Computer Systems I

Class 8

Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

Basic Data Types

- **Integral**
  - Stored & operated on in general (integer) registers
  - Signed vs. unsigned depends on instructions used

Intel	GAS	Bytes	C
byte	`b`	1	`[unsigned] char`
word	`w`	2	`[unsigned] short`
double word	`l`	4	`[unsigned] int`
quad word	`q`	8	`[unsigned] long int` (x86-64)

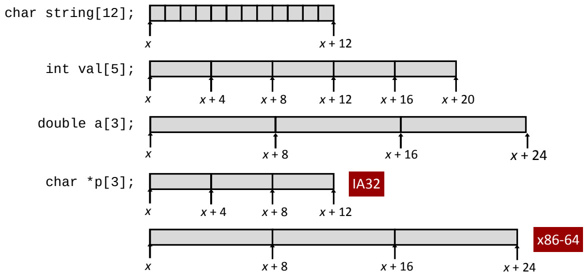
- **Floating Point**
  - Stored & operated on in floating point registers

Intel	GAS	Bytes	C
Single	`s`	4	`float`
Double	`l`	8	`double`
Extended	`t`	10/12/16	`long double`

Array Allocation

- **Basic Principle**

  T **A[*L*];**
  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **sizeof**(*T*) bytes

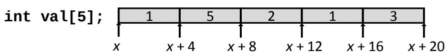


Systems I

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**



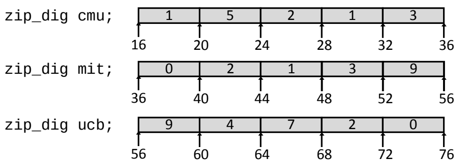
Reference	Type	Value
`val[4]`		
`val`		
`val+1`		
`&val[2]`		
`val[5]`		
`*(val+1)`		
`val + i`		

Systems I

Array Example

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- **Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"**
- **Example arrays were allocated in successive 20 byte blocks**
  - Not guaranteed to happen in general

Array Accessing Example

`zip_dig cmu;`

1	5	2	1	3

16 20 24 28 32 36

```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```

IA32

```
  # %edx = z
  # %eax = dig
  movl (%edx,%eax,4),%eax  # z[dig]
```

- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at `4*%eax + %edx`
- Use memory reference `(%edx,%eax,4)`

Referencing Examples

`zip_dig cmu;`

1	5	2	1	3

16 20 24 28 32 36

`zip_dig mit;`

0	2	1	3	9

36 40 44 48 52 56

`zip_dig ucb;`

9	4	7	2	0

56 60 64 68 72 76

Reference	Address	Value	Guaranteed?
`mit[3]`			
`mit[5]`			
`mit[-1]`			
`cmu[15]`			

Referencing Examples

`zip_dig cmu;`

1	5	2	1	3

16 20 24 28 32 36

`zip_dig mit;`

0	2	1	3	9

36 40 44 48 52 56

`zip_dig ucb;`

9	4	7	2	0

56 60 64 68 72 76

Reference	Address	Value	Guaranteed?
`mit[3]`	`36 + 4* 3 = 48`	3	Yes
`mit[5]`	`36 + 4* 5 = 56`	9	No
`mit[-1]`	`36 + 4*-1 = 32`	3	No
`cmu[15]`	`16 + 4*15 = 76`	??	No

- No bound checking
- Out of range behavior implementation-dependent
- No guaranteed relative allocation of different arrays

Array Loop Example

- **Original**

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

- **Transformed**
  - As generated by GCC
  - Eliminate loop variable `i`
  - Convert array code to pointer code
  - Express in do-while form (no test at entrance)

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while (z <= zend);
  return zi;
}
```


Array Loop Implementation (IA32)

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

```
  # %ecx = z
  xorl %eax,%eax
  leal 16(%ecx),%ebx
.L59:
  leal (%eax,%eax,4),%edx
  movl (%ecx),%eax
  addl $4,%ecx
  leal (%eax,%edx,2),%eax
  cmpl %ebx,%ecx
  jle .L59
```


Array Loop Implementation (IA32)

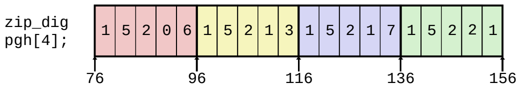
- **Registers**
  - `%ecx z`
  - `%eax zi`
  - `%ebx zend`
- **Computations**
  - `10*zi + *z` implemented as `*z + 2*(zi+4*zi)`
  - `z++` increments by 4

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

```
  # %ecx = z
  xorl %eax,%eax            # zi = 0
  leal 16(%ecx),%ebx        # zend  = z+4
.L59:
  leal (%eax,%eax,4),%edx   # 5*zi
  movl (%ecx),%eax          # *z
  addl $4,%ecx              # z++
  leal (%eax,%edx,2),%eax   # zi = *z + 2*(5*zi)
  cmpl %ebx,%ecx            # z : zend
  jle .L59                  # if <= goto loop
```

Nested Array Example

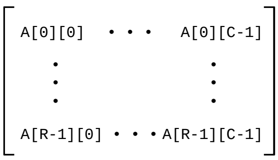
```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



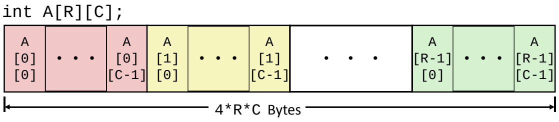
- **"`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"**
  - Variable **pgh**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**'s, allocated contiguously
- **"Row-Major" ordering of all elements guaranteed**

Multidimensional (Nested) Arrays

- **Declaration**
  T **A**[*R*][*C*];
  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes
- **Array Size**
  - *R* * *C* * *K* bytes
- **Arrangement**
  - Row-Major Ordering



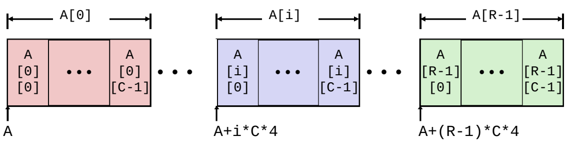
`int A[R][C];`



Nested Array Row Access

- **Row Vectors**
  - **`A[i]`** is array of *C* elements
  - Each element of type *T* requires *K* bytes
  - Starting address **A +** *i* * (*C* * *K*)

`int A[R][C];`



Nested Array Row Access Code

```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

- **What data type is `pgh[index]`?**
- What is its starting address?

```
  # %eax = index
  leal (%eax,%eax,4),%eax
  leal pgh(,%eax,4),%eax
```

Will disappear
Blackboard?

Nested Array Row Access Code

```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

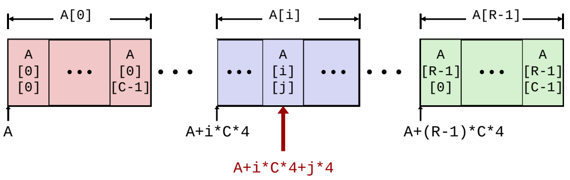
```
  # %eax = index
  leal (%eax,%eax,4),%eax  # 5 * index
  leal pgh(,%eax,4),%eax   # pgh + (20 * index)
```

- **Row Vector**
  - **`pgh[index]`** is array of 5 **`int`**'s
  - Starting address **`pgh+20*index`**
- **IA32 Code**
  - Computes and returns address
  - Compute as **`pgh + 4*(index+4*index)`**

Nested Array Row Access

- **Array Elements**
  - **`A[i][j]`** is element of type *T*, which requires *K* bytes
  - Address **`A +`** $i * (C * K) + j * K = A + (i * C + j) * K$

```
int A[R][C];
```



Two dimensional array “facts”

- `slot_type A[R][C];`
- Let K = type size (4 for int, 8 for double, 1 for char, etc.)
- Let R = 1st dimension (number of rows)
- Let C = 2nd dimension (number of columns)
- Size of row = K*C (cell size * number of cells per row)
- Size of A = (K*C)*R (row size * number of rows)
- A is a pointer to the beginning of the whole table
- A[i] is a pointer to the beginning of row i = A + (i*K*C)
- A[i][j] is the entry in row i, column j. It can be found at address A + (i*K*C) + j*K
- What is the value of A+1?

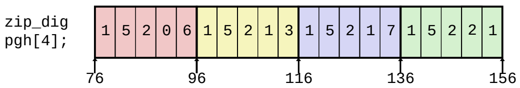
Nested Array Element Access Code

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

```
 # %ecx = dig
 # %eax = index
 leal 0(,%ecx,4),%edx         # 4*dig
 leal (%eax,%eax,4),%eax      # 5*index
 movl pgh(%edx,%eax,4),%eax   # *(pgh + 4*dig + 20*index)
```

- **Array Elements**
  - **`pgh[index][dig]`** is **`int`**
  - Address: **`pgh + 20*index + 4*dig`**
- **IA32 Code**
  - Computes address **`pgh + 4*dig + 4*(index+4*index)`**
  - **`movl`** performs memory reference

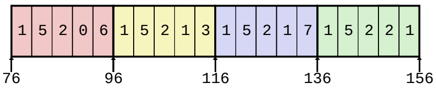
Strange Referencing Examples



Reference	Address	Value	Guaranteed?
`pgh[3][3]`	Will disappear		
`pgh[2][5]`			
`pgh[2][-1]`			
`pgh[4][-1]`			
`pgh[0][19]`			
`pgh[0][-1]`			

Strange Referencing Examples

```
zip_dig
pgh[4];
```



Reference	Address	Value	Guaranteed?
`pgh[3][3]`	`76+20*3+4*3 = 148`	`2`	Yes
`pgh[2][5]`	`76+20*2+4*5 = 136`	`1`	Yes
`pgh[2][-1]`	`76+20*2+4*-1 = 112`	`3`	Yes
`pgh[4][-1]`	`76+20*4+4*-1 = 152`	`1`	Yes
`pgh[0][19]`	`76+20*0+4*19 = 152`	`1`	Yes
`pgh[0][-1]`	`76+20*0+4*-1 = 72`	`??`	No

- Code does not do any bounds checking
- Ordering of elements within array guaranteed

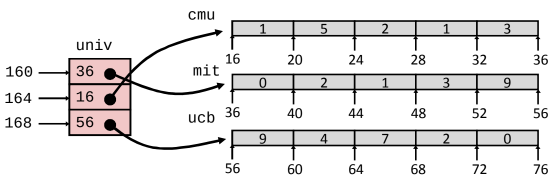
Systems I

Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- **Variable `univ` denotes array of 3 elements**
- **Each element is a pointer**
  - 4 bytes
- **Each pointer points to array of `int`'s**



Systems I

Element Access in Multi-Level Array

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

```
# %ecx = index
# %eax = dig
leal 0(,%ecx,4),%edx
movl univ(%edx),%edx
movl (%edx,%eax,4),%eax
```

Will disappear
Blackboard?

Element Access in Multi-Level Array

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

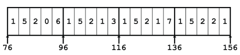
```
  # %ecx = index
  # %eax = dig
  leal 0(,%ecx,4),%edx     # 4*index
  movl univ(%edx),%edx     # Mem[univ+4*index]
  movl (%edx,%eax,4),%eax  # Mem[...+4*dig]
```

- **Computation (IA32)**
  - Element access **`Mem[Mem[univ+4*index]+4*dig]`**
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

Array Element Accesses

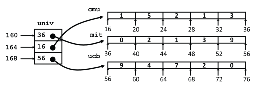
Nested array

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



Multi-level array

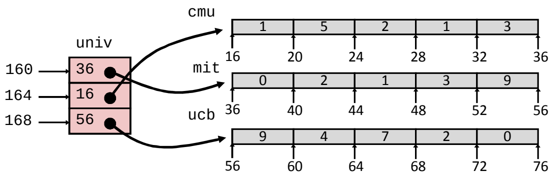
```
int get_univ_digit
  (int index, int dig)
{
  return univ[index][dig];
}
```



C Access looks similar, but underlying computation is different:

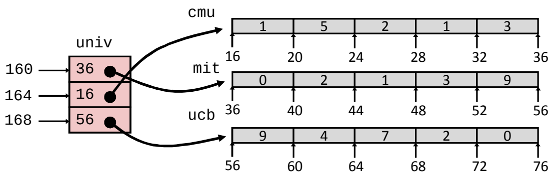
`Mem[pgh+20*index+4*dig]`   `Mem[Mem[univ+4*index]+4*dig]`

Strange Referencing Examples



Reference	Address	Value	Guaranteed?
`univ[2][3]`	Will disappear		
`univ[1][5]`			
`univ[2][-1]`			
`univ[3][-1]`			
`univ[1][12]`			

Strange Referencing Examples



Reference	Address	Value	Guaranteed?
`univ[2][3]`	`56+4*3 = 68`	`2`	Yes
`univ[1][5]`	`16+4*5 = 36`	`0`	No
`univ[2][-1]`	`56+4*-1 = 52`	`9`	No
`univ[3][-1]`	`??`	`??`	No
`univ[1][12]`	`16+4*12 = 64`	`7`	No

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed

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, int i, int 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
 (int n, int *a, int i, int j)
{
  return a[IDX(n,i,j)];
}
```

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

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


16 X 16 Matrix Access

- **Array Elements**
  - Address **A +** *i* * (*C* * *K*) + *j* * *K*
  - C = 16, K = 4

```
/* Get element a[i][j] */
int fix_ele(fix_matrix a, int i, int j) {
  return a[i][j];
}
```

```
  movl  12(%ebp), %edx     # i
  sall  $6, %edx           # i*64
  movl  16(%ebp), %eax     # j
  sall  $2, %eax           # j*4
  addl  8(%ebp), %eax      # a + j*4
  movl  (%eax,%edx), %eax  # *(a + j*4 + i*64)
```

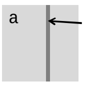
n X n Matrix Access

- **Array Elements**
  - Address **A** + *i* * (*C* * *K*) + *j* * *K*
  - C = n, K = 4

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

```
  movl   8(%ebp), %eax      # n
  sall   $2, %eax           # n*4
  movl   %eax, %edx         # n*4
  imull  16(%ebp), %edx     # i*n*4
  movl   20(%ebp), %eax     # j
  sall   $2, %eax           # j*4
  addl   12(%ebp), %eax     # a + j*4
  movl   (%eax,%edx), %eax  # *(a + j*4 + i*n*4)
```


Optimizing Fixed Array Access



j-th column

```
#define N 16
typedef int fix_matrix[N][N];
```

```
/* Retrieve column j from array */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

- **Computation**
  - Step through all elements in column j
- **Optimization**
  - Retrieving successive elements from single column

Optimizing Fixed Array Access

- **Optimization**
  - Compute ajp = &a[i][j]
    - Initially = a + 4*j
    - Increment by 4*N

```
/* Retrieve column j from array */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

Register	Value
%ecx	ajp
%ebx	dest
%edx	i

```
.L8:                            # loop:
   movl  (%ecx), %eax           #   Read *ajp
   movl  %eax, (%ebx,%edx,4)    #   Save in dest[i]
   addl  $1, %edx               #   i++
   addl  $64, %ecx              #   ajp += 4*N
   cmpl  $16, %edx              #   i:N
   jne   .L8                    #   if !=, goto loop
```

Systems I

Optimizing Variable Array Access

- Compute ajp = &a[i][j]
  - Initially = a + 4*j
  - Increment by 4*n

Register	Value
%ecx	ajp
%edi	dest
%edx	i
%ebx	4*n
%esi	n

```
/* Retrieve column j from array */
void var_column
  (int n, int a[n][n],
   int j, int *dest)
{
  int i;
  for (i = 0; i < n; i++)
    dest[i] = a[i][j];
}
```

```
.L18:                         # loop:
   movl  (%ecx), %eax         #   Read *ajp
   movl  %eax, (%edi,%edx,4)  #   Save in dest[i]
   addl  $1, %edx             #   i++
   addl  $ebx, %ecx           #   ajp += 4*n
   cmpl  $edx, %esi           #   n:i
   jg    .L18                 #   if >, goto loop
```

Systems I

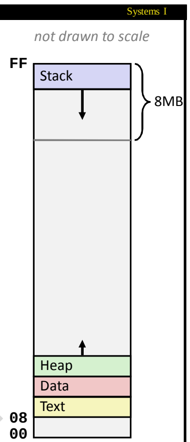
Rest of Today

- **Memory layout**
- Buffer overflow, worms, and viruses

Systems I

IA32 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
- **Heap**
  - Dynamically allocated storage
  - When call `malloc(), calloc(), new()`
- **Data**
  - Statically allocated data
  - E.g., arrays & strings declared in code
- **Text**
  - Executable machine instructions
  - Read-only



Upper 2 hex digits = 8 bits of address

Memory Allocation Example

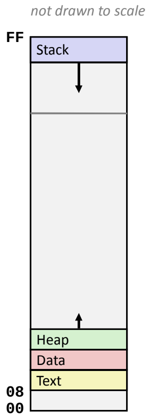
```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

Where does everything go?

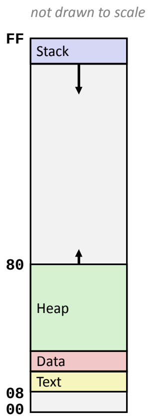


IA32 Example Addresses

address range ~2^{32}

$esp	0xffffbcd0
p3	0x65586008
p1	0x55585008
p4	0x1904a110
p2	0x1904a008
&p2	0x18049760
beyond	0x08049744
big_array	0x18049780
huge_array	0x08049760
main()	0x080483c6
useless()	0x08049744
final malloc()	0x006be166

malloc() is dynamically linked
address determined at runtime

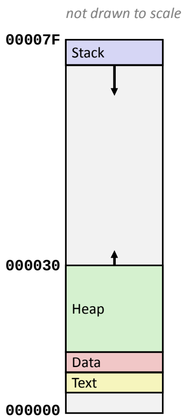


x86-64 Example Addresses

address range ~2^{47}

$rsp	0x7ffffff8d1f8
p3	0x2aaabaadd010
p1	0x2aaaaaadc010
p4	0x000011501120
p2	0x000011501010
&p2	0x000010500a60
beyond	0x000000500a44
big_array	0x000010500a80
huge_array	0x000000500a50
main()	0x000000400510
useless()	0x000000400500
final malloc()	0x00386ae6a170

malloc() is dynamically linked
address determined at runtime



C operators

Operators	*Associativity*
`() [] -> .`	left to right
`! ~ ++ -- + - * & (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= != <<= >>=`	right to left
`,`	left to right

- **`->` has very high precedence**
- **`( )` has very high precedence**
- **monadic `*` just below**

C Pointer Declarations: Test Yourself!

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

C Declarations (see book and WWW)

`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

Avoiding Complex Declarations

- Use `typedef` to build up the declaration
- Instead of `int (*(*x[3])())[5]`:

```
typedef int fiveints[5];
typedef fiveints* p5i;
typedef p5i (*pfr_p5is)();
pfr_p5is x[3];
```

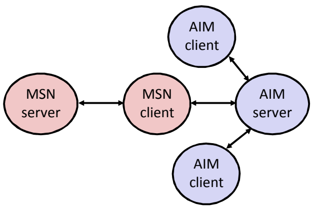
- x is an array of 3 elements, each of which is a pointer to a function returning an array of 5 ints

Rest of Today

- Memory layout
- Buffer overflow, worms, and viruses

Internet Worm and IM War

- **November, 1988**
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?
- **July, 1999**
  - Microsoft launches MSN Messenger (instant messaging system).
  - Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



Internet Worm and IM War (cont.)

- **August 1999**
  - Mysteriously, Messenger clients can no longer access AIM servers.
  - Microsoft and AOL begin the IM war:
    - AOL changes server to disallow Messenger clients
    - Microsoft makes changes to clients to defeat AOL changes.
    - At least 13 such skirmishes.
  - How did it happen?
- **The Internet Worm and AOL/Microsoft War were both based on *stack buffer overflow* exploits!**
    - many Unix functions do not check argument sizes.
    - allows target buffers to overflow.

String Library Code

- **Implementation of Unix function `gets()`**

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  - No way to specify limit on number of characters to read
- **Similar problems with other Unix functions**
  - **`strcpy`**: Copies string of arbitrary length
  - **`scanf`, `fscanf`, `sscanf`,** when given **`%s`** conversion specification

Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
int main()
{
  printf("Type a string:");
  echo();
  return 0;
}
```

```
unix>./bufdemo
Type a string:1234567
1234567
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

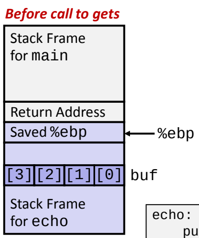
```
unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
```

Buffer Overflow Disassembly

```
080484f0 <echo>:
 80484f0:  55                push   %ebp
 80484f1:  89 e5             mov    %esp,%ebp
 80484f3:  53                push   %ebx
 80484f4:  8d 5d f8          lea    0xfffffff8(%ebp),%ebx
 80484f7:  83 ec 14          sub    $0x14,%esp
 80484fa:  89 1c 24          mov    %ebx,(%esp)
 80484fd:  e8 ae ff ff ff    call   80484b0 <gets>
 8048502:  89 1c 24          mov    %ebx,(%esp)
 8048505:  e8 8a fe ff ff    call   8048394 <puts@plt>
 804850a:  83 c4 14          add    $0x14,%esp
 804850d:  5b                pop    %ebx
 804850e:  c9                leave
 804850f:  c3                ret

 80485f2:  e8 f9 fe ff ff    call   80484f0 <echo>
 80485f7:  8b 5d fc          mov 0xfffffffc(%ebp),%ebx
 80485fa:  c9                leave
 80485fb:  31 c0             xor    %eax,%eax
 80485fd:  c3                ret
```


Buffer Overflow Stack

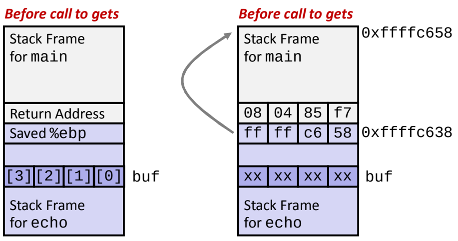



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp              # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx              # Save %ebx
    leal  -8(%ebp),%ebx     # Compute buf as %ebp-8
    subl  $20, %esp         # Allocate stack space
    movl  %ebx, (%esp)      # Push buf on stack
    call  gets              # Call gets
    . . .
```

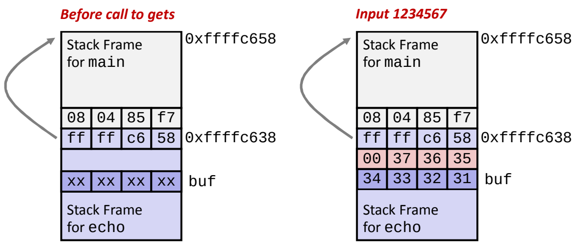

Buffer Overflow Stack Example

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x $ebp
$1 = 0xffffc638
(gdb) print /x *(unsigned *)$ebp
$2 = 0xffffc658
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f7
```



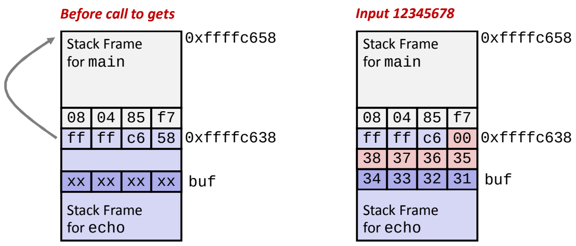

```
80485f2:call 80484f0 <echo>
80485f7:mov  0xfffffffc(%ebp),%ebx # Return Point
```

Buffer Overflow Example #1



Overflow buf, but no problem

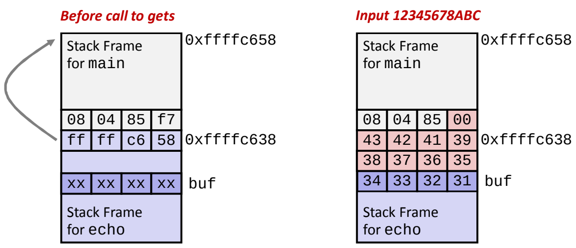
Buffer Overflow Example #2



Base pointer corrupted

```
. . .
804850a: 83 c4 14   add    $0x14,%esp   # deallocate space
804850d: 5b         pop    %ebx         # restore %ebx
804850e: c9         leave               # movl %ebp, %esp; popl %ebp
804850f: c3         ret                 # Return
```


Buffer Overflow Example #3



Return address corrupted

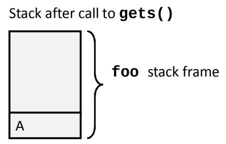
```
80485f2: call 80484f0 <echo>
80485f7: mov  0xfffffffc(%ebp),%ebx # Return Point
```

Malicious Use of Buffer Overflow

```
void foo(){
  bar();
  ...
}
```

return address A

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

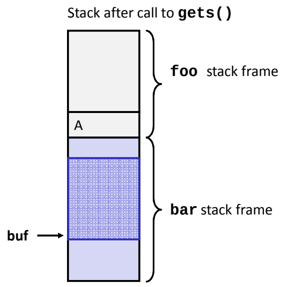


Malicious Use of Buffer Overflow

```
void foo(){
  bar();
  ...
}
```

return address A

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

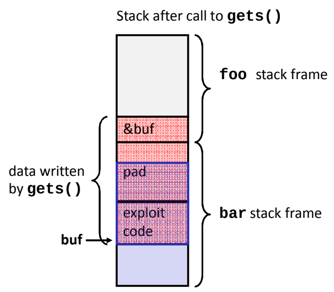


Malicious Use of Buffer Overflow

```
void foo(){
  bar();
  ...
}
```

return address A

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



- **Input string contains byte representation of executable code**
- **Overwrite return address with address of buffer**
- **When `bar()` executes `ret`, will jump to exploit code**

Exploits Based on Buffer Overflows

- ***Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines***
- **Internet worm**
  - Early versions of the finger server (fingerd) used **`gets()`** to read the argument sent by the client:
    - **`finger droh@cs.cmu.edu`**
  - Worm attacked fingerd server by sending phony argument:
    - **`finger`** ***`"exploit-code padding new-return-address"`***
    - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

Exploits Based on Buffer Overflows

- ***Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines***
- **IM War**
  - AOL exploited existing buffer overflow bug in AIM clients
  - exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server.
  - When Microsoft changed code to match signature, AOL changed signature location.


```
Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you
might find interesting because you are an Internet security expert with
experience in this area. I have also tried to contact AOL but received
no response.

I am a developer who has been working on a revolutionary new instant
messaging client that should be released later this year.
...
It appears that the AIM client has a buffer overrun bug. By itself
this might not be the end of the world, as MS surely has had its share.
But AOL is now *exploiting their own buffer overrun bug* to help in
its efforts to block MS Instant Messenger.
....
Since you have significant credibility with the press I hope that you
can use this information to help inform people that behind AOL's
friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com
```

It was later determined that this email originated from within Microsoft!

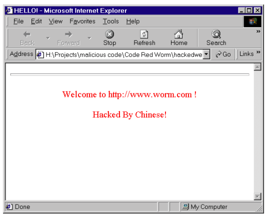
Code Red Worm

- **History**
  - June 18, 2001. Microsoft announces buffer overflow vulnerability in IIS Internet server
  - July 19, 2001. over 250,000 machines infected by new virus in 9 hours
  - White house must change its IP address. Pentagon shut down public WWW servers for day
- **When We Set Up CS:APP Web Site**
  - Received strings of form

```
GET
  /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
  N....NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN%u909
  0%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u
  6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003%u8b0
  0%u531b%u53ff%u0078%u0000%u00=a
HTTP/1.0" 400 325 "-" "-"
```


Code Red Exploit Code

- **Starts 100 threads running**
- **Spread self**
  - Generate random IP addresses & send attack string
  - Between 1st & 19th of month
- **Attack www.whitehouse.gov**
  - Send 98,304 packets; sleep for 4-1/2 hours; repeat
    - Denial of service attack
  - Between 21st & 27th of month
- **Deface server's home page**
  - After waiting 2 hours



Code Red Effects

- **Later Version Even More Malicious**
  - Code Red II
  - As of April, 2002, over 18,000 machines were infected
  - Was still spreading
- **Paved Way for NIMDA**
  - Variety of propagation methods
  - One was to exploit vulnerabilities left behind by Code Red II
- **ASIDE (security flaws start at home)**
  - .rhosts used by Internet Worm
  - Attachments used by MyDoom

Avoiding Overflow Vulnerability

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small!
*/
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- **Use library routines that limit string lengths**
  - `fgets` instead of `gets`
  - strncpy instead of `strcpy`
  - Don't use `scanf` with `%s` conversion specification
    - Use `fgets` to read the string
    - Or use `%ns` where `n` is a suitable integer

System-Level Protections

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for hacker to predict beginning of inserted code
- **Nonexecutable code segments**
  - In traditional x86, can mark region of memory as either "read-only" or "writeable"
    - Can execute anything readable
  - Add explicit "execute" permission

```
unix> gdb bufdemo
(gdb) break echo

(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638

(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08

(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```


Worms and Viruses

- **Worm: A program that**
  - Can run by itself
  - Can propagate a fully working version of itself to other computers
- **Virus: Code that**
  - Add itself to other programs
  - Cannot run independently
- **Both are (usually) designed to spread among computers and to wreak havoc**