

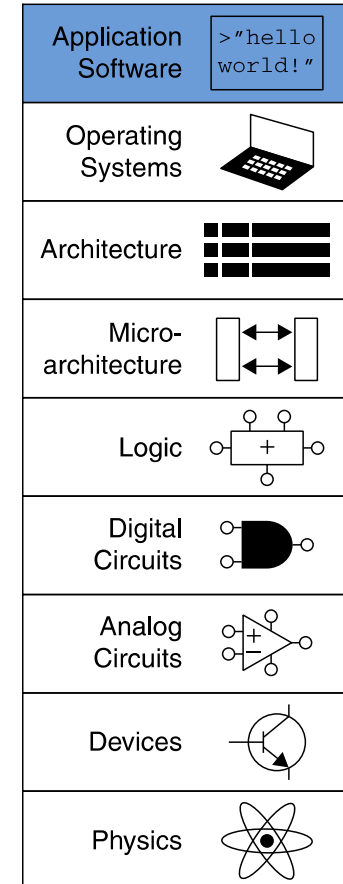
Digital Design & Computer Architecture

Sarah Harris & David Harris

Appendix C: C Programming

Appendix C :: Topics

- C Basics
- Functions
- Operators
- Control Flow
- Loops
- Arrays & Strings
- Structures
- Memory
- Pointers
- Dynamic Memory Allocation



Overview

- C programming language developed at Bell Labs around 1973
- Capable of controlling a computer to do nearly anything, including directly interacting with the hardware
- Suitable for generating high performance code
- Relatively easy to use
- Available from supercomputers to microcontrollers
- Closely related to other important languages including C++, C#, Objective C, Java, Arduino

C is Libertarian

- Lets you do just about anything
- Interacts directly with the hardware
- Does NOT protect you from your own stupidity
- Assumes YOU know the size of arrays and variables
- Unless sandboxed, can write ANYWHERE in memory

Example

```
// factorial.c
// David Harris@hmc.edu 22 October 2019

int fact(int n) {
    if (n <= 1) return 1;
    else return n*fact(n-1);
}

void main(void) {
    int result;
    result = fact(4);
}
```

Steps to C Programming

- Write code
- Compile code
- Execute code
- Debug code

Appendix C: C Programming

C Basics

Comments

- Single-line comments begin with “//” and continue to the end of the line.
`x += 2; //This is a single-line comment.`
- Multi-line comments begin with “/*” end with “*/”.
`/* You can hide or disable a section of code such as this block
with a multi-line comment
 x = bob ? x : y;
 y -= 5;
*/`
- Always start code with the file name, your name, email, and date. This gives you copyright ownership & helps the next programmer track you down.

Constants, Defines, or Macros

- Constants are named using the `#define` directive

```
#define MAXGUESSES 5
#define PI 3.14159
```
- The `#` indicates that this line in the program will be handled by the preprocessor.
- Before compilation, the preprocessor replaces each occurrence of the identifier `MAXGUESSES` in the program with `5`.
- By convention, `#define` lines are located at the top of the file and identifiers are written in all capital letters.

Global and Local Variables

- Global variables are declared outside of any function
 - Accessible from all functions
 - Often lead to hard-to-debug code
 - Should be avoided, especially in large programs
- Local variables are declared inside a function
 - Only accessible in that function
 - Should be your preferred choice

Primitive Data Types

Type	Size (bits)	Minimum	Maximum
char	8	$-2^{-7} = -128$	$2^7 - 1 = 127$
unsigned char	8	0	$2^8 - 1 = 255$
short	16	$-2^{15} = -32,768$	$2^{15} - 1 = 32,767$
unsigned short	16	0	$2^{16} - 1 = 65,535$
long	32	$-2^{31} = -2,147,483,648$	$2^{31} - 1 = 2,147,483,647$
unsigned long	32	0	$2^{32} - 1 = 4,294,967,295$
long long	64	-2^{63}	$2^{63} - 1$
unsigned long	64	0	$2^{64} - 1$
int	machine-dependent		
unsigned int	machine-dependent		
float	32	$\pm 2^{-126}$	$\pm 2^{127}$
double	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

Integer Sizes

- Integer sizes in C may vary with the machine
 - int may be 16 or 32 bits
 - long may be 32 or 64 bits
 - Best to use sized types if size truly matters
 - But their names are a bit cumbersome
 - `#include <stdint.h>`
- Signed: `int16_t, int32_t, int64_t`
- Unsigned: `uint16_t, uint32_t, uint64_t`

ASCII Table

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(88	58	1011000	130	X					
41	29	101001	51)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

<https://commons.wikimedia.org/wiki/File:ASCII-Table.svg>

Appendix C: C Programming

Functions

Functions

- A function may take some inputs and may return at most one output
- The type of the inputs is declared in the function declaration
- Functions pass variables by *value* not *reference*
- Curly braces {} enclose the body of the function, which may contain zero or more statements
- The type of returned value is declared in the function declaration
- The return statement indicates the value that the function should return to its caller
- A function must be either declared BEFORE it is used or a function prototype declared BEFORE it is used

Function Example

// Return the sum of the three input variables

```
int sum3(int a, int b, int c) {  
    int result = a + b + c;  
    return result;  
}
```


Function Prototypes

```
// sum3example.c
// David Harris@hmc.edu 22 October 2019

////////////////////////////////////
// Prototypes
////////////////////////////////////
int sum3(int, int, int); // needed because sum3 is called before declared

////////////////////////////////////
// main
////////////////////////////////////

void main(void) {
    int answer;
    answer = sum3(6, 7, 8);
}

////////////////////////////////////
// other functions
// prototype not needed if these were moved before main
////////////////////////////////////

int sum3(int a, int b, int c) {
    int result = a + b + c;
    return result;
}
```

Prototypes are Sometimes Unavoidable

```
// Prototypes needed for f1 and/or f2 because they  
// can't both be declared before each other
```

```
int f1(int);  
int f2(int);
```

```
int f1(int n) {  
    return f2(n-1) + 1;  
}
```

```
int f2(int n) {  
    return f1(n-1)*2;  
}
```

```
void main(void) {  
    int answer;  
    answer = f1(5);  
}
```

Includes

- The function prototypes for the standard libraries are included at the top of a file with the `#include` directive:

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

```
#include <math.h>
```

- Your own function prototypes (or anything else you want to include) is done with quotes instead of brackets for relative or absolute path:

e.g., `#include "other/myFuncs.h"`

Appendix C: C Programming

Operators

Boolean (True/False) in C

- A variable or expression is considered FALSE if its value is 0
- A variable is considered TRUE if it has any other value
 - 1, 42, and -1 are all TRUE for C
- Logical operators assign FALSE as 0 and TRUE as 1

Operators and Precedence

Category	Operator	Description	Example
Unary	++	post-increment	a++; // a = a+1
	--	post-decrement	x--; // x = x-1
	&	memory address of a variable	x = &y; // x = the memory // address of y
	~	bitwise NOT	z = ~a;
	!	Boolean NOT	!x
	-	negation	y = -a;
	++	pre-increment	++a; // a = a+1
	--	pre-decrement	--x; // x = x-1
	(type)	casts a variable to (type)	x = (int)c; // cast c to an // int and assign it to x
	sizeof()	size of a variable or type in bytes	long int y; x = sizeof(y); // x = 4

Operators Continued

Multiplicative	*	multiplication	<code>y = x * 12;</code>
	/	division	<code>z = 9 / 3; // z = 3</code>
	%	modulo	<code>z = 5 % 2; // z = 1</code>
Additive	+	addition	<code>y = a + 2;</code>
	-	subtraction	<code>y = a - 2;</code>
Bitwise Shift	<<	bitshift left	<code>z = 5 << 2; // z = 0b00010100</code>
	>>	bitshift right	<code>x = 9 >> 3; // x = 0b00000001</code>
Relational	==	equals	<code>y == 2</code>
	!=	not equals	<code>x != 7</code>
	<	less than	<code>y < 12</code>
	>	greater than	<code>val > max</code>
	<=	less than or equal	<code>z <= 2</code>
	>=	greater than or equal	<code>y >= 10</code>

Operators Continued

Table eC.3 Operators listed by decreasing precedence—Cont'd

Category	Operator	Description	Example
Bitwise	&	bitwise AND	<code>y = a & 15;</code>
	^	bitwise XOR	<code>y = 2 ^ 3;</code>
		bitwise OR	<code>y = a b;</code>
Logical	&&	Boolean AND	<code>x && y</code>
		Boolean OR	<code>x y</code>
Ternary	? :	ternary operator	<code>y = x ? a : b; // if x is TRUE, // y=a, else y=b</code>

Operators Continued

Assignment

=	assignment	<code>x = 22;</code>	
+=	addition and assignment	<code>y += 3;</code>	<code>// y = y + 3</code>
-=	subtraction and assignment	<code>z -= 10;</code>	<code>// z = z - 10</code>
*=	multiplication and assignment	<code>x *= 4;</code>	<code>// x = x * 4</code>
/=	division and assignment	<code>y /= 10;</code>	<code>// y = y / 10</code>
%=	modulo and assignment	<code>x %= 4;</code>	<code>// x = x % 4</code>
>>=	bitwise right-shift and assignment	<code>x >>= 5;</code>	<code>// x = x >> 5</code>
<<=	bitwise left-shift and assignment	<code>x <<= 2;</code>	<code>// x = x << 2</code>
&=	bitwise AND and assignment	<code>y &= 15;</code>	<code>// y = y & 15</code>
=	bitwise OR and assignment	<code>x = y;</code>	<code>// x = x y</code>
^=	bitwise XOR and assignment	<code>x ^= y;</code>	<code>// x = x ^ y</code>

Examples

```
int a = 42;
int b = 0x15;           // hexadecimal; = 21 in decimal
char c = 0b00001010;    // binary; = 10 in decimal

char d = !c;            // 0, because c was nonzero
char e = ~c;            // 0b11110101 bitwise NOT
char f = e | c;          // 0b11111111 bitwise OR
char g = c << 2;         // 0b00101000 shift left by 2
int h = (a > b);         // 1 because a is greater than b
int i = (a > b) && (c != e); // 1 because both are TRUE
int j = (a > b) ? a : b; // 42 because a > b
int k = sizeof(a);       // 4 on most computers
g &= c;                  // 0b00001000 bitwise AND
```

Appendix C: C Programming

Control Flow

Control Flow Statements

if

```
if (expression)  
    statement;
```

if/else

```
if (expression)  
    statement1;  
else  
    statement2;
```

switch/case

```
switch (variable) {  
    case (expression1): statement1; break;  
    case (expression2): statement2; break;  
    case (expression3): statement3; break;  
    default: statement4;  
}
```

Don't forget "break" or "default"

If example

```
if (n <= 1) return 1;
```

Compound Statements

- When a statement has more than one line, enclose it in {}

```
if (answer == 42) {  
    ultimateQuestion = 1;  
    hitchhikersGuide = 1;  
}
```

If/else example

```
if (n <= 1) return 1;  
else      return fact(n-1);
```

Switch/case example

```
switch (state) {  
    case (0): if (ta) state = 0; else state = 1; break;  
    case (1): state = 2; break;  
    case (2): if (tb) state = 2; else state = 3; break;  
    case (3): state = 0; break;  
    default: state = 0;  
}
```


Appendix C: C Programming

Loops

Loops

while

```
while (condition)  
    statement;
```

do/while

```
do {  
    statement;  
} while (condition);
```

for

```
for (initialization; condition; loop operation)  
    statement;
```

While example

```
int fact(int n) {  
    int result = 1;  
    while (n > 1) {  
        result = result * n; // or write result *= n;  
        n = n - 1;           // or write n--  
    }  
    return result;  
}
```

// Alternative while loop is shorter but less clear

```
int fact(int n) {  
    int result = 1;  
    while (n > 1) result *= n--;  
    return result;  
}
```

Do/while example

```
int fact(int n) {  
    int result = 1;  
    do {  
        result *= n;  
    } while (n-- > 1);  
    return result;  
}
```

- Do always executes the statement at least once.
- Longer and not preferred for this example

For example

```
int fact(int n) {  
    int result = 1;  
    int i;  
  
    for (i=1; i <= n; i++)  
        result *= i;  
    return result;  
}
```

- First do initialization ($i = 1$)
- Then check condition ($i \leq n$)
 - If satisfied, do body ($\text{result} *= i$)
 - Then do loop operation ($i++$)
- Then repeat from checking condition

Appendix C: C Programming

Arrays & Strings

Data Types: Arrays

- Array contains multiple elements

```
float accel[3];
```

- The elements are numbered from 0 to N-1, where N is the length of the array
- Initialize your arrays.
 - An uninitialized array can contain anything

- Arrays can be multidimensional

```
#define NUMSTUDENTS 120  
#define NUMLABS 11  
int grades[NUMSTUDENTS][NUMLABS];
```

Array Example

```
#include <math.h>
```

```
double mag(double v[3]) {  
    return sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);  
}
```


Data Types: Strings

- A string is an array of characters
- Last entry is zero to indicate end ("NULL terminated")

```
char name[20] = "BOB";
```

- Stored as:

```
name[0] = 66; // ASCII value for B
```

```
name[1] = 79; // ASCII value for O
```

```
name[2] = 66; // ASCII value for B
```

```
name[3] = 0; // NULL termination
```

other entries are junk, ignored

Examples: String Handling

```
#define MAXLEN 80

int strlen(char str[]) {
    int len=0;

    while (str[len] && len < MAXLEN) len++;
    return len;
}

void strcpy(char dest[], char src[]) {
    int i = 0;

    do {
        dest[i] = src[i];
    } while (src[i++] && i < MAXLEN);
}
```

Examples: Using Strings

```
#include <string.h>
#define MAXLEN 80

void main(void) {
    char name[80];
    int len;
    char c;

    strcpy(name, "BOB"); // copy BOB into name
    len = strlen(name);  // len = 3
    c = name[1];         // c = 'O' (79)
}
```

Appendix C: C Programming

Structures

Structures

- Store a collection of related information
- General format:

```
struct name {  
    type1 element1;  
    type2 element2;  
    ...  
};
```

Structures

```
struct contact {  
    char name[30];  
    int age;  
    float height; // in meters  
};
```

```
struct contact c1;  
strcpy(c1.name, "Ben Bitdiddle");  
c1.age = 20;  
c1.height = 1.82;
```

Typedef

- If you're using lots of the same structure, you can shorten your typing by using typedef.
- `typedef type name;`

```
typedef struct contact {  
    char name[30];  
    int age;  
    float height; // in meters  
} contact; // defines contact as shorthand for "struct contact"
```

```
contact c1; // now we can declare the variable as type contact
```

Structure Examples

```
typedef struct point {  
    int x;  
    int y;  
} point;
```

```
point p1;  
p1.x = 42; p1.y = 9;
```

```
typedef struct rect {  
    point ll;  
    point ur;  
    int color;  
} rect;
```

```
rect r1;  
r1.color = 1;  
r1.ll = p1;  
r1.ur.x = r1.ll.x + width;  
r1.ur.y = r1.ll.y + height;
```


Appendix C: C Programming

Memory

Memory

- Variables are stored in memory
- Each primitive data type has a size
 - char 1 byte
 - short at least 2 bytes
 - long at least 4 bytes, 8 on some 64-bit computers
 - int at least 2 bytes, 4 on most 32 & 64-bit computers
 - float 4 bytes
 - double 8 bytes
- Arrays & structs stored in multiple consecutive locations

Sizeof

- Sizeof operator returns size of a datatype

```
char c;  
double d;  
point p;  
rect r;  
int s1 = sizeof c;    // s1 = 1  
int s2 = sizeof(d);   // s2 = 8  
int s3 = sizeof(p);   // s3 = 4 + 4 = 8  
int s4 = sizeof(r);   // s4 = 8 + 8 + 4 = 20
```

Memory Example: Array

C Code Example eC.21 ARRAY INITIALIZATION AT DECLARATION USING {}

```
long scores[3]={93, 81, 97}; // scores[0]=93; scores[1]=81; scores[2]=97;
```

Address (Byte #)	Data	Variable Name
0x4B		
0x4A		
0x49	97	
0x48		
0x47		
0x46	81	
0x45		
0x44		
0x43		
0x42	93	
0x41		
0x40		

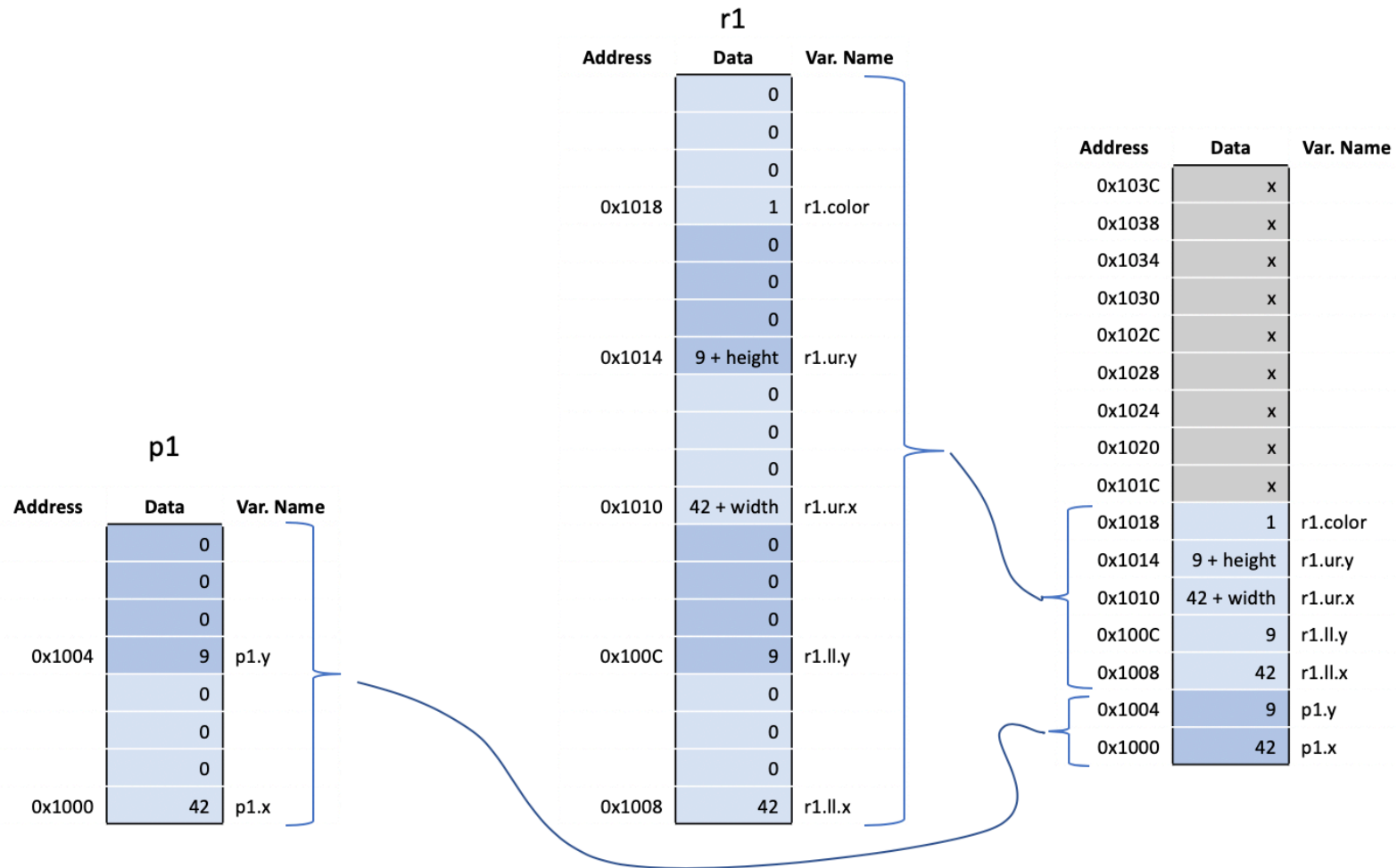
Memory

Address (Byte #)	Data	Variable Name
0x4B	0x00	
0x4A	0x00	
0x49	0x00	
0x48	0x61	
0x47	0x00	
0x46	0x00	
0x45	0x00	
0x44	0x51	
0x43	0x00	
0x42	0x00	
0x41	0x00	
0x40	0x5D	

Memory

Figure eC.4 scores array stored in memory

Memory Example: Structure



Appendix C: C Programming

Pointers

Pointers

- A pointer is an address in memory
- Pointer variables are declared with `*` and a data type to which the pointer points

```
int salary1, salary2;
```

```
int *ptr;    // a pointer to an integer
```

- `&` returns address of a variable

```
salary1 = 98500;    // suppose this is at address 100 in memory
```

```
ptr = &salary1;    // ptr contains 100 (the address of salary1)
```

- `*` dereferences a pointer (finds value it points to)

```
salary2 = *ptr + 1000; // salary2 gets 99500
```

Arrays and Pointers

- An array in C is viewed as the address of the zeroth element
- Equivalent to a pointer to the beginning of the array

Pointer Example

→ Now add:
int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028
int a = 37, b;
int *ptr;
int i;

Address	Data	Var. Name
0x103C	x	
0x1038	x	
0x1034	x	
0x1030	x	
0x102C	x	
0x1028	x	
0x1024	x	
0x1020	x	
0x101C	x	
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Pointer Example

Now add:

→ `int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028`

`int a = 37, b;`

`int *ptr;`

`int i;`

Address	Data	Var. Name
0x103C	x	
0x1038	x	
0x1034	x	
0x1030	x	
0x102C	x	
0x1028	x	ary[3]
0x1024	x	ary[2]
0x1020	x	ary[1]
0x101C	x	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Pointer Example

Now add:

int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028

→ int a = 37, b; // suppose at addresses 0x102C, 0x1030

int *ptr;

int i;

Address	Data	Var. Name
0x103C	x	
0x1038	x	
0x1034	x	
0x1030	x	b
0x102C	37	a
0x1028	x	ary[3]
0x1024	x	ary[2]
0x1020	x	ary[1]
0x101C	x	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Pointer Example

Now add:

```
int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028
```

```
int a = 37, b; // suppose at addresses 0x102C, 0x1030
```

→

```
int *ptr; // suppose ptr is at address 0x1034, initially undefined
```

```
int i;
```

Address	Data	Var. Name
0x103C	x	
0x1038	x	
0x1034	x	ptr
0x1030	x	b
0x102C	37	a
0x1028	x	ary[3]
0x1024	x	ary[2]
0x1020	x	ary[1]
0x101C	x	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Pointer Example

Now add:

```
int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028
```

```
int a = 37, b; // suppose at addresses 0x102C, 0x1030
```

```
int *ptr; // suppose ptr is at address 0x1034, initially undefined
```

→ `int i;`

Address	Data	Var. Name
0x103C	x	
0x1038	x	i
0x1034	x	ptr
0x1030	x	b
0x102C	37	a
0x1028	x	ary[3]
0x1024	x	ary[2]
0x1020	x	ary[1]
0x101C	x	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Pointer Example

Now add:

```
int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028
```

```
int a = 37, b; // suppose at addresses 0x102C, 0x1030
```

```
int *ptr; // suppose ptr is at address 0x1034, initially undefined
```

```
int i;
```

```
for (i=0; i<3; i++) ary[i] = i*i;
```

```
ptr = &a;
```

```
b = *ptr;
```

```
*ptr = 3;
```

```
ptr = ary;
```

```
ptr[1] = b;
```

```
*(ptr+2) = 7;
```

```
ary[4] = 1;
```

```
*(ptr+5) = 2;
```

Address	Data	Var. Name
0x103C	x	
0x1038	x	i
0x1034	x	ptr
0x1030	x	b
0x102C	37	a
0x1028	x	ary[3]
0x1024	x	ary[2]
0x1020	x	ary[1]
0x101C	x	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Pointer Example

Now add:

```
int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028
```

```
int a = 37, b; // suppose at addresses 0x102C, 0x1030
```

```
int *ptr; // suppose ptr is at address 0x1034, initially undefined
```

```
int i;
```

→ **for (i=0; i<3; i++) ary[i] = i*i; // Note: ary[3] not changed**

```
ptr = &a;
```

```
b = *ptr;
```

```
*ptr = 3;
```

```
ptr = ary;
```

```
ptr[1] = b;
```

```
*(ptr+2) = 7;
```

```
ary[4] = 1;
```

```
*(ptr+5) = 2;
```

Address	Data	Var. Name
0x103C	x	
0x1038	3	i
0x1034	x	ptr
0x1030	x	b
0x102C	37	a
0x1028	x	ary[3]
0x1024	4	ary[2]
0x1020	1	ary[1]
0x101C	0	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Pointer Example

Now add:

```
int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028
```

```
int a = 37, b; // suppose at addresses 0x102C, 0x1030
```

```
int *ptr; // suppose ptr is at address 0x1034, initially undefined
```

```
int i;
```

```
for (i=0; i<3; i++) ary[i] = i*i;
```

```
ptr = &a; // ptr = 0x102C
```

```
b = *ptr;
```

```
*ptr = 3;
```

```
ptr = ary;
```

```
ptr[1] = b;
```

```
*(ptr+2) = 7;
```

```
ary[4] = 1;
```

```
*(ptr+5) = 2;
```

Address	Data	Var. Name
0x103C	x	
0x1038	3	i
0x1034	0x102C	ptr
0x1030	x	b
0x102C	37	a
0x1028	x	ary[3]
0x1024	4	ary[2]
0x1020	1	ary[1]
0x101C	0	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Pointer Example

Now add:

```
int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028
```

```
int a = 37, b; // suppose at addresses 0x102C, 0x1030
```

```
int *ptr; // suppose ptr is at address 0x1034, initially undefined
```

```
int i;
```

```
for (i=0; i<3; i++) ary[i] = i*i;
```

```
ptr = &a; // ptr = 0x102C
```

```
→ b = *ptr; // dereference pointer, b = 37
```

```
*ptr = 3;
```

```
ptr = ary;
```

```
ptr[1] = b;
```

```
*(ptr+2) = 7;
```

```
ary[4] = 1;
```

```
*(ptr+5) = 2;
```

Address	Data	Var. Name
0x103C	x	
0x1038	3	i
0x1034	0x102C	ptr
0x1030	37	b
0x102C	37	a
0x1028	x	ary[3]
0x1024	4	ary[2]
0x1020	1	ary[1]
0x101C	0	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Pointer Example

Now add:

```
int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028
```

```
int a = 37, b; // suppose at addresses 0x102C, 0x1030
```

```
int *ptr; // suppose ptr is at address 0x1034, initially undefined
```

```
int i;
```

```
for (i=0; i<3; i++) ary[i] = i*i;
```

```
ptr = &a; // ptr = 0x102C
```

```
b = *ptr; // dereference pointer, b = 37
```

```
→ *ptr = 3; // a = 3
```

```
ptr = ary;
```

```
ptr[1] = b;
```

```
*(ptr+2) = 7;
```

```
ary[4] = 1;
```

```
*(ptr+5) = 2;
```

Address	Data	Var. Name
0x103C	x	
0x1038	3	i
0x1034	0x102C	ptr
0x1030	37	b
0x102C	3	a
0x1028	x	ary[3]
0x1024	4	ary[2]
0x1020	1	ary[1]
0x101C	0	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Pointer Example

Now add:

```
int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028
```

```
int a = 37, b; // suppose at addresses 0x102C, 0x1030
```

```
int *ptr; // suppose ptr is at address 0x1034, initially undefined
```

```
int i;
```

```
for (i=0; i<3; i++) ary[i] = i*i;
```

```
ptr = &a; // ptr = 0x102C
```

```
b = *ptr; // dereference pointer, b = 37
```

```
*ptr = 3; // a = 3
```

```
ptr = ary; // ptr = 0x101C
```

```
ptr[1] = b;
```

```
*(ptr+2) = 7;
```

```
ary[4] = 1;
```

```
*(ptr+5) = 2;
```

Address	Data	Var. Name
0x103C	x	
0x1038	3	i
0x1034	0x101C	ptr
0x1030	37	b
0x102C	3	a
0x1028	x	ary[3]
0x1024	4	ary[2]
0x1020	1	ary[1]
0x101C	0	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Pointer Example

Now add:

```
int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028
int a = 37, b; // suppose at addresses 0x102C, 0x1030
int *ptr; // suppose ptr is at address 0x1034, initially undefined
int i;
```

```
for (i=0; i<3; i++) ary[i] = i*i;
ptr = &a; // ptr = 0x102C
b = *ptr; // dereference pointer, b = 37
*ptr = 3; // a = 3
ptr = ary; // ptr = 0x101C
→ ptr[1] = b; // ary[1] = 37
*(ptr+2) = 7;
ary[4] = 1;
*(ptr+5) = 2;
```

Address	Data	Var. Name
0x103C	x	
0x1038	3	i
0x1034	0x101C	ptr
0x1030	37	b
0x102C	3	a
0x1028	x	ary[3]
0x1024	4	ary[2]
0x1020	37	ary[1]
0x101C	0	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Pointer Example

Now add:

```
int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028
int a = 37, b; // suppose at addresses 0x102C, 0x1030
int *ptr; // suppose ptr is at address 0x1034, initially undefined
int i;
```

```
for (i=0; i<3; i++) ary[i] = i*i;
ptr = &a; // ptr = 0x102C
b = *ptr; // dereference pointer, b = 37
*ptr = 3; // a = 3
ptr = ary; // ptr = 0x101C
ptr[1] = b; // ary[1] = 37
→ *(ptr+2) = 7; // ary[2] = 7, note offset is in int sizes, not bytes
ary[4] = 1;
*(ptr+5) = 2;
```

Address	Data	Var. Name
0x103C	x	
0x1038	3	i
0x1034	0x101C	ptr
0x1030	37	b
0x102C	3	a
0x1028	x	ary[3]
0x1024	7	ary[2]
0x1020	37	ary[1]
0x101C	0	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Pointer Example

Now add:

```
int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028
int a = 37, b; // suppose at addresses 0x102C, 0x1030
int *ptr; // suppose ptr is at address 0x1034, initially undefined
int i;
```

```
for (i=0; i<3; i++) ary[i] = i*i;
ptr = &a; // ptr = 0x102C
b = *ptr; // dereference pointer, b = 37
*ptr = 3; // a = 3
ptr = ary; // ptr = 0x101C
ptr[1] = b; // ary[1] = 37
*(ptr+2) = 7; // ary[2] = 7, note offset is in int sizes, not bytes
→ ary[4] = 1; // a = 1, BAD: trash variable past end of array
*(ptr+5) = 2;
```

Address	Data	Var. Name
0x103C	x	
0x1038	3	i
0x1034	0x101C	ptr
0x1030	37	b
0x102C	1	a
0x1028	x	ary[3]
0x1024	7	ary[2]
0x1020	37	ary[1]
0x101C	0	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Pointer Example

Now add:

```
int ary[4]; // suppose at addresses 0x101C, 0x1020, 0x1024, 0x1028
int a = 37, b; // suppose at addresses 0x102C, 0x1030
int *ptr; // suppose ptr is at address 0x1034, initially undefined
int i;
```

```
for (i=0; i<3; i++) ary[i] = i*i;
ptr = &a; // ptr = 0x102C
b = *ptr; // dereference pointer, b = 37
*ptr = 3; // a = 3
ptr = ary; // ptr = 0x101C
ptr[1] = b; // ary[1] = 37
*(ptr+2) = 7; // ary[2] = 7, note offset is in int sizes, not bytes
ary[4] = 1; // a = 1, BAD: trash variable past end of array
→ *(ptr+5) = 2; // b = 2, BAD: trash variable past end of array
```

Address	Data	Var. Name
0x103C	x	
0x1038	3	i
0x1034	0x101C	ptr
0x1030	2	b
0x102C	1	a
0x1028	x	ary[3]
0x1024	7	ary[2]
0x1020	37	ary[1]
0x101C	0	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Another Example

C Code

```
#include <stdio.h>

int main (void)
{
    char age = 30;
    char *p;
    p = &age;
    printf("age = %d\n", age);
    printf("p = %p\n", p);
    printf("**p = %d\n", *p);
    printf("sizeof(age) = %ld\n", sizeof(age));
    printf("sizeof(p) = %ld\n", sizeof(p));
    *p = 40;
    printf("**p = %d\n", *p);
    printf("age = %d\n", age);
    return 0;
}
```

Program Output

```
age = 30
p = 0x7ffee31
*p = 30
sizeof(age) = 1
sizeof(p) = 8
*p = 40
age = 40
```


Another Example

C Code

```
#include <stdio.h>

int main (void)
{
    char age = 30;
    char *p;
    p = &age;
    printf("age = %d\n", age);
    printf("p = %p\n", p);
    printf("**p = %d\n", *p);
    printf("sizeof(age) = %ld\n", sizeof(age));
    printf("sizeof(p) = %ld\n", sizeof(p));
    *p = 40;
    printf("**p = %d\n", *p);
    printf("age = %d\n", age);
    return 0;
}
```

Program Output

```
age = 30
p = 0x7ffee311e82b
*p = 30
sizeof(age) = 1
sizeof(p) = 8
*p = 40
age = 40
```

Pointers and Structures

```
rect *rptr; // Let rptr know it's pointing to a rect  
rptr = &r1; // Have rptr point at r1
```

```
(*rptr).color = 3; // Change r1.color to 3  
rptr->color = 4;   // Change r1.color to 4
```

```
// Use dot "." when you are using the structure name.  
// Arrow "->" (member access operator) is preferred when you are using the  
// pointer.
```

Address	Data	Var. Name
0x103C	x	
0x1038	3	i
0x1034	0x101C	ptr
0x1030	2	b
0x102C	1	a
0x1028	x	ary[3]
0x1024	7	ary[2]
0x1020	37	ary[1]
0x101C	0	ary[0]
0x1018	1	r1.color
0x1014	9 + height	r1.ur.y
0x1010	42 + width	r1.ur.x
0x100C	9	r1.ll.y
0x1008	42	r1.ll.x
0x1004	9	p1.y
0x1000	42	p1.x

Appendix C: C Programming

Memory Odds & Ends

Passing Structures to Functions

Complex data structures and arrays are normally passed to C programs by address rather than copied; it's more efficient.

```
void createRect(int xl, int yl, int width, int height, int color, rect *r) {  
    r->ll.x = xl; r->ll.y = yl;  
    r->ur.x = xl + width; r->ur.y = yl + height;  
    r->color = color;  
}  
  
int main(void) {  
    rect r1;  
    createRect(3, 5, 10, 20, 1, &r1);  
}
```

Multidimensional Arrays

- Stored in consecutive addresses
 - last dimension first

```
double field[2][3][3];
```

Address0	Entry
0x1068	field[1][2][2]
0x1060	field[1][2][1]
0x1068	field[1][2][0]
0x1060	field[1][1][2]
0x1068	field[1][1][1]
0x1060	field[1][1][0]
0x1068	field[1][0][2]
0x1060	field[1][0][1]
0x1068	field[1][0][0]
0x1060	field[0][2][2]
0x1068	field[0][2][1]
0x1060	field[0][2][0]
0x1068	field[0][1][2]
0x1060	field[0][1][1]
0x1058	field[0][1][0]
0x1050	field[0][0][2]
0x1048	field[0][0][1]
0x1040	field[0][0][0]

Complex Structures in Memory

```
typedef struct foo {  
    double d[4][5];  
    unsigned short s[16];  
} foo;  
  
foo z[10];  
int s5 = sizeof(z[0]);  
// 8*4*5 + 2*16 = 192 = 0xC0  
int s5 = sizeof(z);  
// 10*192 = 1920 = 0x780
```

Address	Entry
0x277E	z[9].s[15]
..	...
0x217E	z[1].s[15]
..	...
0x20C0	z[1].d[0][0]
0x20BE	z[0].s[15]
...	...
0x20A2	z[0].s[1]
0x20A0	z[0].s[0]
0x2098	z[0].d[3][4]
...	...
0x2008	z[0].d[0][1]
0x2000	z[0].d[0][0]

Appendix C: C Programming

Dynamic Memory Allocation

Memory Allocation

- malloc returns a pointer to allocated memory of a certain number of bytes.
- free frees this memory.
- These functions are declared in stdlib
- `int *ary = (int*)malloc(10*sizeof(int));`

Example: Variable Sized Arrays

- In standard C, multidimensional array sizes must be declared at compile time.
- Treat variable-sized M row x N column array as 1-dimensional array of M x N entries

Variable Dimension Matrix Example

```
#include <stdlib.h> // for malloc
```

```
double* newMatrix(int m, int n) {  
    double *mat;
```

```
    mat = (double*)malloc(m*n*sizeof(double));  
    return mat;  
}
```

```
double* newIdentityMatrix(int n) {  
    double *mat = newMatrix(n, n);  
    int i, j;
```

```
    for (i=0; i<n; i++)  
        for (j=0; j<n; j++)  
            mat[j+i*n] = (i==j);  
    return mat;  
}
```

Variable Dimension Matrix Example

```
void scaleMatrix(double *mat, double *scaled, int m, int n, double c) {  
    int i, j;  
  
    for (i=0; i<m; i++)  
        for (j=0; j<n; j++)  
            scaled[j+i*n] = mat[j+i*n]*c;  
}
```

```
int main(void) {  
    double *m1, *m2;  
  
    m1 = newIdentityMatrix(3);  
    m2 = newMatrix(3, 3);  
    scaleMatrix(m1, m2, 3, 3, 10);  
    free(m1);  
}
```

About these Notes

Digital Design and Computer Architecture Lecture Notes

© 2021 Sarah Harris and David Harris

These notes may be used and modified for educational and/or non-commercial purposes so long as the source is attributed.