Pointers and Arrays
Every pointer has a type

Indicates what kind of thing the pointer points to

You can cast pointers to be pointers to other types.
  Example: The return value of `malloc` must be cast

Every pointer has a value

  The address of an object (of a particular type)
  All pointers are 8 bytes for x86-64
    8 bytes = 64 bits
The “address-of” operator: &

Can use & on all “lvalues”

Anything that can appear on the left-hand side of an assignment

The “dereference” operator: *

Result is a value having type associated with pointer

Arrays and pointers closely related.

Name of array can be referenced as if it were a pointer

Array referencing equivalent to pointer arithmetic and dereferencing  

\[ a[3] == *(a+3) \]
Pointers and arrays review

\texttt{array\ \ z[10]}

- \texttt{z[i]} returns the \textit{i\textsuperscript{th}} element of array \texttt{z}
- \texttt{&z[i]} returns the \textbf{address} of the \textit{i\textsuperscript{th}} element of array \texttt{z}
- \texttt{z} alone returns address the array begins at
  - = the address of the \textit{0\textsuperscript{th}} element of array \texttt{z}
  - = \texttt{&z[0]}

\begin{verbatim}
int* ip;
int z[10];
ip = z;    /* equivalent to \texttt{ip = \&z[0]}; */
z[5] = 3;  /* equivalent to \texttt{ip=\&z[5]; *ip=3}; */
\end{verbatim}
Pointers and arrays review

Recall: pointer arithmetic is based on type of pointer

```c
char* cp1;
int* ip1;
double* dp1;

cp1++;  // Increments address by 1
ip1++;  // Increments address by 4
dp1++;  // Increments address by 8
```
Array allocation

Type ArrayName [Length];

Contiguously allocated region of Length × sizeof(Type) bytes

char string[12];

int val[5];

double a[3];

char *p[3];
Pointer arithmetic with arrays

As a result of contiguous allocation

- Elements are accessed by scaling the index by the size of the element and adding to the starting address
- Assembly: scaled index memory mode

<table>
<thead>
<tr>
<th>Array</th>
<th>Element Size</th>
<th>Total Size</th>
<th>Start Address</th>
<th>Element $i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>12</td>
<td>$x_A$</td>
<td>$x_A + i$</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>32</td>
<td>$x_B$</td>
<td>$x_B + 4i$</td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td>48</td>
<td>$x_C$</td>
<td>$x_C + 8i$</td>
</tr>
<tr>
<td>D</td>
<td>8</td>
<td>40</td>
<td>$x_D$</td>
<td>$x_D + 8i$</td>
</tr>
</tbody>
</table>
Array Accessing Example

```c
int myNums[5];

int get_num (int z[], int num {  
    return z[num];  
}
```

Assembly Code uses scaled, indexed mode

```assembly
# On entry:  
#    %rdi = z  
#    %rsi = num  
movl (%rdi,%rsi,4),%eax
```

Base + Index × size

```assembly
%rdi + %rsi × 4  
(%rdi,%rsi,4)
```
### Practice Problem

```c
short S [7];
short *T [3]
short **U [6];
```

<table>
<thead>
<tr>
<th>Array</th>
<th>Element Size</th>
<th>Total Size</th>
<th>Start Address</th>
<th>Element $i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td>$x_S$</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td>$x_T$</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td>$x_U$</td>
<td></td>
</tr>
</tbody>
</table>
### Practice Problem

```c
short S[7];
short *T[3]
short **U[6];
```

<table>
<thead>
<tr>
<th>Array</th>
<th>Element Size</th>
<th>Total Size</th>
<th>Start Address</th>
<th>Element $i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>2</td>
<td>14</td>
<td>$x_S$</td>
<td>$x+2i$</td>
</tr>
<tr>
<td>T</td>
<td>8</td>
<td>24</td>
<td>$x_T$</td>
<td>$x+8i$</td>
</tr>
<tr>
<td>U</td>
<td>8</td>
<td>48</td>
<td>$x_U$</td>
<td>$x+8i$</td>
</tr>
</tbody>
</table>
Array Loop Example

```c
void zincr(int z[5]) {
    size_t i;
    for (i = 0; i < 5; i++)
        z[i]++;
}
```

zincr:
```
    # %rdi = z
    movl $0, %eax              # i = 0
    jmp .L3                    # goto test
.L4:                        # loop:
    addl $1, (%rdi,%rax,4)    # z[i]++
    addq $1, %rax             # i++
.L3:                        # test:
    cmpq $4, %rax             # i:4
    jbe .L4                   # if <=, goto loop
rep; ret
```
Arrays as function arguments

The basic data types in C are passed by value. What about arrays?

Example:

```c
int exp[32000000];
int x = foo(exp);
```

What is the declaration of the function `foo`?

```c
int foo(int* f) { ... }
```

The name of an array is equivalent to what?

- Pointer to the first element of array!
- Arrays are passed by reference
Arrays of pointers

Arrays of pointers are quite common in C

Example: print out name of month given its number

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

char *monthName(int n) {
    static char *name[] = {
        "Illegal month", "January", "February", "March",
        "April", "May", "June", "July", "August",
        "September", "October", "November", "December"
    }
    return ( n < 1 || n > 12 ) ? name[0] : name[n];
}

int main(int argc, char *argv[]) {
    if (2 != argc) {
        fprintf(stderr, "Usage: %s <int>\n", argv[0]);
        return 0;
    }
    printf("%s\n", monthName(atoi(argv[1])));
    return 0;
}
```
argv

Command line arguments are passed in the `argv` array

Prototype for main routine

```c
int main(int argc, char *argv[]);
```

`argv` is an array of what kind of things?

Can be declared like this

```c
int main(int argc, char **argv);
```

```c
argv → &argv[0]
argv[0] → char*
```
Problem

Consider the following code

```c
char *pLines[3];
char *a = "abc";
char *d = "def";
char *g = "ghi";

pLines[0] = a;
pLines[2] = g;
```

What are the types and values of

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pLines</code></td>
<td>char **</td>
</tr>
<tr>
<td><code>pLines[0]</code></td>
<td>char *</td>
</tr>
<tr>
<td><code>*pLines</code></td>
<td>char *</td>
</tr>
<tr>
<td><code>*pLines[0]</code></td>
<td>char</td>
</tr>
<tr>
<td><code>**pLines</code></td>
<td>char</td>
</tr>
<tr>
<td><code>pLines[0][0]</code></td>
<td>char</td>
</tr>
</tbody>
</table>
Array access in C

Arrays can be accessed two ways

Via index \( a[4] \)
Via pointer arithmetic \( *(a+4) \)

```
int a[10];
```

\[
(a+N) == &a[N]  
*(a+N) == a[N]
\]
### Array access examples

```c
int val[5];
```

#### val:

```
1 5 2 1 3
x  x+4  x+8  x+12  x+16  x+20
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td>x</td>
</tr>
<tr>
<td>val+3</td>
<td>int *</td>
<td>x+12</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td>x+8</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>?</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td>x+4i</td>
</tr>
</tbody>
</table>
Arrays in Assembly

Arrays typically have very regular access patterns

Optimizing compilers are very good at optimizing array indexing code

Unfortunately, assembly code may not look at all like the source!
### Decimal conversion code

- Takes 5 integers between 0-9 and produces their base-10 value

```c
int decimal5 (int *x) {
    int i;
    int val = 0;
    for (i = 0; i < 5; i++)
        val = (10 * val) + x[i];
    return val;
}
```

```c
int decimal5_opt (int *x) {
    int val = 0;
    int *xend = x + 4;
    do {
        val = (10 * val) + *x;
        x++;  
    } while (x <= xend);
    return val;
}
```
```c
int decimal5_opt (int *x) {
    int val = 0;
    int *xend = x + 4;
    do {
        val = (10 * val) + *x;
        x++;
    } while (x <= xend);
    return val;
}
```
Multi-Dimensional Arrays

```c
int myArr[R][C];
```

An $R \times C$ matrix

- $R$ rows, $C$ elements per row

The dimensions of an array must be declared constants

- $R$ and $C$ must be `#define` constants
  
  Compiler must be able to generate proper indexing code

Can also have higher dimensions: `int my3dArr[R][C][D];`

**Multidimensional arrays in “C” are stored in “row major” order**

Data grouped by rows

- All elements of a given row are stored contiguously
  
  $A[0][*] = \text{in contiguous memory followed by } A[1][*]$

  The last dimension is the one that varies the fastest with linear access through memory

Important to know for performance!
Multi-Dimensional Array Access

Type \(A[R][C]\);
\(R = \# \text{ of rows}, \ C = \# \text{ of columns}, \ T = \text{type (which has size } K)\)

What is the size of a row in \(A\)?
\(C \times K\)

What is the address of \(A[2][5]\)?
\(A + 2 \times C \times K + 5 \times K\)

What is the address of \(A[i][j]\) given in \(A, C, K, i,\) and \(j\)?
\(A + (i \times C \times K) + (j \times K)\)

<table>
<thead>
<tr>
<th>(A[0][0])</th>
<th>(A[0][1])</th>
<th>(A[0][C-1])</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A[1][0])</td>
<td>(A[1][1])</td>
<td>(A[1][C-1])</td>
</tr>
<tr>
<td>(A[R-1][0])</td>
<td>(A[R-1][1])</td>
<td>(A[R-1][C-1])</td>
</tr>
</tbody>
</table>
Multi-Dimensional Array Access

Example: Where is $A[i][j]$ stored?

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

Size of each row: $C \times 4$

![Diagram showing multi-dimensional array access](image)
Multi-Dimensional Array Access

Example: Where is $A[i][j]$ stored?

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

Size of each row: $C*4$

![Diagram showing the storage of a two-dimensional array](image)
Multi-Dimensional Array Access

Example: Where is $A[i][j]$ stored?

```plaintext
int A[R][C];
Size of each row: C*4
```

![Diagram of multi-dimensional array access](image-url)
Multi-Dimensional Array Access

Example: Where is $A[i][j]$ stored?

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

Size of each row: C*4

```
A[0][0] ... A[0][C-1]
... ...
A[i][j] ...
A[R-1][C-1] ... A[R-1][C-1]
```

$A$ + $(i*C*4)$  

$A$ + $((R-1)*C*4)$
Multi-Dimensional Array Access

Example: Where is $A[i][j]$ stored?

```c
int A[R][C];
Size of each row: C*4
```

![Diagram showing array access]

- $A[0][0]$ to $A[0][C-1]$
- $A[i][j]$ is at $A+(i*C*4) + (j*4)$
- $A[R-1][C-1]$
Watch Out For Indexing Errors

```c
int A[12][13]; // A has 12 rows of 13 ints each
```

Will the C compiler permit us to do this?

```c
int x = A[3][26];
```

What will happen?

Indexing calculation is done assuming a 12x13 array

\[
A + (C*i + j) * K = A + (13*3 + 26) * 4 = A + (13*5 + 0) * 4
\]

Same as \( A[5][0] \)

What about this?

```c
int x = A[14][2];
```

\[\textbf{C does not check array bounds!}\]

(Contrast this to managed languages)
Improving Code Efficiency

Fixed sized arrays are easy for the compiler to optimize
  But resulting assembly code can be difficult to understand.

Can we write code that compiles more efficiently?

Example: Compute the dot-product

```c
#define N 16
typedef int fix_matrix[N][N]
fix_matrix A, B;
```

![Diagram of arrays A and B with index i and k]
Dot Product

for i,k...

\[ C[i,k] = 0; \]

for all j...

\[ C[i,k] += A[i,j] \times B[j,k] \]
```c
#define N 16
typedef int fix_matrix[N][N];

int fix_prod_ele (fix_matrix A, fix_matrix B, int i, int k)
{
    int j;
    int result = 0;
    for (j = 0; j < N; j++)
        result += A[i][j] * B[j][k];
    return result;
}

int fix_prod_ele_opt(fix_matrix A, fix_matrix B, int i, int k)
{
    int *Aptr = &A[i][0];
    int *Bptr = &B[0][k];
    int cnt = N - 1;
    int result = 0;
    do {
        result += (*Aptr) * (*Bptr);
        Aptr += 1;
        Bptr += N;
        cnt--;
    } while (cnt >= 0);
    return result;
}
```
#define N 16
typedef int fix_matrix[N][N];

int fix_prod_ele (fix_matrix A, fix_matrix B, int i, int k)
{
    int j;
    int result = 0;

    for (j = 0; j < N; j++)
        result += A[i][j] * B[j][k];

    return result;
}

int fix_prod_ele_opt(fix_matrix A, fix_matrix B, int i, int k)
{
    int *Aptr = &A[i][0];
    int *Bptr = &B[0][k];
    int cnt = N - 1;
    int result = 0;

    do {
        result += (*Aptr) * (*Bptr);
        Aptr += 1;
        Bptr += N;
        cnt--;
    } while (cnt >= 0);

    return result;
}
Dynamically Allocated Arrays

What if we don’t know the dimensions for our array at compile-time?

The compiler cannot generate multi-dimensional indexing code unless dimensions are known at compile-time

Solution: Perform indexing / address calculations in C code

```c
typedef int *VarMatrix;

VarMatrix is a pointer to an int
Can also be a pointer to a matrix of ints!
```

How to allocate an one of these, of dimension n x n:

```c
VarMatrix newVarMatrix(int n) {
    return (VarMatrix) malloc(sizeof(int) * n*n);
}
VarMatrix m = newVarMatrix(n);
```
Accessing Dynamic Arrays

Must do the indexing explicitly

Write the C code for a function that returns \( A[i][j] \)

```c
int getElt(VarMatrix A, int i, int j, int n)
{
    ... 
}

x = getElt (myMat, 4, 33, n);
```

A points to an \( n \times n \) matrix of integers.
The dimension \( n \) is an argument.
We want the value of \( A[i][j] \).

Memory \[ A+4*(n*i + j) \]