The C Language And Some Extensions

18-613: Foundations of Computer Systems

3rd Additional Lecture, Jan. 31, 2019

Instructor:
Franz Franchetti

Scope of C in 18-613

- **Bootcamp: Linux and Windows**
- Basic data types and functions
- Structures, arrays, etc.
- C Standard Library
- Linux/POSIX Libraries
- Language Extensions
- Final thoughts

Linux and Windows

- **Linux Bootcamp:**
  http://www.cs.cmu.edu/afs/cs/academic/class/15213-s19/www/activities/linux-bootcamp/linux-bootcamp.pdf
- **Navigate and work on the shark machines**
  `vim`, `emacs`, `nano`, `ls`, `mkdir`, `cd`, `ps`,…
- **Windows Setup:**
  http://www.cs.cmu.edu/~213/activities/linux-bootcamp/windows-setup.pdf
- **Windows SSH, SFTP, public key setup**
  Putty, Tectia, Bitvise
- **GIT Client:** TortoiseGIT
- **Remote Editor: Visual Studio Code + sftp**
- **Xterm/X-Windows:** X-Win32 **or** MobaXterm
- Cisco AnyConnect VPN
- OpenAFS
- **See Windows section at**
  http://www.cs.cmu.edu/afs/cs/academic/class/15213-s19/www/faq.html

Scope of C in 18-613

- Bootcamp: Linux and Windows
- **Basic data types and functions**
- Structures, arrays, etc.
- C Standard Library
- Linux/POSIX Libraries
- Language Extensions
- Final thoughts

C For Non-C Programmers

- **You need to be firm with C in 18-613**
  https://fresh2refresh.com/c-programming/
  https://www.youtube.com/watch?v=KJgsSFOSQv0
  https://www.edx.org/course/c-programming-language-foundations
  https://www.edx.org/course/c-programming-advanced-data-types
- **Remember the main differences from your favorite language**
  - Strings, arrays, multidimensional arrays
  - Arithmetic and logic and/or/xor/not
  - Type casting/type checking, scoping, Not object oriented
  - (No) array bounds checking
  - Pointers, function pointers, composite data structures, void pointers
  - Typedefs, structs and unions
  - Malloc/free, File I/O, keyboard I/O
  - C preprocessor/macros
  - Standard libraries and POSIX libraries

Basic Data Types

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
pointer	4	8	8

Remember: unexpected size guarantees

Bit-Level Operations in C

- **Operations &, |, ~, ^ Available in C**
  - Apply to any "integral" data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (Char data type)**
  - ~0x41 → 0xBE
    - ~0100 0001_2 → 1011 1110_2
  - ~0x00 → 0xFF
    - ~0000 0000_2 → 1111 1111_2
  - 0x69 & 0x55 → 0x41
    - 0110 1001_2 & 0101 0101_2 → 0100 0001_2
  - 0x69 | 0x55 → 0x7D
    - 0110 1001_2 | 0101 0101_2 → 0111 1101_2

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Contrast: Logic Operations in C

- **Contrast to Bit-Level Operators**
  - **Logic Operations: &&, ||, !**
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination
- **Examples (char data type)**
  - !0x41 → 0x00
  - !0x00 → 0x01
  - !!0x41→ 0x01

  - 0x69 && 0x55 → 0x01
  - 0x69 || 0x55 → 0x01
  - p && *p (avoids null pointer access)

Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have “U” as suffix

    `0U, 4294967259U`
- **Casting**
  - Explicit casting between signed & unsigned same as U2T and T2U

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

  - Implicit casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty;
```

```
int fun(unsigned u);
uy = fun(tx);
```

Why Should I Use Unsigned? (cont.)

- ***Do*** **Use When Performing Modular Arithmetic**
  - Multiprecision arithmetic
- ***Do*** **Use When Using Bits to Represent Sets**
  - Logical right shift, no sign extension
- ***Do*** **Use In System Programming**
  - Bit masks, device commands,…

Casting Surprises

- **Expression Evaluation**
  - If there is a mix of unsigned and signed in single expression, ***signed values implicitly cast to unsigned***
  - Including comparison operations **<, >, ==, <=, >=**
  - Examples for *W* = 32: **TMIN = -2,147,483,648 , TMAX = 2,147,483,647**

Constant$_1$	**Constant$_2$**	**Relation**	**Evaluation**
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

Integer C Puzzles

Initialization

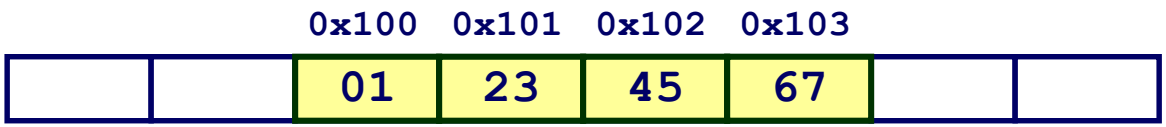
```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

`x < 0`	⇒	`((x*2) < 0)`	✗
`ux >= 0`			✓
`x & 7 == 7`	⇒	`(x<<30) < 0`	✓
`ux > -1`			✗
`x > y`	⇒	`-x < -y`	✗
`x * x >= 0`			✗
`x > 0 && y > 0`	⇒	`x + y > 0`	✗
`x >= 0`	⇒	`-x <= 0`	✓
`x <= 0`	⇒	`-x >= 0`	✗
`(x|-x)>>31 == -1`			✗
`ux >> 3 == ux/8`			✓
`x >> 3 == x/8`			✗
`x & (x-1) != 0`			✗

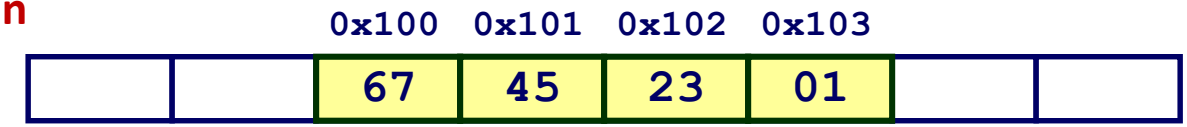
Byte Ordering Example

- **Example**
  - Variable x has 4-byte value of **0x01234567**
  - Address given by **&x** is **0x100**

Big Endian



Little Endian



This is key in gdb and ProxyLab

Representing Strings

```
char S[6] = "18213";
```

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character “0” has code 0x30
      - Digit *i* has code 0x30+*i*
  - String should be null-terminated
    - Final character = 0
- **Compatibility**
  - Byte ordering not an issue

x86-64

31
38
32
31
33
00

Floating Point in C

- **C Guarantees Two Levels**
  - **`float`** single precision
  - **`double`** double precision
- **Conversions/Casting**
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`double`**/**`float`** → **`int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int`** → **`double`**
    - Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`int`** → **`float`**
    - Will round according to rounding mode

Floating Point Puzzles

- **For each of the following C expressions, either:**
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;
float f = …;
double d = …;
```

Assume neither
`d` nor `f` is NaN

- `x == (int)(float) x` ✗
- `x == (int)(double) x` ✓
- `f == (float)(double) f` ✓
- `d == (double)(float) d` ✗
- `f == -(-f);` ✓
- `2/3 == 2/3.0` ✗
- `d < 0.0 ⇒ ((d*2) < 0.0)` ✓
- `d > f ⇒ -f > -d` ✓
- `d * d >= 0.0` ✓
- `(d+f)-d == f` ✗

Summarizing C Control Flow

- **C Control**
  - if-then-else
  - do-while
  - while, for
  - switch
- **What to watch out for**
  - **`if-then-else`**
    proper {…} for **`else if`**, logical condition in (…), **`=`** vs. **`==`**
  - **`do-while`**, **`while`**, **`for`**:
    init, test, update: which ones execute at least once,…
    break, continue
  - **`switch`**
    break, default
  - **`goto`**, **`labels`**, **`setjmp/longjmp`**

Functions

- **Functions**
  have return value, can be ignored
- **Procedures: no return value**
  `void x();`
- **Call by value/by reference**
  use of pointers, when to copy
- **Recursion: nothing special**
  Declaration and definition
- **Modifiers: `__inline`, `__naked`, …**
- **Variable length arguments:**
  `printf`, `scanf`,…
- **In your own functions:**
  `int sum(int n, …)`
  `va_arg()`

Scope of C in 18-613

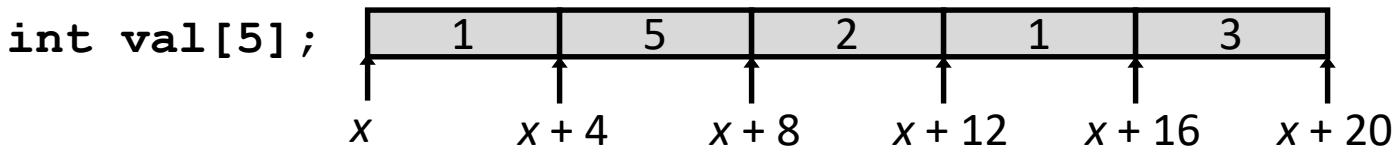
- Bootcamp: Linux and Windows
- Basic data types and functions
- **Structures, arrays, etc.**
- C Standard Library
- Linux/POSIX Libraries
- Language Extensions
- Final thoughts

Array Access

- Basic Principle

  T **`A[`***L***`];`**

  - Array of data type *T* and length *L*
  - Identifier **`A`** can be used as a pointer to array element 0: Type *T**



- Reference Type Value

Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	*x* + 4 *i*

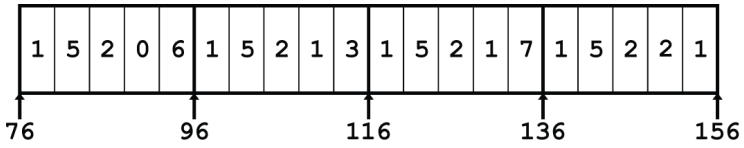
Array Element Accesses

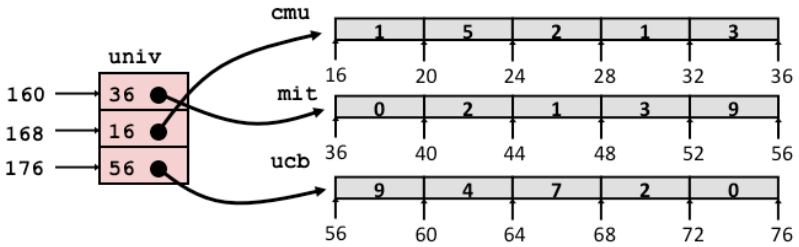
Nested array

```
int get_pgh_digit
   (size_t index, size_t digit)
{
  return pgh[index][digit];
}
```

Multi-level array

```
int get_univ_digit
   (size_t index, size_t digit)
{
  return univ[index][digit];
}
```





Accesses looks similar in C, but address computations very different:

`Mem[pgh+20*index+4*digit]` **`Mem[Mem[univ+8*index]+4*digit]`**

N X *N* Matrix Code

- **Fixed dimensions**
  - Know value of *N* at compile time

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element A[i][j] */
int fix_ele(fix_matrix A,
            size_t i, size_t j)
{
  return A[i][j];
}
```

- **Variable dimensions, explicit indexing**
  - Traditional way to implement dynamic arrays

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element A[i][j] */
int vec_ele(size_t n, int *A,
            size_t i, size_t j)
{
  return A[IDX(n,i,j)];
}
```

- **Variable dimensions, implicit indexing**
  - Now supported by gcc

```
/* Get element a[i][j] */
int var_ele(size_t n, int A[n][n],
            size_t i, size_t j) {
  return A[i][j];
}
```

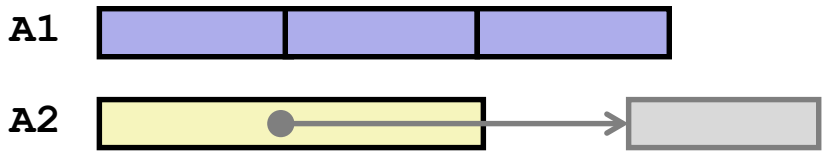
Understanding Pointers & Arrays #1

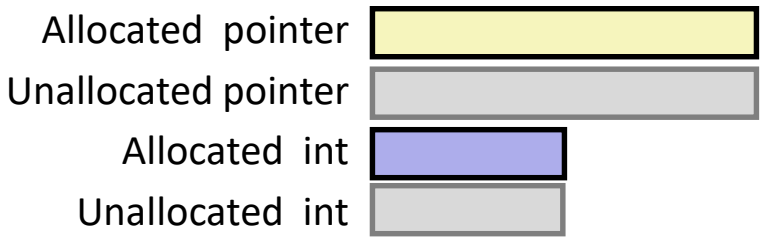
Decl	*An*			**An*		
	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`						
`int *A2`						

- Cmp: Compiles (Y/N)
- Bad: Possible bad pointer reference (Y/N)
- Size: Value returned by `sizeof`

Understanding Pointers & Arrays #1

Decl	*An*			**An*		
	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4
`int *A2`	Y	N	8	Y	Y	4





- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

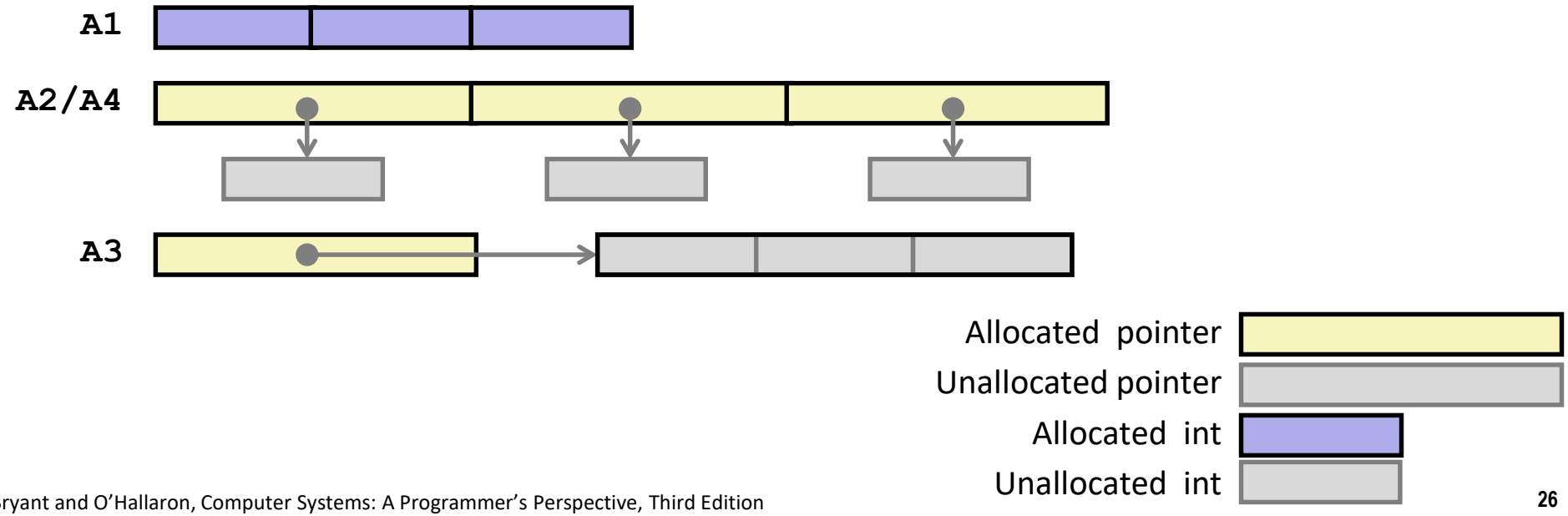
Understanding Pointers & Arrays #2

Decl	*An*			**An*			***An*		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`									
`int *A2[3]`									
`int (*A3)[3]`									
`int (*A4[3])`									

- Cmp: Compiles (Y/N)
- Bad: Possible bad pointer reference (Y/N)
- Size: Value returned by `sizeof`

Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
int A1[3]	Y	N	12	Y	N	4	N	–	–
int *A2[3]	Y	N	24	Y	N	8	Y	Y	4
int (*A3)[3]	Y	N	8	Y	Y	12	Y	Y	4
int (*A4[3])	Y	N	24	Y	N	8	Y	Y	4

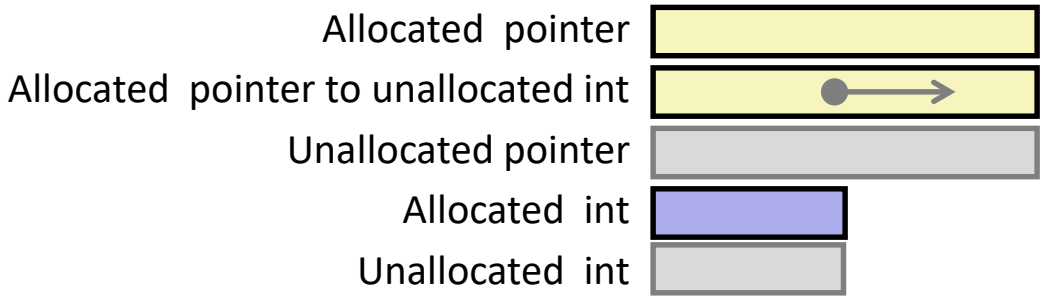


Understanding Pointers & Arrays #3

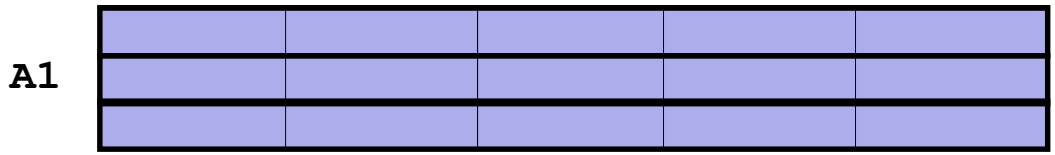
Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3][5]`									
`int *A2[3][5]`									
`int (*A3)[3][5]`									
`int *(A4[3][5])`									
`int (*A5[3])[5]`									

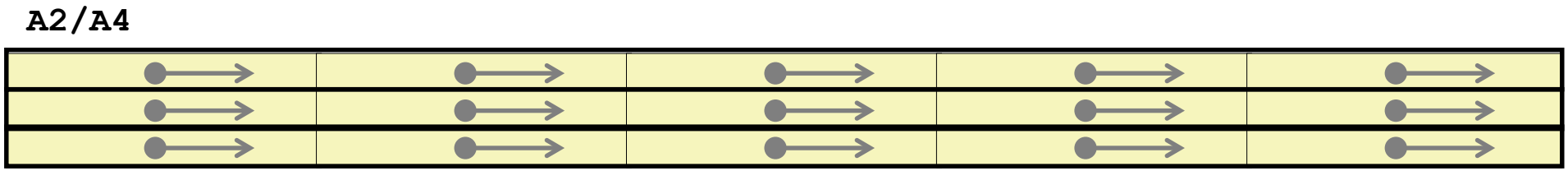
- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

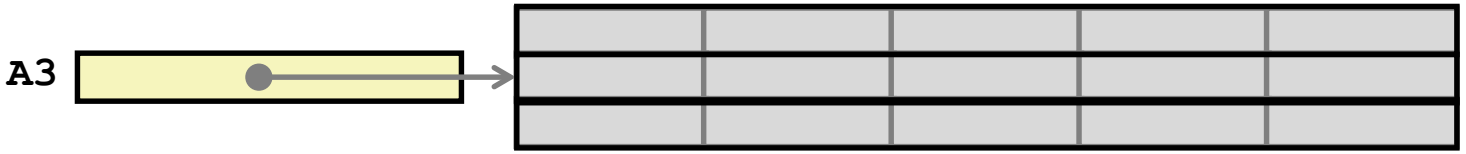
Decl	***An		
	Cmp	Bad	Size
`int A1[3][5]`			
`int *A2[3][5]`			
`int (*A3)[3][5]`			
`int *(A4[3][5])`			
`int (*A5[3])[5]`			

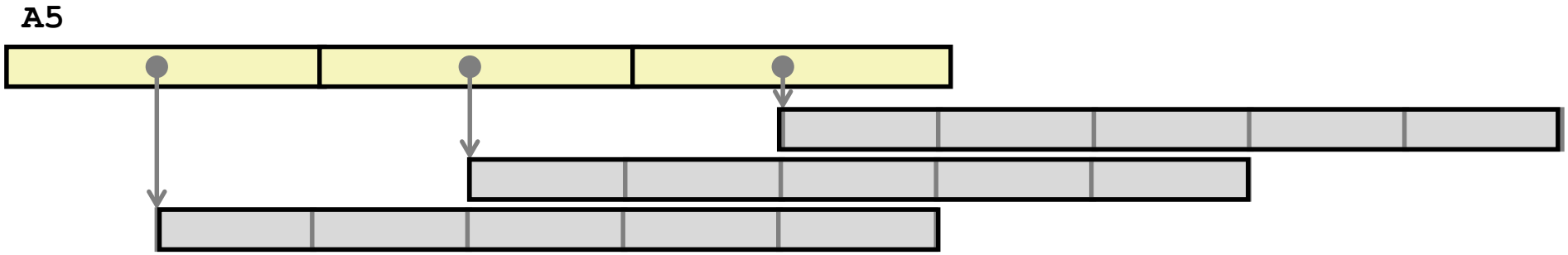


Declaration
`int A1[3][5]`
`int *A2[3][5]`
`int (*A3)[3][5]`
`int *(A4[3][5])`
`int (*A5[3])[5]`









Understanding Pointers & Arrays #3

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3][5]`	Y	N	60	Y	N	20	Y	N	4
`int *A2[3][5]`	Y	N	120	Y	N	40	Y	N	8
`int (*A3)[3][5]`	Y	N	8	Y	Y	60	Y	Y	20
`int *(A4[3][5])`	Y	N	120	Y	N	40	Y	N	8
`int (*A5[3])[5]`	Y	N	24	Y	N	8	Y	Y	20

- Cmp: Compiles (Y/N)
- Bad: Possible bad pointer reference (Y/N)
- Size: Value returned by `sizeof`

Decl	***An		
	Cmp	Bad	Size
`int A1[3][5]`	N	–	–
`int *A2[3][5]`	Y	Y	4
`int (*A3)[3][5]`	Y	Y	4
`int *(A4[3][5])`	Y	Y	4
`int (*A5[3])[5]`	Y	Y	4

Pointers and Function Pointers

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints

Source: K&R Sec 5.12

Typedef, Struct, Volatile, Memory Error…

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

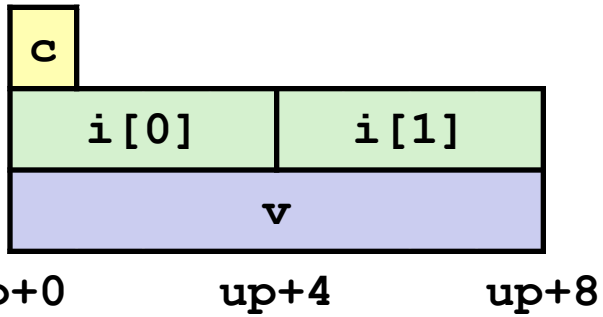
```
fun(0)   ->    3.14
fun(1)   ->    3.14
fun(2)   ->    3.1399998664856
fun(3)   ->    2.00000061035156
fun(4)   ->    3.14
fun(6)   ->    Segmentation fault
```

- Result is system specific

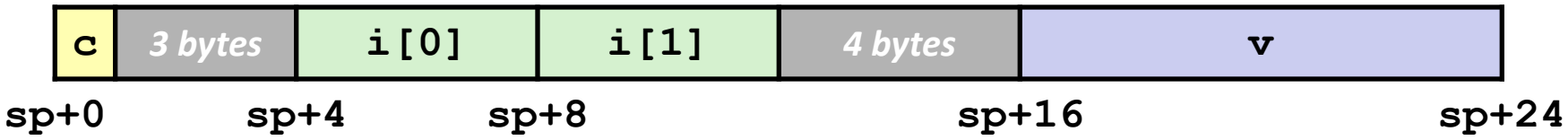
Union

- Allocate according to largest element
- Can only use one field at a time

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

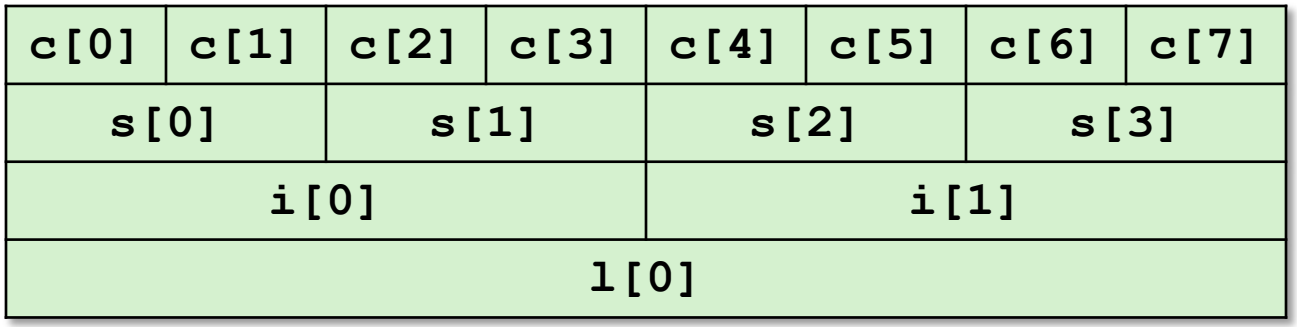



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Union: Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```



Variable Declarations: Multiple Files

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (**p1**)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to **x** will refer to the same uninitialized int. Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to **x** in **p2** might overwrite **y**! Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to **x** in **p2** will overwrite **y**! Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to **x** will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Global Variables

- **Avoid if you can**

- **Otherwise**
  - Use **`static`** if you can
  - Initialize if you define a global variable
  - Use **`extern`** if you reference an external global variable

Preprocessor: Pragmas and Macros

- Textual expansion to avoid copy-and-paste

```
#define MIN(a,b) (((a)<(b))?(a):(b))
```

- Control compilation, can set from command line `gcc -DDEBUG`

```
#ifdef DEBUG
…
#else
…
#endif
```

- Pragmas control compilation and compiler options

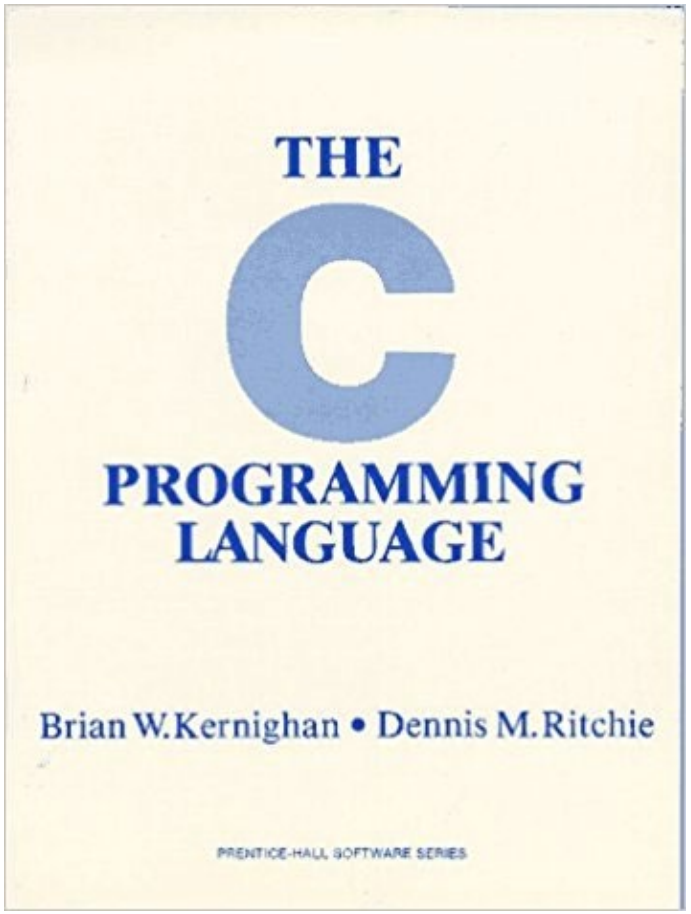
```
foo() {
#pragma GCC diagnostic ignored "-Wunused-variable"
  int i, j;
  i=3;
}
```

Scope of C in 18-613

- Bootcamp: Linux and Windows
- Basic data types and functions
- Structures, arrays, etc.
- **C Standard Library**
- Linux/POSIX Libraries
- Language Extensions
- Final thoughts

C Standard Libraries: K&R Appendix B

- Input & output: `<stdio.h>`
- String functions: `<string.h>`
- Mathematical functions: `<math.h>`
- Utility functions: `<stdlib.h>`
- Date and time: `<time.h>`
- and a few more …



Standard I/O Functions

- **The C standard library (`libc.so`) contains a collection of higher-level *standard I/O* functions**
  - Documented in Appendix B of K&R

- **Examples of standard I/O functions:**
  - Opening and closing files (**`fopen`** and **`fclose`**)
  - Reading and writing bytes (**`fread`** and **`fwrite`**)
  - Reading and writing text lines (**`fgets`** and **`fputs`**)
  - Formatted reading and writing (**`fscanf`** and **`fprintf`**)

Standard I/O Streams

- **Standard I/O models open files as *streams***
  - Abstraction for a file descriptor and a buffer in memory

- **C programs begin life with three open streams (defined in `stdio.h`)**
  - **`stdin`** (standard input)
  - **`stdout`** (standard output)
  - **`stderr`** (standard error)

```
#include <stdio.h>
extern FILE *stdin;  /* standard input  (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error  (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

malloc Example

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

void foo(int n) {
    int i, *p;

    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;


    /* Return allocated block to the heap */
    free(p);
}
```

Nonlocal Jumps: `setjmp/longjmp`

- **Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location**
  - Controlled to way to break the procedure call / return discipline
  - Useful for error recovery and signal handling
- **`int setjmp(jmp_buf j)`**
  - Must be called before longjmp
  - Identifies a return site for a subsequent longjmp
  - Called **once**, returns **one or more** times
- **`void longjmp(jmp_buf j, int i)`**
  - Meaning:
    - return from the **`setjmp`** remembered by jump buffer **`j`** again ...
    - ... this time returning **`i`** instead of 0
  - Called after **`setjmp`**
  - Called **once**, but **never** returns

Scope of C in 18-613

- Bootcamp: Linux and Windows
- Basic data types and functions
- Structures, arrays, etc.
- C Standard Library
- **Linux/POSIX Libraries**
- Language Extensions
- Final thoughts

Functions in Shared Objects

```
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

/* malloc wrapper function */
void *malloc(size_t size)
{
    void *(*mallocp)(size_t size);
    char *error;

    mallocp = dlsym(RTLD_NEXT, "malloc"); /* Get addr of libc malloc */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    char *ptr = mallocp(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

Process Control

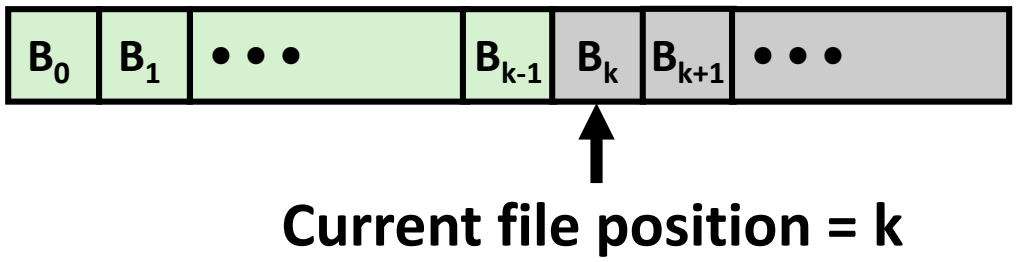
- **Spawning processes**
  - Call **`fork`**
  - One call, two returns
- **Process completion**
  - Call **`exit`**
  - One call, no return
- **Reaping and waiting for processes**
  - Call **`wait`** or **`waitpid`**
- **Loading and running programs**
  - Call **`execve`** (or variant)
  - One call, (normally) no return

Signals

```
void sigint_handler(int sig) /* SIGINT handler */
{
    printf("So you think you can stop the bomb with ctrl-c, do you?\n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK. :-)\n");
    exit(0);
}

int main()
{
    /* Install the SIGINT handler */
    if (signal(SIGINT, sigint_handler) == SIG_ERR)
        unix_error("signal error");

    /* Wait for the receipt of a signal */
    pause();

    return 0;
}
```

Unix I/O

- **Elegant mapping of files to devices allows kernel to export simple interface called *Unix I/O:***
  - Opening and closing files
    - **`open()`** and **`close()`**
  - Reading and writing a file
    - **`read()`** and **`write()`**
  - Changing the ***current file position*** (seek)
    - indicates next offset into file to read or write
    - **`lseek()`**



Networking: Sockets Interface

- **Set of system-level functions used in conjunction with Unix I/O to build network applications.**
- **Created in the early 80's as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols.**
- **Available on all modern systems**
  - Unix variants, Windows, OS X, IOS, Android, ARM

Clients and servers use the `socket` function to create a *socket descriptor*:

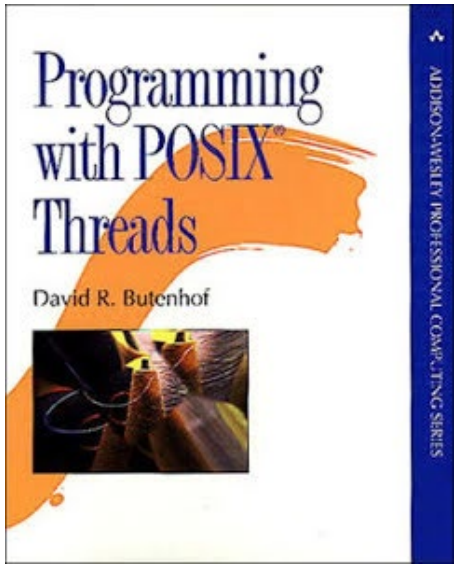
```
int socket(int domain, int type, int protocol)
```

Example:

```
int clientfd = Socket(AF_INET, SOCK_STREAM, 0);
```

Posix Threads (Pthreads) Interface

- ***Pthreads:*** **Standard interface for ~60 functions that manipulate threads from C programs**
  - Creating and reaping threads
    - **`pthread_create()`**
    - **`pthread_join()`**
  - Determining your thread ID
    - **`pthread_self()`**
  - Terminating threads
    - **`pthread_cancel()`**
    - **`pthread_exit()`**
    - **`exit()`** [terminates all threads] , **`RET`** [terminates current thread]
  - Synchronizing access to shared variables
    - **`pthread_mutex_init`**
    - **`pthread_mutex_[un]lock`**



Scope of C in 18-613

- Bootcamp: Linux and Windows
- Basic data types and functions
- Structures, arrays, etc.
- C Standard Library
- Linux/POSIX Libraries
- **Language Extensions**
- Final thoughts

C Standards

- Kerninghan & Ritchie
- ANSI C/C 89
- GNU C 89
- C99
- C11

  Shark machines: gcc version 4.8.5
- C18: not yet supported

C99, C11, Advanced Constructs

Variable length arrays
Designated initializers
Type-generic math library
New datatypes: long long, _Complex, _Bool
restrict pointers, _Generic
Intermingled declarations of variables
Inline functions
One-line comments that begin with //

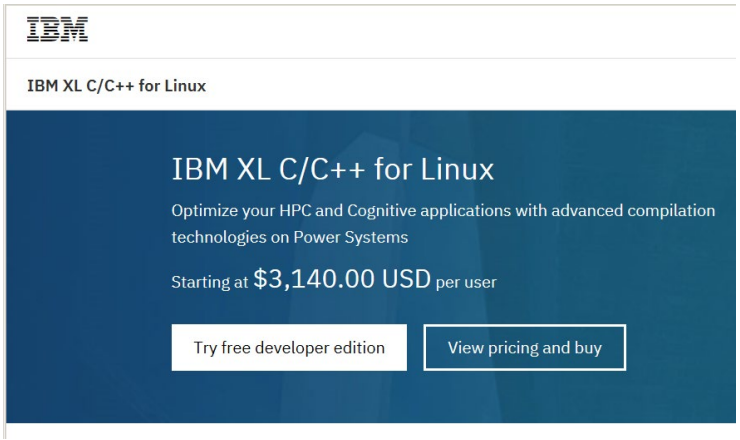
Advanced C/C++ Compilers

Intel C/C++

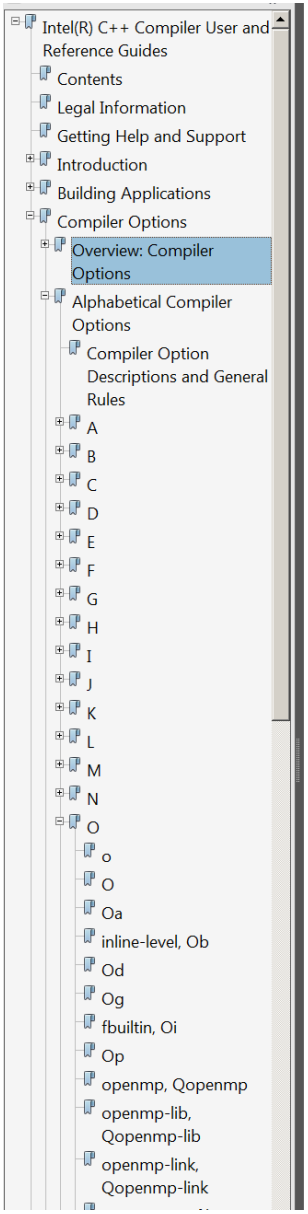
LLVM and GCC

PGI Compiler

IBM XL C



Compiler Options



Windows* OS Options	Description	Default
/QaxSSE4.2 (i32, i64em)	Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors. Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for the Intel® Core™ processor family.	OFF
/Qcov-dir dir (i32, i64em)	Specifies a directory for profiling information output files that can be used with the codecov or tselect tool.	OFF
/Qcov-file file (i32, i64em)	Specifies an alternate file name for the profiling summary files that can be used with the codecov or tselect tool.	OFF
/Qdiag-enable:sc-parallel (i32, i64em)	Enables analysis of parallelization in source code (parallel lint diagnostics).	OFF
/Qdiag-error-limit:n	Specifies the maximum number of errors allowed before compilation stops.	n=30
/Qdiag-once:id[,id,...]	Tells the compiler to issue one or more diagnostic messages only once.	OFF
/Qdiag-error-limit:n	Specifies the maximum number of errors allowed before compilation stops.	n=30
/Qdiag-once:id[,id,...]	Tells the compiler to issue one or more diagnostic messages only once.	OFF
/Qfast-transcendentals	Enables the compiler to replace calls to transcendental functions with faster but less precise implementations.	OFF
/Qfma (i64 only)	Enables the combining of floating-point multiplies and add/subtract operations.	ON

Beyond 18-613: Bit Fields

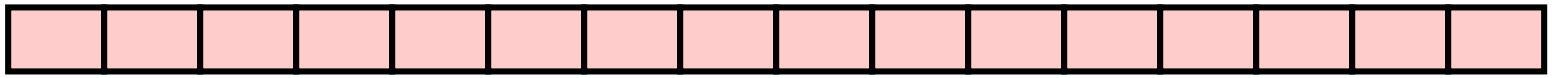
- Convenient way to access bits in status registers etc.
- Useful with union to avoid type casting (up to `long long`)
- Must fit within integral type

```
#include <stdio.h>
struct ieee754d {
   unsigned int s:    1;
   unsigned int exp:  11;
   unsigned int frac: 52;
};

int main() {
   struct ieee754d d = {0U, 0U, 33729U};
   d.exp = 113U;
   printf("%e", (double *)(&d));
   return 0;
}
```

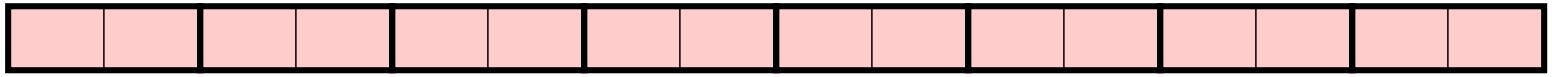
18-613: Programming with SSE3,…,AVX512

XMM registers (later versions: YMM, ZMM)

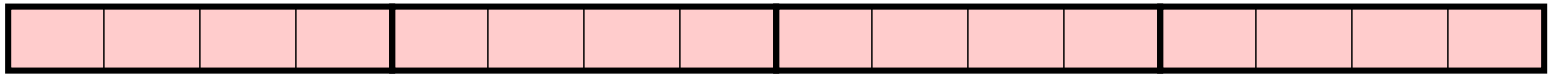
- 16 total, each 16 bytes
- 16 single-byte integers



- 8 16-bit integers



- 4 32-bit integers

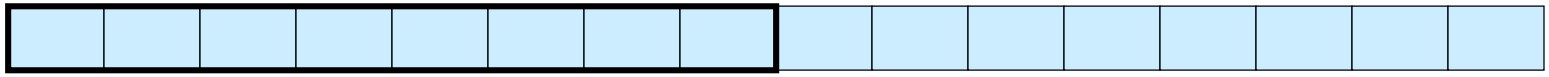


- 4 single-precision floats

- 2 double-precision floats

- 1 single-precision float

- 1 double-precision float



Vector Instructions: Language Extension

- **Language extension**
  - Aligned C library functions: `_mm_malloc()`, `_mm_free()`
  - Attributes: `_assume_aligned()`, `__declspec(__align())`
  - Pragmas

    ```
    #pragma vector aligned | unaligned | always
    #pragma ivdep, #pragma novector
    ```

- **Data types**
  - `__m128 f; // ={float f3, f2, f1, f0}`
  - `__m128d d; // ={double d1, d0}`
- **Intrinsics**
  - **Native instructions:** `_mm_add_ps()`, `_mm_mul_ps()`,…
  - **Multi-instruction:** `_mm_setr_ps()`, `_mm_set1_ps`, …
- **Macros**
  - **Transpose:** `_MM_TRANSPOSE4_PS()`,…
  - **Helper:** `_MM_SHUFFLE()`

intel Intrinsics Guide

_mm_search ?

Technologies
- MMX
- SSE
- SSE2
- SSE3
- SSSE3
- SSE4.1
- SSE4.2
- AVX
- AVX2
- FMA
- AVX-512
- KNC
- SVML
- Other

Categories
- Application-Targeted
- Arithmetic
- Bit Manipulation
- Cast
- Compare
- Convert
- Cryptography
- Elementary Math Functions
- General Support
- Load
- Logical
- Mask
- Miscellaneous
- Move
- OS-Targeted
- Probability/Statistics
- Random
- Set
- Shift
- Special Math Functions
- Store
- String Compare
- Swizzle
- Trigonometry

Legal Statement

__m512i _mm512_4dpwssd_epi32 (_m512i src, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b)	vp4dpwssd
__m512i _mm512_mask_4dpwssd_epi32 (_m512i src, _mmask16 k, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b)	vp4dpwssd
__m512i _mm512_maskz_4dpwssd_epi32 (_mmask16 k, _m512i src, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b)	vp4dpwssd
__m512i _mm512_4dpwssds_epi32 (_m512i src, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b)	vp4dpwssds
__m512i _mm512_mask_4dpwssds_epi32 (_m512i src, _mmask16 k, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b)	vp4dpwssds
__m512i _mm512_maskz_4dpwssds_epi32 (_m512i src, _mmask16 k, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b)	vp4dpwssds
__m512 _mm512_4fmadd_ps (_m512 a, _m512i b0, _m512i b1, _m512i b2, _m512i b3, _m128i * c)	v4fmaddps
__m512 _mm512_mask_4fmadd_ps (_m512 a, _mmask16 k, _m512i b0, _m512i b1, _m512i b2, _m512i b3, _m128i * c)	v4fmaddps
__m512 _mm512_maskz_4fmadd_ps (_m512 a, _mmask16 k, _m512i b0, _m512i b1, _m512i b2, _m512i b3, _m128i * c)	v4fmaddps
__m128 _mm_4fmadd_ss (__m128 a, __m128 b0, __m128 b1, __m128 b2, __m128 b3, __m128 * c)	v4fmaddss
__m128 _mm_mask_4fmadd_ss (__m128 a, __mmask8 k, __m128 b0, __m128 b1, __m128 b2, __m128 b3, __m128 * c)	v4fmaddss
__m128 _mm_maskz_4fmadd_ss (__m128 a, __mmask8 k, __m128 b0, __m128 b1, __m128 b2, __m128 b3, __m128 * c)	v4fmaddss
__m512 _mm512_4fnmadd_ps (_m512 a, _m512i b0, _m512i b1, _m512i b2, _m512i b3, _m128i * c)	v4fnmaddps
__m512 _mm512_mask_4fnmadd_ps (_m512 a, _mmask16 k, _m512i b0, _m512i b1, _m512i b2, _m512i b3, _m128i * c)	v4fnmaddps
__m512 _mm512_maskz_4fnmadd_ps (_m512 a, _mmask16 k, _m512i b0, _m512i b1, _m512i b2, _m512i b3, _m128i * c)	v4fnmaddps
__m128 _mm_4fnmadd_ss (__m128 a, __m128 b0, __m128 b1, __m128 b2, __m128 b3, __m128 * c)	v4fnmaddss
__m128 _mm_mask_4fnmadd_ss (__m128 a, __mmask8 k, __m128 b0, __m128 b1, __m128 b2, __m128 b3, __m128 * c)	v4fnmaddss
__m128 _mm_maskz_4fnmadd_ss (__m128 a, __mmask8 k, __m128 b0, __m128 b1, __m128 b2, __m128 b3, __m128 * c)	v4fnmaddss
__m128i _mm_abs_epi16 (__m128i a)	pabsw
__m128i _mm_mask_abs_epi16 (__m128i src, __mmask8 k, __m128i a)	vpabsw
__m128i _mm_maskz_abs_epi16 (__mmask8 k, __m128i a)	vpabsw
__m256i _mm256_abs_epi16 (__m256i a)	vpabsw
__m256i _mm256_mask_abs_epi16 (__m256i src, __mmask16 k, __m256i a)	vpabsw
__m256i _mm256_maskz_abs_epi16 (__mmask16 k, __m256i a)	vpabsw
__m512i _mm512_abs_epi16 (__m512i a)	vpabsw
__m512i _mm512_mask_abs_epi16 (__m512i src, __mmask32 k, __m512i a)	vpabsw
__m512i _mm512_maskz_abs_epi16 (__mmask32 k, __m512i a)	vpabsw

SSE4.1 Code Example

```
int dwmonitor(float  *X, double  *D) {
    __m128d u1, u2, u3, u4, u5, u6, u7, u8 , x1, x10, x13, x14, x17, x18, x19, x2, x3, x4, x6, x7, x8, x9;
    int w1;
    unsigned _xm = _mm_getcsr();
    _mm_setcsr(_xm & 0xffff0000 | 0x0000dfc0);
    u5 = _mm_set1_pd(0.0);
    u2 = _mm_cvtps_pd(_mm_addsub_ps(_mm_set1_ps(FLT_MIN), _mm_set1_ps(X[0])));
    u1 = _mm_set_pd(1.0, (-1.0));
    for(int i5 = 0; i5 <= 2; i5++) {
        x6 = _mm_addsub_pd(_mm_set1_pd((DBL_MIN + DBL_MIN)), _mm_loaddup_pd(&(D[i5])));
        x1 = _mm_addsub_pd(_mm_set1_pd(0.0), u1);
        x2 = _mm_mul_pd(x1, x6);
        x3 = _mm_mul_pd(_mm_shuffle_pd(x1, x1, _MM_SHUFFLE2(0, 1)), x6);
        x4 = _mm_sub_pd(_mm_set1_pd(0.0), _mm_min_pd(x3, x2));
        u3 = _mm_add_pd(_mm_max_pd(_mm_shuffle_pd(x4, x4, _MM_SHUFFLE2(0, 1)), _mm_max_pd(x3, x2)), _mm_set1_pd(DBL_MIN));
        u5 = _mm_add_pd(u5, u3);
        x7 = _mm_addsub_pd(_mm_set1_pd(0.0), u1);
        x8 = _mm_mul_pd(x7, u2);
        x9 = _mm_mul_pd(_mm_shuffle_pd(x7, x7, _MM_SHUFFLE2(0, 1)), u2);
        x10 = _mm_sub_pd(_mm_set1_pd(0.0), _mm_min_pd(x9, x8));
        u1 = _mm_add_pd(_mm_max_pd(_mm_shuffle_pd(x10, x10, _MM_SHUFFLE2(0, 1)), _mm_max_pd(x9, x8)), _mm_set1_pd(DBL_MIN));
    }
    u6 = _mm_set1_pd(0.0);
    for(int i3 = 0; i3 <= 1; i3++) {
        u8 = _mm_cvtps_pd(_mm_addsub_ps(_mm_set1_ps(FLT_MIN), _mm_set1_ps(X[(i3 + 1)])));
        u7 = _mm_cvtps_pd(_mm_addsub_ps(_mm_set1_ps(FLT_MIN), _mm_set1_ps(X[(3 + i3)])));
        x14 = _mm_add_pd(u8, _mm_shuffle_pd(u7, u7, _MM_SHUFFLE2(0, 1)));
        x13 = _mm_shuffle_pd(x14, x14, _MM_SHUFFLE2(0, 1));
        u4 = _mm_shuffle_pd(_mm_min_pd(x14, x13), _mm_max_pd(x14, x13), _MM_SHUFFLE2(1, 0));
        u6 = _mm_shuffle_pd(_mm_min_pd(u6, u4), _mm_max_pd(u6, u4), _MM_SHUFFLE2(1, 0));
    }
    x17 = _mm_addsub_pd(_mm_set1_pd(0.0), u6);
    x18 = _mm_addsub_pd(_mm_set1_pd(0.0), u5);
    x19 = _mm_cmpge_pd(x17, _mm_shuffle_pd(x18, x18, _MM_SHUFFLE2(0, 1)));
    w1 = (_mm_testc_si128(_mm_castpd_si128(x19), _mm_set_epi32(0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff)) -
          (_mm_testnzc_si128(_mm_castpd_si128(x19), _mm_set_epi32(0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff))));
    __asm nop;
    if (_mm_getcsr() & 0x0d) {
        _mm_setcsr(_xm);
        return -1;
    }
    _mm_setcsr(_xm);
    return w1;
}
```

Beyond 18-613: Inline Assembly

- Write assembly instructions directly in C
- Compiler and machine specific

```
int src = 1;
int dst;

asm ("mov %1, %0\n\t"
     "add $1, %0"
     : "=r" (dst)
     : "r" (src));

printf("%d\n", dst)
```

- Can trigger breakpoints
  (but there are better ways to do this)

```
asm volatile ("int 3");
```

18-613: OpenMP for Multicore

```
//compute the sum of two arrays in parallel
#include <stdio.h>
#include <omp.h>
#define N 1000000
int main(void) {
  float a[N], b[N], c[N];
  int i;

  /* Initialize arrays a and b */
  for (i = 0; i < N; i++) {
    a[i] = i * 2.0;
    b[i] = i * 3.0;
  }

 /* Compute values of array c = a+b in parallel. */
  #pragma omp parallel shared(a, b, c) private(i)
  {
    #pragma omp for
    for (i = 0; i < N; i++)
      c[i] = a[i] + b[i];
  }
}
```

18-613: CUDA for Nvidia GPUs

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Scope of C in 18-613

- Bootcamp: Linux and Windows
- Basic data types and functions
- Structures, arrays, etc.
- C Standard Library
- Linux/POSIX Libraries
- Language Extensions
- **Final thoughts**

18-645: How to Write Fast Code

How To Write Fast Numerical Code: A Small Introduction

Srinivas Chellappa, Franz Franchetti, and Markus Püschel

Electrical and Computer Engineering
Carnegie Mellon University
{schellap, franzf, pueschel}@ece.cmu.edu

Abstract. The complexity of modern computing platforms has made it extremely difficult to write numerical code that achieves the best possible performance. Straightforward implementations based on algorithms that minimize the operations count often fall short in performance by at least one order of magnitude. This tutorial introduces the reader to a set of general techniques to improve the performance of numerical code, focusing on optimizations for the computer's memory hierarchy. Further, program generators are discussed as a way to reduce the implementation and optimization effort. Two running examples are used to demonstrate these techniques: matrix-matrix multiplication and the discrete Fourier transform.

1 Introduction

The growth in the performance of computing platforms in the past few decades has followed a reliable pattern usually referred to as Moore's Law. Moore observed in 1965 [1] that the number of transistors per chip roughly doubles every 18 months and predicted—correctly—that this trend would continue. In parallel, due to the shrinking size of transistors, CPU frequencies could be increased at roughly the same exponential rate. This trend has been the big supporter for many performance demanding applications in scientific computing (such as climate modeling and other physics simulations), consumer computing (such as audio, image, and video processing), and embedded computing (such as control, communication, and signal processing). In fact, these domains have a practically unlimited need for performance (for example, the ever growing need for higher resolution videos), and it seems that the evolution of computers is well on track to support these needs.

However, everything comes at a price, and in this case it is the increasing difficulty of writing the fastest possible software. In this tutorial, we focus on *numerical* software. By that we mean code that mainly consists of floating point computations.

The problem. To understand the problem we investigate Fig. 1, which considers various Intel architectures from the first Pentium to the (at the time of this writing) latest Core2 Extreme. The x-axis shows the year of release. The y-axis, in log-scale, shows both the CPU frequency (in MHz) and the single/double precision theoretical peak performance (in Mflop/s = Mega FLoating point OPerations per Second) of the respective machines.

Tentative Course Calendar:

Date	Day	Class Activity
August		
27	Mon.	Semester Begin – Overview of course
29	Wed.	Architecture and its impact on performance
September		
3	Mon.	Labor Day; No Classes
5	Wed.	Designing fast kernels
10	Mon.	Benchmarking
12	Wed.	Intro to SIMD Programming
17	Mon.	Compiler Optimizations
19	Wed.	Class Cancelled – Replaced with Project Kickoff Meetings
24	Mon.	The memory hierarchy
26	Wed.	Optimization for the memory hierarchy – Part 1
October		
1	Mon.	Optimization for the memory hierarchy – Part 2
3	Wed.	Intro to Explicit Parallelism
8	Mon.	Collective Communications – Part 1
10	Wed.	Class Cancelled – Replaced with Project Midterm Review Meetings
15	Mon.	Collective Communications – Part 2
17	Wed.	Designing Efficient Communication Patterns – Part 1
19	Fri.	Mid-Semester Break; No Classes
22	Mon.	Designing Efficient Communication Patterns – Part 2
24	Wed.	Shared Memory Parallelism
29	Mon.	From Shared Memory to Heterogeneous Architectures
31	Wed.	Introduction to Algorithm Design
November		
5	Mon.	Putting everything together
7	Wed.	Class Cancelled – Replaced with Project Final Review Meetings
12	Mon.	Midterm Review
14	Wed.	Midterm Exam
19	Mon.	Project Meeting Time
21-23	W-F	Thanksgiving Break; No Classes
26	Mon.	Project Presentations
28	Wed.	Project Presentations
December		
3	Mon.	Project Presentations
5	Wed.	Last Day of Class – Wrap Up

https://courses.ece.cmu.edu/18645
http://www.ece.cmu.edu/~franzf/papers/gttse07.pdf
https://www.inf.ethz.ch/personal/markusp/teaching/18-645-CMU-spring08/course.html

15-418/15-618: Parallel Computer Architecture and Programming, Spring 2018: Schedule

Notes https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15418-s18/www/schedule.html

- The exact topics of the lectures are subject to change.
- We do not anticipate changing any of the other dates (exams, assignments, etc.)

Date	Topic	Assignment
Jan 15	MLK Day. *No class*	
Jan 17	Why parallelism (pdf , video)	Assignment 1 out (pdf)
Jan 19	Modern multicore processors (pdf , video)	
Jan 22	Parallel programming models (pdf , video)	
Jan 24	Parallel programming basics (pdf , video)	Assignment 1 due for waitlisted students
Jan 26	Work distribution and scheduling (pdf , video)	
Jan 29	**Recitation**: ILP, SIMD instructions (pdf , pptx , video)	Assignment 1 due for registered students, assignment 2 out (pdf)
Jan 31	Graphic processing units and CUDA (pdf , video)	
Feb 2	**Recitation**: CUDA programming 1 (pdf , pptx)	
Feb 5	Locality, communication, and contention (pdf , video)	
Feb 7	Application case studies (pdf , video)	
Feb 9	**Recitation**: CUDA programming 2 (pdf , pptx)	
Feb 12	Workload-driven performance evaluation (pdf , video)	
Feb 14	Snooping-based cache coherence (pdf , video)	Assignment 2 due, assignment 3 out (pdf)
Feb 16	**Recitation**: Understanding Assignment 3 (pdf , pptx , video)	
Feb 19	Directory-based cache coherence (pdf , video)	
Feb 21	Snooping implementation (pdf , video)	

```
#include<fcntl.h>
#include<unistd.h>
#include<termios.h>
#include <sys/time.h>
#include  <sys/mman.h>
# define L } if(!i -- ){
struct timeval F,G; struct termios H,U={ T} ; enum{ N=64,a=N<<7,b=a-1,c=a*32,d
=c-1,    e=c/    2,f=    a*2,    g=a/2,h =g/2,j =h/ 2,Q=V*j*5} ; char*s=P,K,M;
int*      p,      l[      a]      ,m,n,J,o=A,    O=j,E,R,i,k,t,r,q,u,v,w,x,y,z
,B,C,    *D,Z    ;int    main    (){ for(D=mmap(D,4*Q,3,W,open(I,2 ),K); *s; o
++[ l]=k|=* s++%N){ k=* s++%N<<12; k|=*s++% N*N; } tcgetattr(q,& H); tcsetattr
(y,2,    &U);    for(    fcntl(B,4,4); ; o&=b){ if(k&    c){ q=- --k%N; if(!q)
k-=c     ;i      =k      /N&7; { L L if(J&1)    m+=      t; J|=m%N*c; J/=2; m
/= 2;    if(!    q&&r    ^n){ m^=d; J^= d; n=0; } L L    J+=J; J|=m>=0; if(q){
m+=m; m|=J/c; m+=m<0?t:-t; } else{ m+=(m<0)*t; if(r)m^=d; if(n^r)J^= d; n=0; }
L if(    (m^2    *m)/    e%2)    k&=d; else{ J+=J; m+=m; m|=J/c; } L m|=n*c; J
|= m     %N      *c      ;m      /=2; J /=2;    L m+=m; J+=J; J|=n; m|=J/c; L
m+=m;    m|=n    ;  }    J&=d    ; } else{ i=k/f; t=i?k&b:16; p=l+t; if(k&a)p=
l+((*p+=13>i&&7<t&&16>t)&b); { L i=1; L*p=m; L*p++=o|n*e; o=p-l; L*p=0; L m=*p
; L m ^=*p ; L t=m; m    +=*p; m+=d<m    ; n|=((m^t)&    (m^*p))/e; L    m+=*p
; n=m/c; L k=*p; if(      !Z||k/f-8)      /*$    %*/      continue ;      k=-k
; L++*p; o +=!(*p&=d)    ; L m&=*p; L     if(m!=*p)++    o; L o=p -l;    L if(
k&a)n=m/e; if(k&g)J= 0; r=k&h&&m&e; if(k&j)J|=m; else if(r)m^=d; if(k&512)m=0;
i=k/N    &7; { L if(k    &4)J    ^=d; if(k&2)m|=J; if    (k&1    )m|=    q;  }
else     { if(k%N)k      +=      c; { L t =o    ++[      l]      ;       if(r
)J^=d;   L L t=o++[l];   if(r    )J^=d; m-=t; if(m>=0    ){ k    -=c;     n=1;
++o; } } i=2; } L if(Z)k=-1; else{ if(k&8)m=0; t=r=0; i=k/N%N; if(i==27){ if(k
&2)u=v=w=Z=0;    } if    (i==57){ i=k/16&3; { L w =1;    if(k&1)x =0;    if(k&
2) { t= z/N;      t=      t/80*/*/*/100+t% 80;   r=0      ; while(t)      {  r
+=t%10*w; t/=    10;w    <<=4; } m|=r; } if(k&4){ r=m    ; t=0; while    (r) {
t+=r%16*w; w*=10; r>>=4; } r=t/100; t%=100; if(V<=r||79<t)x|=c/8; else z=(r*80
+t)*N    ;  }    w=0;    L if(k& 1&&x    &(e|g) )++o;    if(k& 2)m|=x    |y;if
(k&4     )C      =-      m&65535; L      if(    k&1      )x=y=0; if      (k&2
)B=m;    if (    k&4)    { y^=m&(h|j|    j/2); if(y&j    ){ y^=j;x|=g    ; do{
B&=b; if(y&j/2)z[D]=B[l]; else B[l]=z[D]; ++z; z%=Q; ++B; } while(-- C); } } }
x%=e;    if(x    /a)x    |=e;    if(x&(e|g)&&y&h)u|=c    ; else u%=c;    L if(
k&1)     t=      h;      if      (k&2 )r= e;    if(      k&4){ r=j;      u&=~
h ; }    if(k    &16)    Z=f*    2; L L L t=f; if(k&2    )m|=M|Y; if(    k&4)m
|=u|v; L t=a; if(k&4){ K=m&~Y; write(1,&K,1); u|=t; t=0; } } i=2; if(t){ if(k&
1&&u&t)++ o; if(k&2)u    &=~ t; } if(    r){ if(k&32)    w=r;    else v&=~r; }
} L if(k&a)m=k; else     { t=0; if(      k &    N)t      |=      m/e%2; if(k&
128)t|=!m; if(k&256)t    |= n; if( k&    512 )t=!t; o    +=t;    if(k& h)n =0;
if(k&g)m=0; if(k&1)m^=d; if(k&2)n^=1; if(k&4)m|=S; if(k&8){ m|=n*c; m+=m; if(k
& j)m    +=m;    m|=m    /a/N    ; n=m/c; } if(k&16){ m|=n*c;    m|=m    *2%N*
c; m     /=      2;      if      (k&j) m/=2;    n=m/c; } if      (k      &32)
{ if(    Z)k=    -1 ;    else    break; } } } } n&=1; if(k<c)    {  m    &= d;
o &=b; if(!R--){ if (~u&f &&read(0, &M,1)>0){ if(X&& M== X)break; R=0; u|=f; }
gettimeofday(&G,0); G.tv_usec/=16667; if(G.tv_sec>F.tv_sec||F.tv_usec<G.
tv_usec){ F=G; if(v&j){ p=l+7; ++*p; *p&=d; if(!*p)u|=h; } R=0; } if
(!R){ E=O/4; O=4; } O+=R=E; } if(!++k||(v&e&&u)){ *l=o|n*e|Z;v%=
e; o=1+!k; Z=0; } v|=w; w=0; k=o++[l]; } }tcsetattr(w,1,&H); }
```

https://www.ioccc.org/2018/mills/prog.c

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
            } while (--n > 0);
    }
}
```

https://en.wikipedia.org/wiki/Duff%27s_device