Pointers and Structures

Chapter 4

The Pointer

A pointer is a variable which stores an address as opposed to data itself.

To declare a pointer, we use the asterisk (*) after the type of data we wish to point to.

The allocation size for a pointer depends on the size of the addresses of the machine and not the type of data it is pointing to.

```
char c;       //allocates space for a char
char *ptrc; //allocates space for an
// address which will “point” to a char
```

Using Pointers

Before a pointer can be used, it must be initialized by setting it equal to an address. This can be done by using the ampersand (`&`) symbol.

```
ptrc = &c; // Initialize pointer to point to c
```

Once it has been initialized, it can be used like the variable it points to by using the `*` to dereference it.

```
*ptrc = 'A'; // Puts an 'A' where ptrc
             // "points" to, i.e. c.
```

Memory Map with Pointers

Consider the following code:

```
char c;     // Allocates space for a char, 1 B
char *ptrc;// Allocates space for pointer to a char, 4 B

short int s;     // Allocates space for a short int, 2 B
short int *ptrs;// A pointer to a short int, 4 B;

int i;         // Allocates space for an int, 4 B
int *ptri;     // Space for pointer to an int, 4 B;

double d;        // Allocates space for a double, 8 B
double *ptrd;    // Space for pointer to a double, 4 B;
```

Example Memory Map

It could* have the following Memory Map:

Label	Address Space
`c`	1000
`ptrc`	1001-1004
`s`	1005-1006
`ptrs`	1007-1010
`i`	1011-1014
`ptri`	1015-1018
`d`	1019-1026
`ptrd`	1027-1030

Example Memory Map 2

Or in a different look:

<table>
<tr><th>Addr</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>1000</td><td>c</td><td colspan="4">ptrc</td><td colspan="2">s</td><td></td></tr>
<tr><td>1008</td><td colspan="3">ptrs</td><td colspan="4">i</td><td></td></tr>
<tr><td>1016</td><td colspan="3">ptri</td><td colspan="5">d</td></tr>
<tr><td>1024</td><td colspan="3">d</td><td colspan="4">ptrd</td><td></td></tr>
</table>

How are addresses stored?

Pointers store addresses. But addresses are simply integers. It is how they are used by C which makes them different.

addresses.c

```
int main(void)
{ char c = 'A';
  char *ptrc = &c;

  printf("&c = %d %u 0x%X &ptrc = %d %u 0x%X\n",
                    &c, &c, &c, &ptrc, &ptrc, &ptrc);
  printf("ptrc = %d %u %X *ptrc = %c = %d = 0x%X\n",
                    ptrc, ptrc, ptrc, *ptrc, *ptrc, *ptrc);
}
```

What would the following do?

```
printf("*c = %d\n", *c);
```

Pointer Example

```
char c[4];
char *ptrc;
c[2] = 'A';
ptrc = &c[2];
*ptrc = 'B';
```

Label	Address	Data
c[0]	1000	
c[1]	1001	
c[2]	1002	~~'A'~~ 'B'
c[3]	1003	
ptrc	1004-7	1002

Pointer Example 2

```
char c[4];
char *ptrc;
```

What does this do?

```
ptrc = c;       // equivalent to ptrc = &c[0]
*ptrc = 'C';
```

Label	Address	Data
`c[0]`	1000	'C'
`c[1]`	1001	
`c[2]`	1002	'~~A~~' B'
`c[3]`	1003	
`ptrc`	1004-7	10~~0~~2 1000

Pointer Example 3

```
char c[4];
char *ptrc;
```

So what does this do?

```
ptrc = c;
ptrc[3] ='D'; // equivalent to *(ptrc+3) = 'D'
```

Label	Address	Data
c[0]	1000	'C'
c[1]	1001	
c[2]	1002	~~'A'~~ 'B'
c[3]	1003	'D'
ptrc	1004-7	~~1002~~ 1000

Pointer Arithmetic

Pointer arithmetic involves adding or subtracting values from a pointer (or an address). Consider the following code:

pointerarithmetic.c

```
char *ptrc, short int *ptri;
long *ptrl, double *ptrd;
struct S {char c[128];} *ptrs;
```

```
ptrc = (char *)0x1000;          ++ptrc;   +1
ptri = (short int *) 0x1000; ++ptri;   +2
ptrl = (long *) 0x1000;         ++ptrl;   +4
ptrd = (double *) 0x1000;       ++ptrd;   +8
ptrs = (struct S *) 0x1000;     ++ptrs;   +128
                                          +0b1000 0000
```

Pointer Arithmetic

What do these instructions do?

```
char c[10] = {'A','B','C','D','E','F','G','H','I','J'};
char *ptrc = c
ptrc += 5;
ptrc -= 2;
*ptrc++;
(*ptrc)++;
++*ptrc;
++(*ptrc);
++(*ptrc--);
--(*ptrc++);
(*ptrc--)++;
(*ptrc++)--;
++*--ptrc;
--*++ptrc;
```


pointerarithmetic2.c

Pointer Arithmetic

What do these instructions do?

```
char c[10] = {'A','B','C','D','E','F','G','H','I','J'};
char *ptrc = c
ptrc += 5;                                   'F'
```

Pointer Arithmetic

What do these instructions do?

```
char c[10] = {'A','B','C','D','E','F','G','H','I','J'};
char *ptrc = c;
ptrc += 5;
ptrc -= 2;                        'D'
```

Pointer Arithmetic

What do these instructions do?

```
char c[10] = {'A','B','C','D','E','F','G','H','I','J'};
char *ptrc = c;
ptrc += 5;
ptrc -= 2;
*ptrc++;
```

Dereference then increment pointer

Compiler warning: value computed is not used

Pointer Arithmetic

What do these instructions do?

```
char c[10] = {'A','B','C','D','E','F','G','H','I','J'};
char *ptrc = c;
ptrc += 5;
ptrc -= 2;
*ptrc++;
(*ptrc)++;                    'F'
```

Dereference then increment value

Pre-fix and post-fix have higher precedence than `*` or `&`

Pointer Arithmetic

What do these instructions do?

```
char c[10] = {'A','B','C','D','E','F','G','H','I','J'};
char *ptrc = c;
ptrc += 5;
ptrc -= 2;
*ptrc++;
(*ptrc)++;
++*ptrc;
```

Reset Letters after each operation

Same result

'F'

Evaluate operators `++ -- *` and `&` right to left

Pointer Arithmetic

What do these instructions do?

```
char c[10] = {'A','B','C','D','E','F','G','H','I','J'};
char *ptrc = c;
ptrc += 5;
ptrc -= 2;
*ptrc++;
(*ptrc)++;
++*ptrc;
++(*ptrc);
```

Same result

'F'

Pointer Arithmetic

What do these instructions do?

```
char c[10] = {'A','B','C','D','E','F','G','H','I','J'};
char *ptrc = c;
ptrc += 5;
ptrc -= 2;
*ptrc++;
(*ptrc)++;
++*ptrc;
++(*ptrc);
++(*ptrc--);                         'F'
```

Pre increment before post decrement

Shaky ground! Advise: Don't stack up multiple side effects

Pointer Arithmetic

What do these instructions do?

```
char c[10] = {'A','B','C','D','E','F','G','H','I','J'};
char *ptrc = c;
ptrc += 5;
ptrc -= 2;
*ptrc++;
(*ptrc)++;
++*ptrc;
++(*ptrc);
++(*ptrc--);
--(*ptrc++);                'C'
```

Crazy!

Pre decrement before post increment

Pointer Arithmetic

What do these instructions do?

```
char c[10] = {'A','B','C','D','E','F','G','H','I','J'};
char *ptrc = c;
ptrc += 5;
ptrc -= 2;
*ptrc++;
(*ptrc)++;
++*ptrc;
++(*ptrc);
++(*ptrc--);
--(*ptrc++);
(*ptrc--)++;                    'F'
```

I don't know what is going on here!

Multiple post: increment value, decrement pointer

Pointer Arithmetic

What do these instructions do?

```
char c[10] = {'A','B','C','D','E','F','G','H','I','J'};
char *ptrc = c;
ptrc += 5;
ptrc -= 2;
*ptrc++;
(*ptrc)++;
++*ptrc;
++(*ptrc);
++(*ptrc--);
--(*ptrc++);
(*ptrc--)++;
(*ptrc++)--;
```

'C'

At least it is consistent

Multiple post: decrement value, increment pointer

Pointer Arithmetic

What do these instructions do?

```
char c[10] = {'A','B','C','D','E','F','G','H','I','J'};
char *ptrc = c;
ptrc += 5;
ptrc -= 2;
*ptrc++;
(*ptrc)++;
++*ptrc;
++(*ptrc);
++(*ptrc--);
--(*ptrc++);
(*ptrc--)++;
(*ptrc++)--;
++*--ptrc;
--*++ptrc;
```

pointerarithmetic2.c

Multiple pre: decrement pointer, increment value

Multiple pre: increment pointer, decrement value

Pointer Arithmetic

How about these?

`pointerarithmetic2.c`

```
ptrc = ptrc * 5;

ptrc = ptrc / 3;
```

Compiler error: Invalid operands to binary operator

Pointer arithmetic only involves addition and subtraction. Multiplication and Division are meaningless and not allowed.

Is the following pointer arithmetic?

```
char c[100];
char *ptrc = c + 6;
```

Yes: definition of pointer and initialization of value

And this?

```
ptrc = (char *)(0xF2371289) + 6;
```

Yes: assignment of value

Pointer Arithmetic

What you need to know?

```
char c[10] = {'A','B','C','D','E','F','G','H','I','J'};
char *ptrc = c;
ptrc += 5;
ptrc -= 2;
*ptrc++;
(*ptrc)++;
++*ptrc;
++(*ptrc);
++(*ptrc--);
--(*ptrc++);
(*ptrc--)++;
(*ptrc++)--;
++*--ptrc;
--*++ptrc;
```

Very frequent and topic of many machine problems

Increment pointer

Useful on occasion: see examples

Increment value

Never!

pointerarithmetic2.c

Four Versions of `strcpy`

Array subscript version

```
void strcpy(char *s, char *t)
{
    int i=0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

Assignment then logical test

Pointer version

```
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Four Versions of `strcpy`

```
void strcpy(char *s, char *t)
{   while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Pointer version

```
    void strcpy(char *s, char *t)
    {
        while ((*s++ = *t++) != '\0')
            ;
    }
```

Postfix applied after copy

```
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++))
        ;
}
```

Null is false

What is Wrong With `swap`?

```
void swap(int a, int b) {
     int temp = b;
     b = a;
     a = temp;
}
int main() {
     int x = 5, y = 7;
     swap(x, y);
     printf("after swap x = %d, y = %d\n", x, y);
}
```

Memory space for main

Address	Symbol	Memory Contents
0xe048	x	5
0xe04c	y	7

Memory space for swap

Address	Symbol	Memory Contents
0xf01c		--- 7
0xf018	b	--- 5
0xf02c	temp	7

void

Pass By Address (Reference)

```
void swap(int *a, int *b) {
    int temp = *b;
    *b = *a;
    *a = temp;
}
```

* Means at the address given by

```
int main() {
    int x = 5, y = 7;
    swap( &x, &y);
    printf("after swap x = %d, y = %d\n", x, y);
}
```

& Means the address of

Memory space for main

Address	Symbol	Memory Contents
0xe048	x	~~5~~ 7
0xe04c	y	~~7~~ 5

Memory space for swap

Address	Symbol	Memory Contents
0xf01c	a	
0xf018	b	
0xf02c	temp	

Use of Pointers

The most common use of pointers is to allow a function to modify a variable passed to it.

```
void ChangeA(int *a){(*a)++;}
void ChangeB(int b) {  b++; }
int main(void)
{ int a = 10, b = 20;
  ChangeA(&a);  ChangeB(b);
  printf("a = %d\n\n", a);
  printf("b = %d\n\n", b);
}
```

byref.c

Passing Values Back from a Function

- Call by value passes parameters into function
  - Copies value
  - Return limited to one item
- Call by reference
  - Use & (the address of)
  - Creates a pointer in the function to dereference with *
- Pass in pointer to dynamic memory block
- Use structures with call-by-value and return

Pointers Indexed Like Arrays

Once a pointer is declared, it can be indexed with the `[]` operator just like an array can.

ptrindex.c

```
int main(void)
{ int Array[9] = {0,1,2,3,4,5,6,7,8}, i;
  int *ptrA = Array;
  printf("Array[7] = %d\n", Array[7]);
  printf("*(&Array[7]) = %d\n", *(&Array[7]));
  printf("*(Array + 7) = %d\n", *(Array + 7));
  printf("*(ptrA + 7) = %d\n", *(ptrA + 7));
  printf("ptrA[7] = %d\n", ptrA[7]);
```

Indexed Addresses

Can any address be indexed with the `[]` operator just like an array can?

Yes, but only if precedence order is correct

addressindexing.c

```
int main(void)
{ char c, dummy;
   &c[1] = 'A';
   (&c)[1] = 'B'; √
   (char *)&c[1] = 'C';
   ((char *)&c)[1] = 'D'; √
   (char *)(&c)[1] = 'E';
   (0x100)[1] = 'F';
   (char *)(0x1000)[1] = 'G';
   ((char *)0x1000)[1] = 'H'; √
   *(&c + 1) = 'I';
```

Error: subscripted value is neither array nor pointer nor vector

Warning: cast to pointer from integer of different size
Error: lvalue required as left operand of assignment

Compiles (but seg faults!)

Clearer?

Dynamic Allocation with Pointers

Pointers can be used to allocate memory at runtime. The pointer is used to store where the newly allocated memory is located.

```
#include <stdlib.h>

int main(void)
{ int i, *ptri;

  ptri = (int *)malloc(10*sizeof(int));

  for (i=0; i<10; i++) ptri[i] = i;

  free(ptri);
}
```

malloc.c

Observing the O/S with `malloc`

Where are these arrays allocated?

Malloc2.c

```
#include <stdlib.h>

const int size[4] = {2, 5, 10, 23};

int main(void)
{ int i, j;
  char *ptr[4];

  for (i=0; i<4; i++)
  { ptr[i]=(char *)malloc(size[i]*sizeof(char));
     printf("ptr[%d] = %u\n", i, ptr[i]);
     for (j=0; j<size[i]; j++)
     { ptr[i][j] = i*10 + j;
     }
  }
```

```
ptr[0] = 3413936
ptr[1] = 3414040
ptr[2] = 3414056
ptr[3] = 3414080
```

“Memory Leakage”

If you forget to free memory, or assign a new memory allocation to a pointer before freeing it, the memory allocated becomes unusable.

```
#include <stdlib.h>

int main(void)
{ int *ptri;

  ptri = (int *)malloc(10*sizeof(int));
  ptri = (int *)malloc(100*sizeof(int));

  free(ptri);
}
```

Other Allocation Functions

The `calloc()` function works like `malloc()`, but it zeroes out the memory upon allocation.

Calloc.c

```
#include <stdlib.h>

int main(void)
{ int i, *ptri;

  ptri = (int *)calloc(10, sizeof(int));

  free(ptri);
}
```

Other Allocation Functions

The `realloc()` function works like `malloc()`, but it allows the user to increase the size of an allocation without destroying values saved in the initial allocation.

Realloc.c

```
#include <stdlib.h>
int main(void)
{ int i, int *ptri;
  ptri = (int *)malloc(10 * sizeof(int));
  for (i=0; 10; i++) ptri[i] = i;
  ptri = (int *)realloc(ptri, 20 * sizeof(int));
  for (i=0; i<20; i++)
    printf("ptr[%d] = %d\n", i, ptri[i]);
  free(ptri);
}
```

Dynamically Allocating Multi-dimensional Arrays

Multi-dimensional arrays can be allocated dynamically by the use of pointers and `malloc()`.

MultiMalloc.c

```
#include <stdlib.h>
#define ROWS 10
#define COLS 20
int main(void)
{ int i, **Array;

  Array = (int **)malloc(ROWS*sizeof(int*));

  for (i=0; i<ROWS; i++)
    Array[i] = (int *)malloc(COLS*sizeof(int));
```

Dynamically Allocating Multi-dimensional Arrays

What's the difference in the following code?

```
for (i=0; i<ROWS; i++)
  for (j=0; j<COLS; j++)
    printf("%d\n", Array[i][j]);

for (i=0; i<ROWS; i++)
  for (j=0; j<COLS; j++)
    printf("%d\n", *(*(Array + i) + j));
```

MultiMalloc2.c

Label	Address Space	Value
A	4210704-4210707	4012864
*(A+0)=A[0]	4012864-4012867	4012968
*(A+1)=A[1]	4012868-4012871	4012984
*(A+2)=A[2]	4012872-4012875	4013000
((A+0)+0)=A[0][0]	4012968	0
((A+0)+1)=A[0][1]	4012969	1
((A+0)+2)=A[0][2]	4012970	2
((A+0)+3)=A[0][3]	4012971	3
((A+1)+0)=A[1][0]	4012984	10
((A+1)+1)=A[1][1]	4012985	11
((A+2)+0)=A[2][0]	4013000	20

Three-Dimensional Dynamic Arrays

How would we create a three-dimensional dynamically allocated array?

MultiMalloc3.c

```
#define DEPTH 10
#define ROWS 20
#define COLS 30
int main(void)
{ int d, r, c, ***Array;

 A = (int ***)malloc(DEPTH*sizeof(int**));

  for (d=0; d<DEPTH; d++)
  { A[d] = (int **)malloc(ROWS*sizeof(int*));

    for (r=0; r<ROWS; r++)
      A[i][j] = (int *)malloc(COLS*sizeof(int));
  }
  /* … */
```

Three-Dimensional Dynamic Arrays

How do we free the memory for these multidimensional arrays?

We have to `free()` them in reverse: ***Why?***

```
for (d=0; d<DEPTH; d++)
  { for (r=0; r<ROWS; r++)
    { free (A[d][r]);
    }
  }

  for (d=0; d<DEPTH; d++)
  { free (A[d]);
  }

  free(A);
```

`MultiMalloc3.c`

Pointer Arithmetic Revisited

What is the difference between the following statements?

PointerArithmetic3.c

```
int M[5][4][3];

int main(void)
{
  printf("&M[0][0][0]
  printf("&M + 1 =
  printf("&M[0] + 1 =         %u\n", &M[0] + 1);
  printf("&M[0][0] + 1 =      %u\n", &M[0][0] + 1);
  printf("&M[0][0][0] + 1 = %u\n", &M[0][0][0] + 1);
```

```
&M[0][0][0] =       4210784
&M + 1 =            4211024
&M[0] + 1 =         4210832
&M[0][0] + 1 =      4210796
&M[0][0][0] + 1 =   4210788
```

Static Vs. Dynamic

What are the advantages of Dynamic memory allocation over static allocation?

Speed?

Size?

`DynamicVsStatic.c`

Variable Indices

How would we create a dynamic array with indices `1` to `n` instead of `0` to `n-1`?

```
int *intVector_Create(int low_index, int high_index)
{  int *ptri;
                                        Vector.c
   if (high_index > low_index)
   {  ptri = (int *)malloc((high_index - low_index + 1)
                                           * sizeof(int));
      if (ptri == NULL)
      { return NULL;
      }
      else return ptri - low_index;
   }
   else return NULL;
}
```

Warning about `malloc()`

- Any system call such as `malloc()` should check to insure the system granted the request.

```
ptri = (int *)malloc(1000 * sizeof(int));
if (ptri == NULL)
{ printf("Call to malloc() unsuccessful.\n");
  return 1;
}
```

Structures

A *structure* (using `struct`) allows for the grouping of separate variables under one name. They are akin to arrays, except they are not indexed by integers, but by labels.

```
struct School
{ char Name[32];
  char Mascot[32];
  char Location[32];
  char Color[2][32];
  unsigned int Enrollment;
  unsigned int AvgSAT;
};
struct School clemson;
```

Defines a template but does not allocate space

Declare a variable of new type

`struct.c`

`struct2.c`

Memory Map `struct.c`

Use dot (`.`) to access a member in a structure

Label	Address	Value
`i`	100	
`ptrc`	104	
`clemson.Name[32]`	108-139	Clemson University
`clemson.Mascot[32]`	140-161	Tiger
`clemson.Location[32]`	162-193	Clemson
`clemson.Color[0][32]`	194-225	Burnt Orange
`clemson.Color[1][32]`	226-257	Northwest Purple
`clemson.Enrollment`	258	20768
`clemson.AvgSAT`	262	1287

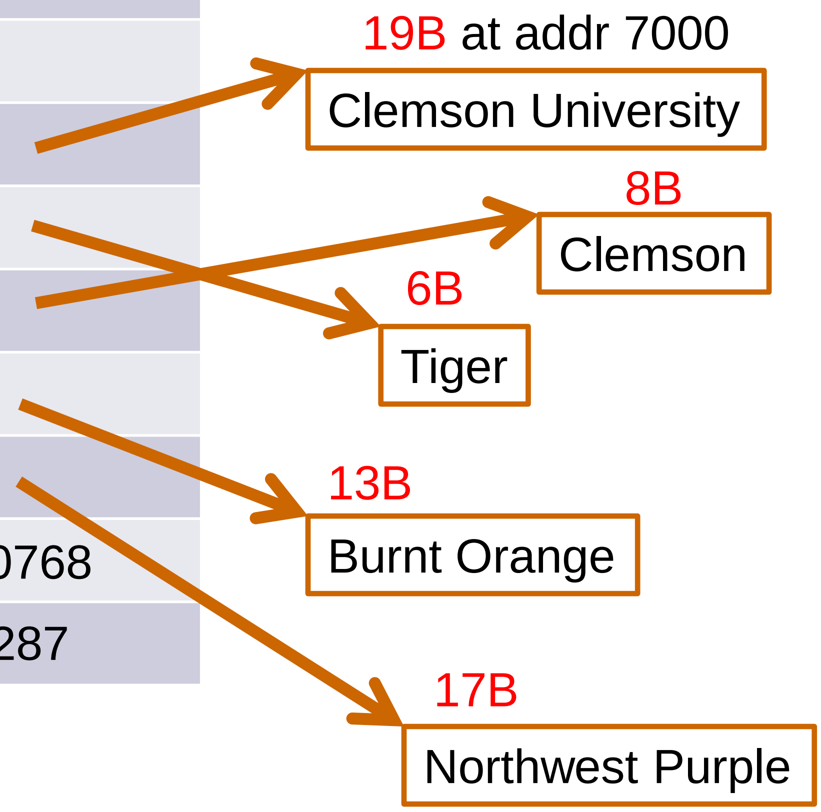
Can copy structure: `clemson2 = clemson`;

Can pass a copy to a function

Name is label for values not pointer

Memory Map `struct2.c`

Label	Address	Value
`i`	100	
`ptrc`	104	
`clemson.Name`	108	
`clemson.Mascot`	112	
`clemson.Location`	116	
`clemson.Color[0]`	120	
`clemson.Color[1]`	124	
`clemson.Enrollment`	128	20768
`clemson.AvgSAT`	132	1287



Attributes

GCC has an `__attribute__` directive which can force the compiler to allocate structures with a certain data alignment.

http://gcc.gnu.org/onlinedocs/gcc/Type-Attributes.html

Arrays of Structures

You can declare arrays of structures just as you can declare arrays of any other type.

```
struct School
{ char Name[32];
  char Mascot[32];
  char Location[32];
  char Color[2][32];
  unsigned int Enrollment;
  unsigned int AvgSAT;
};
struct School acc[ACC_SCHOOLS];
```

structarray.c

Nested Structures

C allows for structures to be part of a structure.

structstruct.c

```
struct Point
{ float x, y, z;
};

struct Triangle
{ struct Point P1, P2, P3;
};

struct Triangle T;
```

Memory Map `structstruct.c`

`struct Triangle T1, T2;`

Label	Address	Value
`T1.P1.x`	108	1.4
`T1.P1.y`	112	3.1
`T1.P1.z`	116	-4.0
`T1.P2.x`	120	2.4
`T1.P2.y`	124	3.1
`T1.P2.z`	128	-4.0
`T1.P3.x`	132	
`T1.P3.y`	136	
`T1.P3.z`	140	5.0

`T1.P2 = T1.P1;`

`T1.P3 = T1.P2;`

`T2 = T1;`

Memory Map `structstruct.c`

```
struct Triangle T1, T2;
```

Label	Address	Value
`T1.P1.x`	108	1.4
`T1.P1.y`	112	3.1
`T1.P1.z`	116	-4.0
`T1.P2.x`	120	2.4
`T1.P2.y`	124	3.1
`T1.P2.z`	128	-4.0
`T1.P3.x`	132	2.4
`T1.P3.y`	136	3.1
`T1.P3.z`	140	5.0

```
T1.P2 = T1.P1;
```

```
memcpy(&T1.P2, &T1.P1, sizeof(struct Point));
```

```
T2 = T1;
```

```
memcpy(&T2, &T1, sizeof(struct Triangle));
```

Unions

Unions are "overlapping" Structures. They allow the user to reference the same memory area in different ways.

```
union Union
{ unsigned int Full;
  unsigned short int Half[2];
  unsigned char Byte[4];
};
int main(void)
{ union Union A;
  A.Full = 0x1234ABCD;
  printf("A = %08X\n", A.Full);
  printf("A = %02X %02X %02X %02X\n", A.Byte[0],
   A.Byte[1], A.Byte[2], A.Byte[3]);
```

Union.c

<table>
<tr><th colspan="4">Union</th></tr>
<tr><td colspan="4">Full</td></tr>
<tr><td colspan="2">Half[0]</td><td colspan="2">Half[1]</td></tr>
<tr><td>Byte[0]</td><td>Byte[1]</td><td>Byte[2]</td><td>Byte[3]</td></tr>
</table>

Bit Fields

Bitfields **allow the programmer to reference bits or groups of bits by name in a byte-addressable machine.**

```
struct Motors
{ unsigned Motor0 : 1;
  unsigned Motor1 : 1;
  unsigned Motor2 : 1;
  unsigned Motor3 : 1;
};

struct Nibbles
{ unsigned Nibble0 : 4;
  unsigned Nibble1 : 4;
};
```

Bitfields.c

Bitfields2.c

BitfieldsBCD.c

BitfieldsUnion.c

Pointers to Structures

C exhibits much of its power when we combine pointers and structures.

structptr.c

```
struct ComplexNumber
{ double Re, Im; };

int main(void)
{ struct ComplexNumber *z1, *z2;

  z1 = (struct ComplexNumber *)
        malloc(sizeof(struct ComplexNumber));
  z1->Re = 1;
  z1->Im = -3;
```

Notice pointers not local variables

Requires malloc (so can pass to functions)

A new syntax to combine pointers and structures

Accessing Structures with Pointers

```
(*z1).Re = 1;
(*z1).Im = 1;
```

Equivalent

- Parentheses are necessary because the precedence of the structure member operator "●" is higher than "*"

- If **`ptr`** is a pointer to a structure, **`ptr->member_of_structure`** refers to a particular member

```
z1->Re = 1;
z1->Im = 1;
```

Pointers to Structures 2

Differences between structures and pointers to structures can be seen in the following program.

StructPtr2.c

```
struct T_Name
{ char *First;  char *Middle;  char *Last;
};
struct T_FullName
{ char *Title;  struct T_Name Name;  char *Suffix;
};
struct T_Person
{ struct T_FullName FullName;
  int Age;  char Sex;
  struct T_Person *BestFriend;
};
```

Memory Map `structptr2.c`

Label	Address	Value
`friend[0]`	108	9000
`friend[1]`	112	
`friend[2]`	116	
`friend[3]`	120	
`friend[4]`	124	
`friend[5]`	128	
`...`	132	

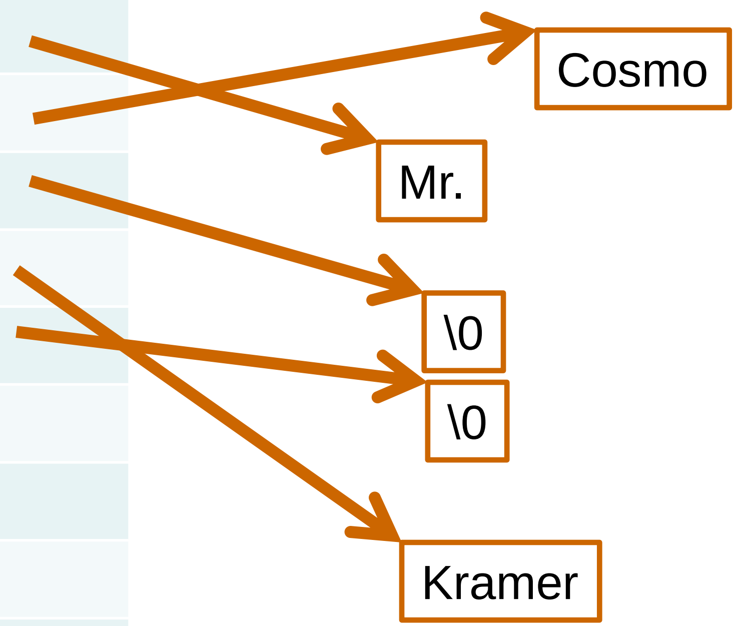
29B at address 9000

Label	Addr	Val
`FullName.Title`	9000	
`FullName.Name.First`	9004	
`FullName.Name.Middle`	9008	
`FullName.Name.Last`	9012	
`FullName.Suffix`	9016	
`Age`	9020	
`Sex`	9024	
`BestFriend`	9025	
	9029	

Memory Map `structptr2.c`

Label	Address	Value
`friend[0]`	108	9000
`friend[1]`	112	
`fr`		
`fr`		
`fr`		
`fr`		
`..`		

Label	Addr	Val
`FullName.Title`	9000	
`FullName.Name.First`	9004	
`FullName.Name.Middle`	9008	
`FullName.Name.Last`	9012	
`FullName.Suffix`	9016	
`Age`	9020	
`Sex`	9024	
`BestFriend`	9025	
	9029	



Passing `struct` Pointers by Reference to Functions

What if we want to use a function to modify the a `struct` *pointer?*

`StructPtr2.c`

```
void Person_Create(struct T_Person **P, const char *T, …)
{
  *P = (struct T_Person *)malloc(sizeof(struct T_Person));

  (*P)->FullName.Title = (char *)malloc(strlen(T)+1);
  strcpy((*P)->FullName.Title, T);
```

```
Person_Create(&friend[0], "Mr.", "Cosmo", "", "Kramer", "");
Person_Create(&friend[1], "Mr.", "George", "Louis", "Costanza", "");
```

Memory Map `structptr2.c`

Label	Address	Value
`friend[0]`	108	9000
`friend[1]`	112	9500
`fr`		
`fr`		
`fr`		
`fr`		
`..`		

Label	Addr	Val
`FullName.Title`	9	
`FullName.Name.First`	9	
`FullName.Name.Middle`	9	
`FullName.Name.Last`	9	
`FullName.Suffix`	9	
`Age`	9	
`Sex`	9	
`BestFriend`	9	
	9	

29B at addr 9500

Label	Addr	Val
FullName.Title	9500	
FullName.Name.First	9504	
FullName.Name.Middle	9508	
FullName.Name.Last	9512	
FullName.Suffix	9516	
Age	9520	
Sex	9524	
BestFriend	9525	
	9529	

Memory Map `structptr2.c`

Label	Address	Value
`friend[0]`	108	9000
`friend[1]`	112	9500
`friend[2]`	116	
`friend[3]`	120	

```
friend[0].BestFriend = friend[1];
```

Label	Addr	Val
FullName.Title	9500	
FullName.Name.First	9504	
FullName.Name.Middle	9508	
FullName.Name.Last	9512	
FullName.Suffix	9516	
Age	9520	
Sex	9524	
BestFriend	9525	
	9529	

Label	Add	
`FullName.Title`	900	
`FullName.Name.First`	900	
`FullName.Name.Middle`	900	
`FullName.Name.Last`	901.	
`FullName.Suffix`	901	
`Age`	9020	
`Sex`	9024	
`BestFriend`	9025	
	9029	

Kramer

Declaring with Typedef

```
typedef int Length;
```

Creates a new data type name

```
int main(void) {
   Length len, maxlen;
   Length *lengths[10];
}
```

Use Caps

```
typedef something YourDataTypeName;
```

Put your complicated structure definition here

Alternative form for arrays

```
typedef char AirportCode[4];

AirportCode my_airport = "GSP";
```

`typedef` Advantages

- Parameterize program against portability problems
  - Use typedef's for data types that may be machine-dependent
  - Ex: use for integer then pick **`short`**, **`int`**, or **`long`** for host machine
  - Standard library: `size_t` and `ptrdiff_t`
- Documentation – code easier to understand

Pointers to Functions

An array of function pointers can be used with an index to choose which function to execute.

New type definition called **`fptr`**

```
typedef void (*fptr)(void);
```

Return value

arguments

funcptr.c

```
fptr Func[5] =
{ PrintGoTigers,  PrintOrangeAndWhite,
  PrintReignSupremeAlway,  PrintGarnetAndBlack,
  PrintGoCocks
};
...
Func[i]() ;
```

`qsort()` and Function Pointers

Function pointers are variables which hold addresses to functions instead of data.

The library function `qsort()` uses a function pointer to pass the rules as to how a list is to be sorted.

```
void qsort(void *base, size_t nelem, size_t width,
               int (*fcmp)(const void *, const void *));
```

QSort1.c

QSort2.c

Practice with `structs` and pointers

```
struct point { int k[3]; };
struct TNode
{ int Mag;
  struct point P1;
  int *Link;
};

struct TNode P[2], *pNode;
int n[2];
struct TNode Node;
int i, j;
int *pp;
```

```
pp = &n[1] - 8;
*(pp + 3) = 11;
Node.P1.k[5] = 22;
P[1].Link = &P[0].Mag;
P[0].Link = n + 4;
Node.Link = &P[1].Mag;
pNode =(int)(&P[0] + 3);
P[0].P1.k[5] = &P + 2;
(*(pNode-2)).Mag = 33;
pNode->P1.k[0] = 44;
n[1] = (P + 1)->Mag + 1;
*(Node.Link - 2) = 55;
```

structs and pointers

```
struct point { int k[3]; };
struct TNode
{ int Mag;
  struct point P1;
  int *Link;
};
```

```
struct TNode P[2], *pNode;
int n[2];
struct TNode Node;
int i, j;
int *pp;
```

Label	Address	Value	Label	Address	Value
P[0].Mag	100		Node.Mag	152	
P[0].P1.k[0]	104		Node.P1.k[0]	156	
P[0].P1.k[1]	108		Node.P1.k[1]	160	
P[0].P1.k[2]	112		Node.P1.k[2]	164	
P[0].Link	116		Node.Link	168	
P[1].Mag	120		i	172	
P[1].P1.k[0]	124		j	176	
P[1].P1.k[1]	128		pp	180	
P[1].P1.k[2]	132			184	
P[1].Link	136			188	
pNode	140			192	
n[0]	144			196	
n[1]	148			200	

structs and pointers

```
struct TNode P[2], *pNode;
  int n[2];
  struct TNode Node;
  int i, j;
  int *pp;
```

```
n[0] = 10; n[1] = 20;
i = 1; j = 2;
pp = &n[1] - 8;
*(pp + 3) = 11;
Node.P1.k[5] = 22;
P[1].Link = &P[0].Mag
```

Label	Address	Value	Label	Address	Value
P[0].Mag	100		Node.Mag	152	
P[0].P1.k[0]	104		Node.P1.k[0]	156	
P[0].P1.k[1]	108		Node.P1.k[1]	160	
P[0].P1.k[2]	112		Node.P1.k[2]	164	
P[0].Link	116		Node.Link	168	
P[1].Mag	120		i	172	1
P[1].P1.k[0]	124		j	176	2 22
P[1].P1.k[1]	128	11	pp	180	116
P[1].P1.k[2]	132			184	
P[1].Link	136	100		188	
pNode	140			192	
n[0]	144	10		196	
n[1]	148	20		200	

structs and pointers 2

```
struct TNode P[2], *pNode;
  int n[2];
  struct TNode Node;
  int i, j;
  int *pp;
```

```
P[0].Link = n+4;
Node.Link = &P[1].Mag
pNode = (int)(&P[0] + 3);
P[0].P1.k[5] = &P + 2;
(*(pNode-2)).Mag = 33;
pNode->P1.k[0] = 44;
```

```
n[1] = (P+1)->Mag + 1;      *(Node.Link - 2) = 55;
```

Label	Address	Value	Label	Address	Value
P[0].Mag	100		Node.Mag	152	
P[0].P1.k[0]	104		Node.P1.k[0]	156	
P[0].P1.k[1]	108		Node.P1.k[1]	160	
P[0].P1.k[2]	112	55	Node.P1.k[2]	164	44
P[0].Link	116	160	Node.Link	168	120
P[1].Mag	120	33	i	172	1
P[1].P1.k[0]	124	180	j	176	22
P[1].P1.k[1]	128	11	pp	180	116
P[1].P1.k[2]	132			184	
P[1].Link	136	100		188	
pNode	140	160		192	
n[0]	144	10		196	
n[1]	148	20 34		200	

Pointer Arithmetic Revisited, Again

What is the address produced by the following pointer calculations?

```
struct
{ int a;
  int b[2];
  struct
  { int c;
    int d[3];
    struct
    { int e;
      int f[4];
    } T[4];
  } U[3];
} V[2];
```

PointerArithmetic4.c

```
&V + 1,                 &V[0] + 1
&V[0].a + 1,            &V[0].b + 1
&V[0].b[1] + 1,         &V[0].U
&V[0].U[1] + 1
&V[0].U[1].d + 1
&V[0].U[1].d[2]
&V[0].U[1].T + 1
&V[0].U[1].T[3]
&V[0].U[1].T[3].e + 1
&V[0].U[1].T[3].f + 1
&V[0].U[1].T[3].f[3] + 1
```