4] */
```

Return pointer to last occurrence of **c** in **s1**, or **NULL** if not found.

- char* strrchr(const char* s1, int c);

```
s1 = “Smeagol and Deagol”;
char a *;
a = strrchr(s1, “a”); /* returns pointer to s1[14] */
```

Can use the returned pointer for other purposes

```
*a = ‘\0’; /* s1 = “Smeagol and De” */
strcat(s1, “spair”); /* s1 = “Smeagol and Despair” */
```

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

```
int * j, *k;
j = (int *) malloc (sizeof(int));
*j = 37;
```

- Request pointer to space for array of elements and initialize to zero

```
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

```
free(k);
```

Example Application: Voice Recorder

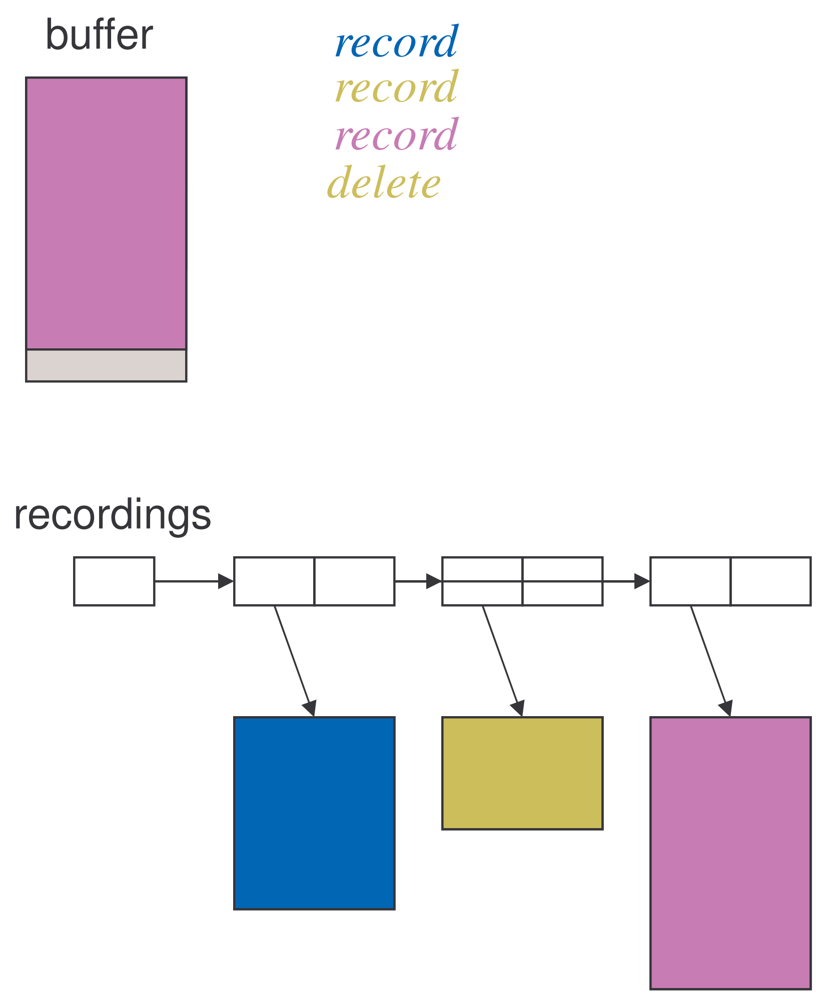
Recording

- 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

```
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();
   }
}
```

Code for handle_forward

```
void handle_forward(void) {
   if (cur_recording)
        cur_recording = cur_recording->Next;
   if (!cur_recording)
        cur_recording = recordings;
}
```

Code for handle_record

```
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

```
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);
}
```

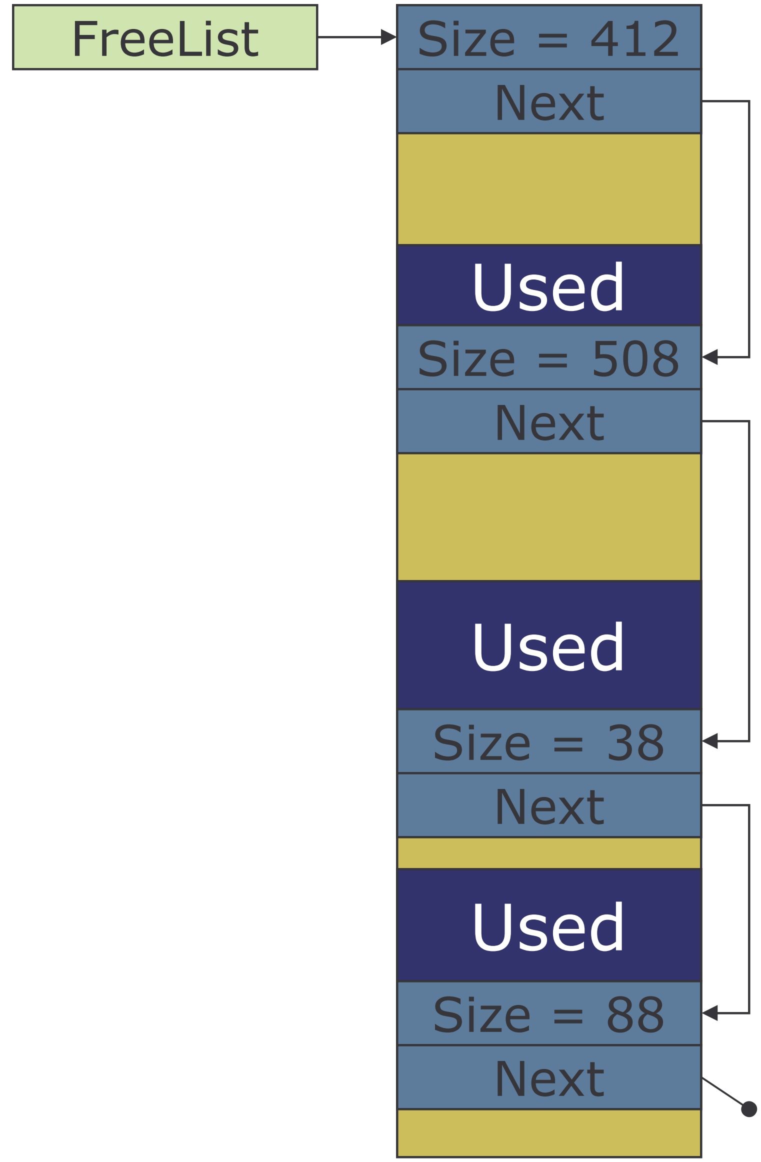
Allocation Data Structures

Keep free memory in sorted list of free blocks

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

Assume hdr takes no space for examples

More details in “Memory Allocation in C,” Leslie Alridge, *Embedded Systems Programming*, August 1989



Allocation Operations

To allocate memory

- find first block of size >= 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?

A formula? A solution? A sequence of steps? A recipe?

A former Vice-President? (Al-Gore-ithm?)

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:

x	y	z
5	4	
5	-4	
-5	4	
-5	-4	
5	5	

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