COMP 1633: Intro to CS II

Pointers Continued

Charlotte Curtis

February 28, 2024

Where we left off

- Intro to pointers
- Assigning and dereferencing pointers

> *Textbook Chapter 9ish*

```
int x = 5;
int *ptr = &x;

*ptr = 10;
cout << "x: " << x << endl;
```

Today's topics

- Using pointers
- Pointers and `const`
- Pointers and structures
- Pointers and arrays

... you get the point

> *Textbook Chapter 9ish, still*

Pointers and `const`

- `const` makes a variable **read-only**

```
const int NUM_STUDENTS = 20;
NUM_STUDENTS = 21; // error! Can't change a const!
```

- What happens with `const` and pointers?

```
int x;
int y = 37;
const int *pci = &y; // pointer to a const int
int * const cpi = &y; // const pointer to an int
const int * const cpci = &y; // const pointer to a const int
```

Things are getting rather `const`-y

- A **pointer to a constant** is a pointer that points to a constant value
  - The value can't be changed **through the pointer**
  - The pointer can be changed to point to a different value
  - Whatever it points to becomes **read-only**
- A **constant pointer** is like a regular `const` variable
  - It must be initialized when declared, and can't be reassigned
  - However, the value at the address it points to can be changed
- A **constant pointer to a constant** combines the two

The same info in table form

Declaration	Can change value	Can change pointer
`int *ptr`	Yes	Yes
`const int *ptr`	No	Yes
`int * const ptr`	Yes	No
`const int *const ptr`	No	No

> *All this gets rather confusing*

Pointers and Arrays

- Recall that an array is a **contiguous block of memory**
- When arrays are declared and space is allocated, the **address of the first element** is associated with the name of the array, e.g.:

```
char A[10]
strcpy(A, "Hello");
```

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Value	H	e	l	l	o	\0	?	?	?	?

- So... if `A` is the **address** of the first element, can we assign it to a pointer?

Pointers and Arrays

```
char A[10] = "Hello";
char *cptr = A; // no &, because A is already an address
```

- `cptr` now points to the first element of `A`
- What if I modify `*cptr`?

  ```
  *cptr = 'J';
  ```

- What if I increment `cptr`?

  ```
  cptr++; // add one sizeof(type) to the address
  ```

> *Pointers can be used to iterate through arrays!*

Side tangent: Pointer arithmetic

- Array indexing using `[]` is "syntactic sugar" for pointer arithmetic
- Given the following declaration:

```
int arr[10];
```

- These operations are identical:

```
arr[0] = 5;
*arr = 5;
```

- Or, more generally, for an integral index `i`:

```
arr[i] = 5;
*(arr + i) = 5;
```

Pointers and Structures

- Recall that a **structure** is a collection of variables that could be different types

```
struct Time {
    int hour;
    int minute;
};
```

- You can access individual fields with `.` syntax:

```
Time t;
t.hour = 5; // it's 5 o'clock somewhere
```

- And just like anything else, you can declare a pointer to a structure:

```
Time *tptr = &t;
```

Pointers and Structures

- To access fields, you first need to **dereference** the pointer, then use `.`:

```
(*tptr).hour = 5;
```

- This is common enough and annoying enough that there's a shorthand:

```
tptr->hour = 5;
```

- This is called the **arrow operator** and is only used to access members of a structure or class via a pointer

```
tptr->hour++; // now it's 6 o'clock
cout << tptr->hour << ':' << tptr->min << endl;
```

Pointer check-in 1/2

Which of the following operators **can not** be used with pointers?

A. `&`
B. `*`
C. `++`
D. `[]`
E. `/`

Edit Feb 29: Originally this slide had a mistake and all the operators were valid!

Pointer check-in 2/2

Which statements are true about the following code snippet? Select all that apply.

A. `x` is a pointer to an `int`

B. `p` points to `x`

C. `p` could be used to change the value of `x`

D. `p` could be reassigned to point to `y`

E. The `const` has no effect

```
int x = 0;
int y = -1;
const int *p = &x;
cout << "x is " << x
     << " and y is " << y << endl;
```

Pointers and functions

- A pointer variable is like any other variable...
    - It can be passed to a function, by **value** or by **reference**
    - It can be returned from a function
- Things get funky passing pointers to functions:

```
void foo(int *iptr) {
    int x = 42;
    iptr = &x;
}

int main() {
    int *iptr = NULL;
    foo(iptr);
    cout << *iptr << endl; // what happens here?
}
```

Passing pointers by value

- Recall that when you pass a variable by value, a **copy** is made

```c
void foo(int *ptr); // Pointer is passed by value
```

- The function receives an **address** and assigns it to a **local pointer variable**
- The pointer can change the **value** at that address in the calling scope, but it can't change what the pointer points to

> *Kinda confusing, let's visualize*

Passing by reference

- If you want to change what the pointer points to, you need to pass it by reference:

```
void foo(int *&ptr); // Pointer is passed by reference
```

> *Read **right-to-left**:* `int *&ptr` *means "reference to pointer to* `int`*"*

- Caution: The new address **must exist** in the *calling* scope
- Remember that **local variables** disappear when the function returns

```
void do_stuff(int *ptr, int n) {
    ptr = &n;
}
```

Protecting what a pointer points to

- Passing a pointer by value still allows modifying the value at that address

```
void foo(int *ptr) {
    (*ptr)++;
}
```

- How do we protect against modifying values? `const`, of course!

```
void foo(const int *ptr);
```

- We've done this before with arrays:

```
void foo(const int arr[]);
```

... and in fact this is **exactly** the same thing!

What can be passed as a pointer?

Given the following function prototypes and variable declarations:

```
void foo(int *ptr);
void bar(int *&ptr);
```

```
int x = 0;
int *iptr = &x;
```

Which of the following are valid?

1. `foo(5);`
2. `foo(&5);`
3. `foo(&x);`
4. `foo(iptr);`
5. `foo(&iptr);`
6. `bar(5);`
7. `bar(&5);`
8. `bar(&x);`
9. `bar(iptr);`
10. `bar(&iptr);`

Side tangent: `typedef`

- `typedef` is a keyword that allows you to create **aliases** for types
- Syntax:

```
typedef <type> <alias>;
```

- Example:

```
typedef int * IntPtr;
IntPtr iptr = NULL;
void bar(IntPtr &ptr); // Can't mess up the order of & and * now
```

- This is recommended in our textbook, but it's a somewhat contentious practice - feel free to experiment and use what makes sense to you

Returning a pointer

- Just like any other variable, a pointer can be returned from a function
- But remember that local variables disappear when the function returns!
- The return value must point to something that **still exists** in the calling scope
- Example: Write a function with the following prototype that returns a **pointer** to the **largest element** in an array

```
int *max(int arr[], int n);
```

Dynamic allocation preview

- It's annoying that we need to guess how much memory we need at compile-time

```
char sentence[256]; // should be enough for a sentence, right?
```

- What if we want to allocate memory as needed?
- What if we want to allocate memory that **persists** after the function returns?
- Dynamic memory allocation to the rescue!

The heap and the stack

- There are two accessible areas of memory for a program:
    - The **stack** is used for local variables and function calls
    - The **heap** (or "freestore") is used for dynamic memory allocation

The `new` operator

To create a variable on the heap, use the `new` operator:

```
int *ptr; // memory for pointer is on the stack
ptr = new int; // what it points at is on the heap
```

This does the following:

1. Allocates enough memory on the heap for an `int`
2. Returns the address of the allocated memory

Some things to be cautious of:

- The allocated `int` can **only** be accessed through `ptr`!
- After you're done with it, you **must** `delete` it to free the memory

Coming up next

- Dynamic memory allocation
- **Midterm!** 🎉

> *Textbook Section 9.2*