C Pointers

CS 2060 Week 6

Prof. Jonathan Ventura

Pointers

- The **pointer** is one of C's most powerful and important features.
- They are also difficult to master and hard to debug.
- Pointers allow us to:
  - Effectively pass-by-reference
  - Pass functions as arguments to functions
  - Create dynamic data structures such as linked lists, trees

Pointers

- A pointer is a variable whose value is a *memory address*.
- The variable does not contain meaningful data itself, but instead **points** to where data is located in memory.

- Given the pointer (a memory address), we can **dereference** the pointer to read/write the value stored at that address.

Defining pointers

- Every data type has an associated pointer type:

```
int count; // integer

int *countPtr; // pointer to an integer
```

- Variable `countPtr` has type `int *`.

Defining pointers

- Every data type has an associated pointer type:

```
int count; // integer
int *countPtr; // pointer to an integer
```

- Variable `countPtr` has type `int *`.

- Be careful with multiple definitions on the same line:

```
int *countPtr, count;
```

Defining pointers

- Every data type has an associated pointer type:

```
int count; // integer
int *countPtr; // pointer to an integer
```

- Variable `countPtr` has type `int *`.

- Be careful with multiple definitions on the same line:

```
int *countPtr, count;
```

- Only countPtr is a pointer; count is still an integer.

Defining pointers

- Every data type has an associated pointer type:

```
int count; // integer
int *countPtr; // pointer to an integer
```

- Variable `countPtr` has type `int *`.

- Be careful with multiple definitions on the same line:

```
int *countPtr, count;
```

- Only countPtr is a pointer; count is still an integer.

```
int *countPtr, *sumPtr;
```

- Both countPtr and sumPtr are pointers.

Initializing pointers

- We make a pointer point to a variable's memory address using the **reference** operator &:

```
int count = 5; // integer

int *countPtr = &count; // pointer to an integer
```

- Variable `countPtr` now holds the memory address of count.

Dereferencing pointers

- To read from / write to the value in memory, we can use the **dereference** operator *:

```
int count = 5; // integer
int *countPtr = &count; // pointer to an integer
*countPtr += 5; // count now = 10
printf("%d\n", count); // outputs 10
printf("%d\n", (*countPtr) ); // outputs 10
printf("%lu\n", countPtr ); // outputs address
```

Initializing pointers

- It is good practice to initialize pointers.
  - Dereferencing an uninitialized pointer may or may not crash the program, leading to hard-to-detect bugs.

Initializing pointers

- It is good practice to initialize pointers.
  - Dereferencing an uninitialized pointer may or may not crash the program, leading to hard-to-detect bugs.
- The constant `NULL` is defined in `stddef.h` (and other standard headers) for this purpose.
  - `NULL` is equal to 0.

```
int *myPtr = NULL; // initialized
```

Initializing pointers

- It is good practice to initialize pointers.
  - Dereferencing an uninitialized pointer may or may not crash the program, leading to hard-to-detect bugs.
- The constant `NULL` is defined in `stddef.h` (and other standard headers) for this purpose.
  - `NULL` is equal to 0.

```
int *myPtr = NULL; // initialized
```

- Dereferencing a `NULL` pointer will *definitely* crash the program!

Inspecting variable addresses

```c
#include<stdio.h>

int main()
{
    int a = 5;
    int b = 10;
    int *aPtr = &a;
    int *bPtr = &b;
    printf("%lu\n",aPtr);
    printf("%lu\n",bPtr);
}
```

Inspecting variable addresses

```c
#include<stdio.h>

int main()
{
    int array[4];
    for ( int i = 0; i < 4; i++ ) {
        printf("%lu ",&array[i]);
    }
    puts("");
}
```

Reference/dereference operator precedence

- We have to be careful to make sure we are referencing / dereferencing the correct address.

```
int array[5];
int *ptr = &array[1];
```

- Is `&` applied to `array` or `array[1]`?

Reference/dereference operator precedence

- We have to be careful to make sure we are referencing / dereferencing the correct address.

```
int array[5];
int *ptr = &array[1];
```

- Is `&` applied to `array` or `array[1]`?

- The `&` is applied to `array[1]`, because `[]` has higher precedence than `&`.

Passing arguments by reference with pointers

- With pointers we can implement pass-by-reference:

```
int myFn( float *valuePtr );
```

- The function `myFn` can modify the value pointed to by `valuePtr` by dereferencing the pointer.

Pass-by-value example

```c
#include <stdio.h>

int cubeByValue( int n );

int main()
{
    int number = 5;

    printf("number is %d\n", number );

    number = cubeByValue( number );

    printf("number is %d\n", number );
}

int cubeByValue( int n )
{
    return n * n * n;
}
```

Pass-by-reference example

```c
#include <stdio.h>

void cubeByReference( int *nPtr );

int main()
{
    int number = 5;

    printf("number is %d\n", number );

    cubeByReference( &number );

    printf("number is %d\n", number );
}

void cubeByReference( int *nPtr )
{
    *nPtr = *nPtr * *nPtr * *nPtr;
}
```

Pass-by-reference example

```c
#include <stdio.h>

void cubeByReference( int *nPtr );

int main()
{
    int number = 5;

    printf("number is %d\n", number );

    cubeByReference( &number );

    printf("number is %d\n", number );
}
void cubeByReference( int *nPtr )
{
    // put pointer dereference in parentheses
    // to make code easier to read
    *nPtr = (*nPtr) * (*nPtr) * (*nPtr);
}
```

Pointers and arrays

- In the function header, a pointer and a one-dimensional array are *interchangeable*.

```
float sumArray( float array[] );
```

```
float sumArray( float *array );
```

- These two function headers are equivalent.

Pointers and strings

- It is possible to assign a string to a pointer:

```
char *stringPtr = "hello!";
char stringArray[] = "hello!";
```

- These are **almost** equivalent, but there is a difference to be discussed later.

const pointers

- Using the const keyword, we can restrict the code from modifying the data pointed to by the pointer.

```
int data = 10;
const int* dataPtr = &data;
int dataCopy = *dataPtr; // allowed
*dataPtr += 1; // not allowed
```

const pointers

- We can also use const to restrict the code from modifying the pointer itself.
  - In this case, the const comes *after* the *.

```
int data = 10;
int * const dataPtr = &data;
int dataCopy = *dataPtr; // allowed
*dataPtr += 1; // allowed
dataPtr += 1; // not allowed
```

const pointers

- Using both constructions, we get a pointer where neither the address nor the data can be changed.

```
int data = 10;
const int * const dataPtr = &data;
int dataCopy = *dataPtr; // allowed
*dataPtr += 1; // not allowed
dataPtr += 1; // not allowed
```

Applications of const pointers

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

void convertToUppercase( char *sPtr );

int main()
{
    char string[] = "Hello, World!";

    puts(string);
    convertToUppercase(string);
    puts(string);
}

void convertToUppercase( char *sPtr )
{
    while ( *sPtr != '\0' ) { // loop until null terminator
        *sPtr = toupper(*sPtr); // convert to uppercase
        ++sPtr; // increment pointer
    }
}
```

Applications of const pointers

```c
#include <stdio.h>
#include <ctype.h>
// printString should not change input data
void printString( const char *sPtr );

int main()
{
    char string[] = "Hello, World!";

    printString(string);
}

void printString( const char *sPtr )
{
    while ( *sPtr != '\0' ) { // loop until null terminator
        putchar( *sPtr ); // print character
        ++sPtr; // increment pointer -- will point to next character
    }
}
```

Applications of const pointers

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

void exponents( const float *input, float *square, float *cube );

int main()
{
    float value = 10.f; // .f indicates float
    float square;
    float cube;

    exponents( &value, &square, &cube );

    printf("%f %f %f\n",value, square, cube);
}

void exponents( const float *input, float *square, float *cube )
{
    *square = (*input) * (*input);
    *cube = (*input) * (*input) * (*input);
}
```

Pointer arithmetic operators

- A pointer is simply an integer:
  - A memory address
  - Number of bytes from the beginning of memory
- So, we can do integer arithmetic (add or subtract) on pointers.
- Allowed operations:
  - `pointer++`
  - `pointer--`
  - `pointer + integer`
  - `pointer - integer`
  - `pointer - pointer`

Pointing to an array

- Pointer arithmetic is useful when iterating over an array.
- First we need to make our pointer point to the array:

```
int array[5] = { 0, 1, 2, 3, 4 };
int *ptr = array;
// --- OR ---
int *ptr = &array[0];
```

Pointer arithmetic example

- Now we can iterate over the array by incrementing the pointer:

```
int array[5] = { 0, 1, 2, 3, 4 };
// initialize pointer to first element of array
int *ptr = array;
// increments pointer
// will now point to second element
ptr++;
// print second element (prints 1)
printf("%d\n", (*ptr) );
```

Pointer arithmetic example

- Now we can iterate over the array by incrementing the pointer:

```
int array[5] = { 0, 1, 2, 3, 4 };
for ( int *ptr = array; ptr != array+5; ptr++ )
{
    printf("%d\n", (*ptr) );
}
```

Pointer arithmetic example

- We can similarly use += to increment by larger step sizes:

```
int array[5] = { 0, 1, 2, 3, 4 };
for ( int *ptr = array; ptr != array+5; ptr+=2 )
{
    printf("%d\n", (*ptr) );
}
```

- What's the error here?

Pointer arithmetic example

- We can similarly use += to increment by larger step sizes:

```
int array[5] = { 0, 1, 2, 3, 4 };
for ( int *ptr = array; ptr < array+5; ptr+=2 )
{
    printf("%d\n", (*ptr) );
}
```

- Need to use < instead of !=.

Pointer arithmetic example

- Using decrement, we can iterate over the array backward:

```
int array[5] = { 0, 1, 2, 3, 4 };
for ( int *ptr = array+4; ptr >= array; ptr-- )
{
    printf("%d\n", (*ptr) );
}
```

Pointer arithmetic example

- Decrementing with a greater step size:

```
int array[5] = { 0, 1, 2, 3, 4 };
for ( int *ptr = array+4; ptr >= array; ptr -= 2 )
{
    printf("%d\n", (*ptr) );
}
```

Pointer arithmetic example

- With a while loop:

```
int array[5] = { 0, 1, 2, 3, 4 };
int *ptr = array;
while ( ptr < array+5 ) printf( "%d\n", (*ptr++) );
```

Reference/dereference operator precedence

Operators	Associativity
`()` `[]` `++` (*postfix*) `--` (*postfix*)	left to right
`+` `-` `++` (*prefix*) `--` (*prefix*) `!` `*` `&` (*type*)	right to left
`+` `/` `%`	left to right
`<` `<=` `>` `>=`	left to right
...	

Table: Operator precedence

Reference/dereference operator precedence

```
#include<stdio.h>

int main()
{
    int data[5] = { 10, 20, 30, 40, 50 };
    int *dataPtr = data;

    printf("%d ",*dataPtr++);
    printf("%d ",*dataPtr);
}
```

What does this print?

Reference/dereference operator precedence

```
#include<stdio.h>

int main()
{
    int data[5] = { 10, 20, 30, 40, 50 };
    int *dataPtr = data;

    printf("%d ",*dataPtr++);
    printf("%d ",*dataPtr);
}
```

Outputs 10 20.

The ++ is applied to the pointer after the first print.

Reference/dereference operator precedence

```
#include<stdio.h>

int main()
{
    int data[5] = { 10, 20, 30, 40, 50 };
    int *dataPtr = data;

    printf("%d ",*++dataPtr);
    printf("%d ",*dataPtr);
}
```

What does this print?

Reference/dereference operator precedence

```
#include<stdio.h>

int main()
{
    int data[5] = { 10, 20, 30, 40, 50 };
    int *dataPtr = data;

    printf("%d ",*++dataPtr);
    printf("%d ",*dataPtr);
}
```

Outputs 20 20.

The ++ is applied to the pointer before the first print.

Reference/dereference operator precedence

```
#include<stdio.h>

int main()
{
    int data[5] = { 10, 20, 30, 40, 50 };
    int *dataPtr = data;

    printf("%d ",++*dataPtr);
    printf("%d ",*dataPtr);
}
```

What does this print?

Reference/dereference operator precedence

```
#include<stdio.h>

int main()
{
    int data[5] = { 10, 20, 30, 40, 50 };
    int *dataPtr = data;

    printf("%d ",++*dataPtr);
    printf("%d ",*dataPtr);
}
```

Outputs 11 11.

The ++ is applied to the *data pointed to by the* pointer before the first print.

Reference/dereference operator precedence

```
#include<stdio.h>

int main()
{
    int data[5] = { 10, 20, 30, 40, 50 };
    int *dataPtr = data;

    printf("%d ",(*dataPtr)++);
    printf("%d ",*dataPtr);
}
```

What does this print?

Reference/dereference operator precedence

```
#include<stdio.h>

int main()
{
    int data[5] = { 10, 20, 30, 40, 50 };
    int *dataPtr = data;

    printf("%d ",(*dataPtr)++);
    printf("%d ",*dataPtr);
}
```

Outputs 10 11.

The ++ is applied to the data pointed to by the pointer after the first print.

Pointer arithmetic example

- We can subtract pointers:

```
int array[5] = { 0, 1, 2, 3, 4 };
int *ptr1 = array;
int *ptr2 = array+4;
int x = ptr2 - ptr1;
```

- What will `x` equal here?
- Why is this useful?

Pointer arithmetic example

- We can subtract pointers:

```
int array[5] = { 0, 1, 2, 3, 4 };
int *ptr1 = array;
int *ptr2 = array+4;
// x will equal 4
int x = ptr2 - ptr1;
```

- This tells us the number of elements from `ptr1` to `ptr2`.

Pointer arithmetic example

- We can subtract pointers:

```
int array[5] = { 0, 1, 2, 3, 4 };
int *ptr = array;
while ( (*ptr) < 3 ) ptr++;
int x = ptr - array;
```

- What will `x` equal here?

Pointer arithmetic example

- We can subtract pointers:

```
int array[5] = { 0, 1, 2, 3, 4 };
int *ptr = array;
while ( (*ptr) < 3 ) ptr++;
int x = ptr - array;
```

- x = 3
- Finds index of first value in array >= 3.

Comparing pointers

- As seen in the previous examples, we can *compare* pointers using >, <, etc.
- The comparison operator on pointers will compare *memory addresses*.

```
int array[5] = { 0, 1, 2, 3, 4 };
int *start = array;
int *end = array+5;
int *ptr = start;
while ( ptr < end ) printf("%d\n", (*ptr++) );
```

void * pointers

- A special type of pointer is a `void *` pointer.
- This pointer has no data type – it can point to any kind of data.

```
int array[5] = { 0, 1, 2, 3, 4 };
void *ptr = array;
```

void * pointers

- We are not allowed to dereference a `void *` pointer.
  - The compiler would not know what data type to produce.

```
int array[5] = { 0, 1, 2, 3, 4 };
void *ptr = array;
char c = (*ptr); // not allowed
```

void * pointers

- Instead, we need to *cast* the pointer to some data type:

```
int array[5] = { 0, 1, 2, 3, 4 };
void *ptr = array;
char *charPtr = (char*)ptr;
char c = (*charPtr); // allowed
```

- What does this code do?

void * pointers

- Instead, we need to *cast* the pointer to some data type:

```
int array[5] = { 0, 1, 2, 3, 4 };
void *ptr = array;
char *charPtr = (char*)ptr;
char c = (*charPtr); // allowed
```

- int is 4 bytes
- char is 1 byte
- char c will equal the first byte array[0].

sizeof

- `sizeof` tells us the size in bytes of an array (or any other data type).

```
int array[10];
size_t nbytes = sizeof(array);
```

- What will `nbytes` equal?
  - `size_t` is an `unsigned long int` (64-bit non-negative integer).

sizeof

- `sizeof` tells us the size in bytes of an array (or any other data type).

```
int array[10];
size_t nbytes = sizeof(array);
```

- `nbytes` is equal to 10×4 bytes $= 40$ bytes.

sizeof

- `sizeof` tells us the size in bytes of an array (or any other data type).

```
double array[10];
size_t nbytes = sizeof(array);
```

- What will `nbytes` equal?

sizeof

- `sizeof` can also be called on basic data types to find out their size – it varies according to platform and compiler.

```
size_t nbytes = sizeof(double); // equals to 8
```

Or we can call `sizeof` on a normal variable:

```
double a = 10.; // . means double
size_t nbytes = sizeof(a); // equals to 8
```

sizeof

- `sizeof` tells us the size in bytes of an array (or any other data type).

```
double array[10];
size_t nbytes = sizeof(array);
```

- A `double` is 64-bit (8 bytes).
- `nbytes` is equal to 10×8 bytes $= 80$ bytes.

sizeof

- There is an exception to the rule about `sizeof` on arrays.
- When an array is passed to a function, the receving function does not know the size of the array.
- What does `sizeof` return in this case?

sizeof

```c
#include <stdio.h>

void myfn( int array[] )
{
    printf("sizeof(array) in function: %lu\n",sizeof(array));
}

int main()
{
    int array[10];

    printf("sizeof(array) where it is defined: %lu\n",sizeof(array));

    myfn( array );
}
```

sizeof

```c
#include <stdio.h>

void myfn( int array[] )
{
    printf("sizeof(array) in function: %lu\n",sizeof(array));
}

int main()
{
    int array[10];

    printf("sizeof(array) where it is defined: %lu\n",sizeof(array));

    myfn( array );
}
```

In main, sizeof returns 40

In myfn, sizeof returns 8

Why?

sizeof

```c
#include <stdio.h>

void myfn( int *array )
{
    printf("sizeof(array) in function: %lu\n",sizeof(array));
}

int main()
{
    int array[10];

    printf("sizeof(array) where it is defined: %lu\n",sizeof(array));

    myfn( array );
}
```

`int array[]` is equivalent to `int *array`

In the function, `sizeof` returns the size of the pointer

Memory address is 64 bits = 8 bytes on this machine.

Arrays of pointers

- Sometimes it is useful to make an **array of pointers**:

```
const char *suit[4] = {
    "Hearts",
    "Diamonds",
    "Clubs",
    "Spades"
};
```

- The array contains four pointers.
- The pointers point to character arrays.

Arrays of pointers

- Sometimes it is useful to make an **array of pointers**:

```
const char *suit[4] = {
    "Hearts",
    "Diamonds",
    "Clubs",
    "Spades"
};
```

- Why is this preferred over creating a 2D array of characters?

Arrays of pointers

- Sometimes it is useful to make an **array of pointers**:

```
const char *suit[4] = {
    "Hearts",
    "Diamonds",
    "Clubs",
    "Spades"
};
```

- A 2D array would need to have a fixed size
- Would need to be as big as the longest string
- Would waste space.

Next Time

- Homework 6 due Wednesday, March 2, 11:55 PM.
- No class Thursday
- Next week: function pointers, midterm review, exam