Making the case for Pointers:

Special Note: From here on in, Quizzes will now be spaced out every 5 weeks:
Quiz 3 will be on the day of class 18;
(reviewed at the end of class 17) –and–
Quiz 4 will be on the day of class 23;
(reviewed at the end of class 22)

- Consider a scenario in main where some data has been captured

```
byte data [500];

for(i=0;i<500;i++) data [i]=PINB;
```

- So that we have 500 data points stored in memory.
- Now, say we want to create a function that will replace every data point with its doubled value. Such a function would need to be designed to "double" 500 various data values in our scenario. What are the mechanism options for passing the values and the collecting the results?
- **Registers?**: This is out--obviously there are not enough registers to pass all the data to and from the function.
- **Stack?**: Consider this scenario:
  - the caller pushes 500 values to the stack;
  - the function process them and then push all 500 values back to the stack;
  - then the caller pops all 500 values from the stack?

  - Using the Stack is not a good approach; it represents a thousand or more unnecessary moves of data to and from the stack.
- There is a better way!

Pointers as a Solution:

- What if at the time of the call, we could just point the function to the place in memory where the data is stored and have it access and manipulate it directly?

- We can do this with variables called **pointers** which are variables in C for storing memory addresses and referencing memory locations.
- Rather than coping all the data, the location/address of the data can be copied to/from the function and the function can access it directly.

- Don't miss the assumption here. How is it possible to point to 500 data points using one variable? One way is to store the 500 data points in a contiguous block of memory. Then, the only information need is:
  1. the address of the first element and
  2. the number of bytes to advance in memory to find the next element.

- Arrays are set up this way in C. In C, a simple data array is stored in contiguous block of memory (so far as the program can tell since any memory paging is abstracted).
  1. The location of the array is tracked using the address of the first element, which is stored in the array variable itself.
  2. The type of the elements (int, char, float, etc…) defines the number of bytes needed to move forward after each access to find subsequent elements.

How Far To Advance Each Pointer?

- So, if you know data[0] exist at memory location 0x0F10, where is the second point?
- You might be tempted to say that if data[0] exists at memory address 0x0F10 then the next data point is at 0x0F11.
- You need to know:
  1. The number of bytes used for each element in the array
  2. How many bytes forward in memory the next location will be when incrementing the address by one
- Lets assume byte-addressable memory and that changing address by one moves to the next byte in memory.
- The programmer needs to know the size of each piece of data:
  - If it is one byte each then data[1] should be in 0x0F11.
  - If each data point was stored using 2 bytes, then data[1] should be in location 0x0F12.

Finding the Size of the Base Data Type

- So how is the size of the base data type known?
- It is important to think assembly here… There are several options:
- One option is that the array is passed to a function as two pieces of information at **RUN TIME**:
  1. starting address (addr)
  2. size in bytes per data point (size)
- At the level of assembly pseudo-code, **assuming parameters are passed on the stack**...

```
Pop size from stack to a register Rs
Pop address from stack to register pair Z
Loop:
  LD Rd, Z
  Do some processing using wordsize Rs
  ST Z, Rd
  ADD Rs to Z
Evaluate exit condition and potentially loop
```

Finding the Size at Compile Time

- It is a better situation where the data type and therefore **size** of each data point is determined at **COMPILE TIME**.

Evaluating at RUN TIME (previous slide)

```
Pop size from stack to a register Rs
Pop address from stack to register pair Z
Loop:
  LD Rd, Z
  Do some processing using wordsize Rs
  ST Z, Rd
  ADD Rs to Z
Evaluate exit condition and potentially loop
```

Evaluating at COMPILE TIME

```
Pop address from stack to register pair Z
Loop:
  LD Rd, Z
  Do some processing using wordsize SIZEOFTYPE
  ST Z, Rd
  ADDIW Z, SIZEOFTYPE
Evaluate exit condition and potentially loop
```

- So if the location of the first point of data in the array is known, then the second point in the array is also known (address of first point + size of datatype) – on so on for the rest of the array.
- Remember, the end of the array must always be encoded via some mechanism or convention: another variable, a termination value in the array, a predefined array length, etc...

Other Reasons to Use Pointers

- So, one reason to use pointers has been discussed: sharing large sets of data with functions without unnecessary copying (arrays have been discussed; other large data structures will be discussed later like structs, lists, trees etc..)

- Later, pointers will be used to keep track of and refer to dynamically allocated (run-time) arrays and structures (as opposed to arrays and variables that are defined at compile time and can have there location known at compile-time).

Syntax of Pointers in C

- Now some syntax in C is needed to provide a way to code all of this. Looking at some function prototypes and what parameters are pushed on the stack (assuming parameters are passed on the stack):

```
int myFunction1(int data);   // Single data value is pushed
int myFunction2(int data[]); // A single address is pushed
```

- The alternative syntax for the second is

```
int myFunction(int * data);  //A single address is pushed
```

  - `*` indicates that an address is being passed and that data is a "**pointer**" variable
  - The preceding type provided (**int**) tells the compiler how to treat the data accessed using that pointer variable and how to index into an array using that variable (This will be referred to as the objective type of the pointer)

Syntax of Pointers in C

- So, a new variable type has been defined: (`int *`).
- Example:

```
int count;
int * ptrCount;
```

- One can then "point to"/reference/store address of count like this:

```
ptrCount = & count;
```

where the `&` (ampersand) prefix means "address of"

- In this case, then, ptrCount can be used as a regular `int` variable through dereferencing.

```
(*ptrCount) = (*ptrCount) + 1; // modifies count
```

'`*`' is the dereference operator

Syntax of Pointers in C

- So, this variable can be used to modify other variables. It can also be passed to functions to do the same.

- Here is an example:

```
void SameMax( int * ptrA, int * ptrB){
if (*ptrA > *ptrB){
*ptrB = *ptrA;
} else {
*ptrA = *ptrB;
}
```

...somewhere in main...

```
int a = 1, b=2;
int * ptrInt;
ptrInt = & a;
```

- Now the pointer can be passed:
  - `SameMax(ptrInt, &b);` (note: same as →) `SameMax(&a, &b);`
  - Both result in `a=2, b=2` . `&a` is a way to directly create a pointer.

What is a Pointer?

- In a generic sense, a "pointer" tells where something can be found.
  - Addresses in the phone book
  - URLs for webpages
- In programming, a pointer variable contains the memory address of
  - A variable
  - An array
  - Dynamically allocated memory (covered in later slide sets)

Why Pointers?

- They allow for reference to large data structures in a compact way
- They facilitate sharing between different parts of programs
- They make it possible to get new memory dynamically as your program is running
- They make it easy to represent relationships among data items.

Pointer Caution:

- They are a powerful low-level feature
- Undisciplined use can be confusing and thus the source of subtle, hard-to-find bugs.
  - Program crashes
  - Memory leaks
  - Unpredictable results

C Pointer Variables

- To declare a pointer variable, two things must be done:
  1. Use the "*" (star) character to indicate that the variable being defined is a pointer type.
  2. Indicate the type of variable to which the pointer will point (the pointee). This is necessary because C provides operations on pointers (e.g., *, ++, etc) whose meaning depends on the type of the pointee.
- General declaration of a pointer

```
type *nameOfPointer;
```

C Pointer Declaration

- The declaration

```
int * ptrInt;
```

- defines the variable `ptrInt` to be a pointer to a variable of type `int`. `ptrInt` will contain the memory address of some `int` variable or `int` array. Read this declaration as
  - "`ptrInt` is a pointer to an `int`", or equivalently
  - "`*ptrInt` is an `int`"

- Caution --Be careful when defining multiple variables on the same line. In this definition

```
int *ptrInt, ptrInt2;
```

→ `ptrInt` is a pointer to an `int`, but `ptrInt2` is not!

C Pointer Operators

- The two primary operators used with pointers are `*` (star) and `&` (ampersand)
  - The `*` operator is used to declare pointer variables and to deference a pointer. "Dereferencing" a pointer means to use the value of the pointee.
  - The `&` operator gives the address of a variable.
    - Recall the use of `&` in scanf( )

Pointer Examples:

```
int x = 1, y = 2, z[10];
int *ip;     /* ip is a pointer to an int */
ip = &x;     /* ip points to (stores the memory address
                of) x */
y = *ip;     /* y is now 1, indirectly copied from x
                using ip */
*ip = 0;     /* x is now 0 */
ip = &z[5]; /* ip now points to z[5] */
```

- When in doubt about order of operations, or to increase readability and conveyance of intent, use parenthesis `ip = &(z[5]);`
- If ip points to **x**, then `*ip` can be used anywhere **x** can be used so in this example `*ip = *ip + 10;` and `x = x + 10;` are equivalent
- The unary operators (one operand) `*` and `&` are higher precedence than binary arithmetic operators (two operands). So `y = *ip + 1;` takes the value of the variable to which `ip` points, adds 1 and assigns it to `y`
- Similarly, the statements `*ip += 1;` and `++*ip;` and `(*ip)++;` all increment the variable to which ip points. (Note that the parenthesis are necessary in the last statement; without them, the expression would increment ip rather than what it points to since unary operators `*` and `++` associate from right to left.)

Pointer and Variable types

- Terminology will now be introduced for this class referring to the type being pointed to or the type of an array. This will now be referred to as the "**objective type**".
- The **objective type** of a pointer and the type of its pointee must match in assignments:

```
int a = 42;
int *ip;
double d = 6.34;
double *dp;
ip = &a;    /* ok --types match */
dp = &d;    /* ok */
ip = &d;    /* compiler error --type mismatch */
dp = &a;    /* compiler error */
```

More Pointer Code and Usage

- Use ampersand (**&**) to obtain the address of the pointee
- Use star (*****) to get / change the value of the pointee
- Use **%p** to print the value of a pointer with printf( )
- What is the output from this code?

```
int a = 1, *ptr1;

/* show value and address of a
** and value of the pointer */
ptr1 = &a ;
printf("a = %d, &a = %p, ptr1 = %p, *ptr1 = %d\n",
     a, &a, ptr1, *ptr1) ;

/* change the value of a by dereferencing ptr1
** then print again */
*ptr1 = 35 ;
printf("a = %d, &a = %p, ptr1 = %p, *ptr1 = %d\n", a,
&a, ptr1, *ptr1) ;
```

Output:

```
a = 1, &a = 0x7ffddc88b424, ptr1 = 0x7ffddc88b424, *ptr1 = 1
a = 35, &a = 0x7ffddc88b424, ptr1 = 0x7ffddc88b424, *ptr1 = 35
```

NULL

- **NULL** is a special value which may be assigned to a pointer
- **NULL** indicates that this pointer does not point to any variable ***(there is no pointee)***
- Often used when pointers are declared

```
int *pInt = NULL;
```

- Often used as the return type of functions that return a pointer to indicate function failure

```
int *myPtr = myFunction( );
if (myPtr == NULL){
  /* something bad happened */
}
```

- Dereferencing a pointer whose value is **NULL** **will** result in program termination -- at least in unix!

Pointers and Function Arguments

- Since C passes all function arguments "by value" there is no direct way for a function to alter a variable in the calling code.
- This version of the swap function doesn't work. *WHY NOT?*

```
/* calling swap from somewhere in main() */
int x = 42, y = 17;
Swap( x, y );

/* wrong version of swap */
void Swap (int a, int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

A corrected swap( )

- The desired effect can be obtained by passing pointers to the values to be exchanged.
- This is a very common use of pointers.

```
/* calling swap from somewhere in main( ) */
int x = 42, y = 17;
Swap( &x, &y );

/* correct version of swap */
void Swap (int *px, int *py)
{
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

More Pointer Function Parameters

- Passing the address of variable(s) to a function can be used to have a function "return" multiple values.
- The pointer arguments point to variables in the calling code which are changed ("returned") by the function.

```
void ConvertTime (int time, int *pHours, int *pMins)
{
  *pHours = time / 60;
  *pMins = time % 60;
}

int main( )
{
  int time, hours, minutes;
  printf("Enter a time duration in minutes: ");
  scanf ("%d", &time);
  ConvertTime (time, &hours, &minutes);
  printf("HH:MM format: %d:%02d\n", hours, minutes);
  return 0;
}
```

Another Exercise

- What is the output from this code?

```
void F (int a, int *b)
{
  a = 7 ;
  *b = a ;
  *b = 4 ;
  printf("%d, %d\n", a, *b)
;
  b = &a ;
}

int main()
{
  int m = 3, n = 5;
  F(m, &n) ;
  printf("%d, %d\n", m, n) ;
  return 0;
}
```

7, 4

3, 4

GeeksforGeeks {ide}

Code:

```
1  #include <stdio.h>
2
3  void F (int a, int *b)
4  {
5    a = 7 ;
6    *b = a ;
7    *b = 4 ;
8    printf("%d, %d\n", a, *b) ;
9    b = &a ;
10 }
11
12 int main()
13 {
14   int m = 3, n = 5;
15   F(m, &n) ;
16   printf("%d, %d\n", m, n) ;
17   return 0;
18 }
19
```

C | C++ | Java | Python 2.7 | Run

Output:

```
7, 4
3, 4
```

Pointers to `struct` (for reference; this slide content covered later)

```
/* define a struct for related student data */
typedef struct student {
  char name[50];
  char major [20];
  double gpa;
} STUDENT;

STUDENT bob = {"Bob Smith", "Math", 3.77};
STUDENT sally = {"Sally", "CSEE", 4.0};
STUDENT *pStudent; /* pStudent is a "pointer to struct
                      student" */

/* make pStudent point to bob */
pStudent = &bob;

/* use -> to access the members */
printf ("Bob's name: %s\n", pStudent->name);
printf ("Bob's gpa : %f\n", pStudent->gpa);

/* make pStudent point to sally */
pStudent = &sally;
printf ("Sally's name: %s\n", pStudent->name);
printf ("Sally's gpa: %f\n", pStudent->gpa);
```

- Note too that the following are equivalent. Why??

```
pStudent->gpa and (*pStudent).gpa /* the parentheses are
                                     necessary */
```

Pointers to `struct` - functions (for reference; this slide content covered later, too)

```
void PrintStudent(STUDENT *studentp)
{
  printf("Name : %s\n", studentp->name);
  printf("Major: %s\n", studentp->major);
  printf("GPA : %4.2f", studentp->gpa);
}
```

- Passing a pointer to a struct to a function is more efficient than passing the struct itself. Why is this true?