Pointers

Section 4.1

Introduction

- **Pointers** can be a challenging topic in C
- Conceptually, similar to reference types in Java or C#
- Nonetheless, important topic in C and widely used

Pointers

- A **pointer** is a variable that holds a memory address
- A pointer data type is denoted by the data type being pointed to and a *
  - **int ***
  - **double ***
  - **char ***

- Could think of pointer types as distinct data types for holding memory locations
  - ints hold whole numbers, floats hold real numbers, pointer types hold memory addresses
- * is notation associated with pointers
  - Nothing to do with multiplication

- To create a pointer variable, we put the data type, a *, and the name of the pointer variable:
  - ***dataType * pointerName*;**
- The * can go next to the data type or next to the variable name or have a space on each side:
  - ***dataType* pointerName*;**
  - ***dataType *pointerName*;**

```
int * ptr;

double * newPointer;

char * aPointer;
```

```
char c;
char *cp;

int i;
int *ip;

float f;
float *fp;

double d;
double *dp;
```

An int variable is 4 bytes. How many bytes is a pointer to an integer? In general, how many bytes is a pointer?

- Depends on the computer hardware
- For a 32-bit computer, each memory address is 32 bits (4 bytes) long, regardless of the data stored at the address
  - So a char * and a long * are the same size
- Reason why a 32-bit computer cannot have more than 4 GB of memory
  - Textbook assumes 32-bit systems, 64-bit systems common today

Variable	Address	Value
c	400	
cp	401 – 404	
i	405 – 408	
ip	409 – 412	
f	413 – 416	
fp	417 – 420	
d	421 – 428	
dp	429 – 432	

Reference Operator

- **&** is called the **reference operator**
- Even though it's an operator, let's pretend it's a function for a moment

- If we think of **&()** as a function, the input is a variable
  - **&(*variable*)**
- The output is the memory address of the variable
  - I.e., the expression evaluates to the memory address of the variable

- Due to order of operations, we generally don't actually need the parentheses when we use & in most cases
- So, we rarely put the parentheses and often just write
  - ***&variable***

- Since a pointer holds a memory address,
- Often use & to get the memory address of a variable to put in a pointer

```
int x = 5;

int * ptr;

ptr = &(x);
```

- To put a value in a pointer, often use the **&** operator:
  - ***pointerName = &variableName*;**
    - Notice we don't use a * here!
  - Puts the address of *variableName* into *pointerName*

Dereference Operator

- * is called the **dereference operator**
  - Also called the **indirection operator**
- Even though it's an operator, let's pretend it's a function for a moment

- If we think of ***()** as a function, the input is a memory address
  - ***(*memoryAddress*)**
- The output is the data at that memory address
  - I.e., the expression evaluates to the value stored at that memory location

- Due to order of operations, we generally don't actually need the parentheses when we use * in most cases
- So, we rarely put the parentheses and often just write
  - ***memoryAddress**

```c
int x = 5;

int * ptr;

ptr = &(x);

*(ptr) = 9;

printf("%d", *(ptr));
```

```c
int x = 5;

int * ptr;

ptr = &x;

*ptr = 9;

printf("%d", *ptr);
```

- To access the memory pointed to by a pointer, we use the **indirection operator** (also called **dereference operator**): ***
  - Allows us to set the value stored at the memory location the pointer points to
    - ***ip = 100;**
  - Allows us to retrieve the value stored in the variable that the pointer points to
    - **printf("%d", *ip);**

```
cp = &c;
ip = &i;
*ip = 42;
```

Variable	Address	Value
c	400	
cp	401 – 404	400
i	405 – 408	42
ip	409 – 412	405
f	413 – 416	
fp	417 – 420	
d	421 – 428	
dp	429 – 432	

- Any variable type's address can be held in a pointer
  - Includes memory address of array element

```
char ca[3];
char *cp;

ca[1] = 3;
cp = &(ca[1]);
*cp = 7;
```

Variable	Address	Value
ca[0]	400	
ca[1]	401	~~3~~ 7
ca[2]	402	
cp	403 – 406	401

- Since a memory address is really just an integer, it can be printed in a few ways
  - Signed integer
  - Unsigned integer
  - Hexadecimal
- Usually, addresses are written in hexadecimal

```
char a;
char *b;

a = 7;
b = &a;

printf("%d %u %p  value = %d", b, b, b, *b);
```

Pointer Arithmetic

- Since all pointers are the same size and just hold a memory location, why need to specify data type when declaring a pointer?
  - **int *ptr;**
- Pointer arithmetic!

- We can add and subtract from a pointer (memory location)

```
char ca[3];
char *cp;

int ia[3];
int *ip;

cp = &(ca[0]);
ip = &(ia[0]);
```

Variable	Address	Value
ca[0]	400	
ca[1]	401	
ca[2]	402	
cp	403 – 406	400
ia[0]	407 – 410	
ia[1]	411 – 414	
ia[2]	415 – 418	
ip	419 – 422	407

What does this do?

```
*(cp + 2) = 8;
*(ip + 2) = 33;
```

Variable	Address	Value
ca[0]	400	
ca[1]	401	
ca[2]	402	8
cp	403 – 406	400
ia[0]	407 – 410	
ia[1]	411 – 414	
ia[2]	415 – 418	33
ip	419 – 422	407

- Notice that..
  - Adding 2 to cp advanced 2 bytes
  - Adding 2 to ip advanced 8 bytes
- Pointer arithmetic takes into account the number of bytes in the data type being pointed to
  - Adding to a char pointer advances in units of 1 bytes
  - Adding to an int pointer advanced in units of 4 bytes

Void Pointers

- A special type of pointers are **void pointers**
- Void pointers hold a memory address, but the type of data at that address is not specified

```
void * ptr;

void * anotherPtr;
```

- Void pointers often used for rudimentary generics and polymorphic effects
- Void pointers cannot be dereferenced
- Often need to be cast to another type of pointer

Null Pointers

- Another special type of pointers are **null pointers**
- Null pointers don't point to anything
  - That is, the memory address they hold is the value **NULL**
- Often used as a default value for a pointer

```
int * ptr = NULL;

/* ... */

if (ptr != NULL)
{
    // Do something
}
```