COMP 1633: Intro to CS II

Pointers

Charlotte Curtis

February 26, 2024

Where we left off

- Reading and writing files
- Stream behaviour
- Command line arguments

> *Textbook Chapter 6, plus off-book*

```
#include <fstream>

int main(int argc, char *argv[]) {
    ofstream output(argv[1]);
    output << "Writing to a file!\n";
    output.close();
    return 0;
}
```

Today's topics

- A bit of midterm info
- Intro to pointers
- Assigning and dereferencing pointers

> *Textbook Chapter 9, kinda*

Midterm Info

- Topics up to and including **Pointers** (this week)
- Format: multiple choice, short answer, tracing, coding
- No cheat sheet, but I will provide the operator precedence table
- Expect coding questions similar to assignments 1 and 2, plus conceptional questions about memory allocation

That mysterious issue from last lecture

- In the `copy_and_meow` function, I had the following loop:

```
char line[256];
while (in.getline(line, 256)) {
     str_replace(line, "now", "meow");
     out << line << endl;
}
```

- Turns out that my text file had a line longer than 256 characters
- I RTFM and realized that while the first 255 chars were read into `line`, the **failbit** was set on `in` and the loop never executed
- Lesson: choose your buffer size wisely!

And now, pointers!

- Pointers are a **powerful** and **confusing** feature of C++
- They allow for dynamically sized arrays, linked data structures, and more
- It's also how pass-by-reference works in C++
- We've been using pointers already!
  - `void add_one(int arr[], int size)`
  - Passing an array to this function passes a **pointer** to the first element of the array - this is why putting a size in the `[]` doesn't matter

> *"It's easier to give someone your address than to make a copy of your house"*
>
> *-- Something I read somewhere, probably Stack Overflow*

Memory and Addresses

- Memory is a sequence of **bytes** (8 bits), the smallest addressable unit
- **Declaring** a variable allocates enough memory to store the value, and also allows us to reference the location by name
- The **address** of a variable is the location in memory where it is stored

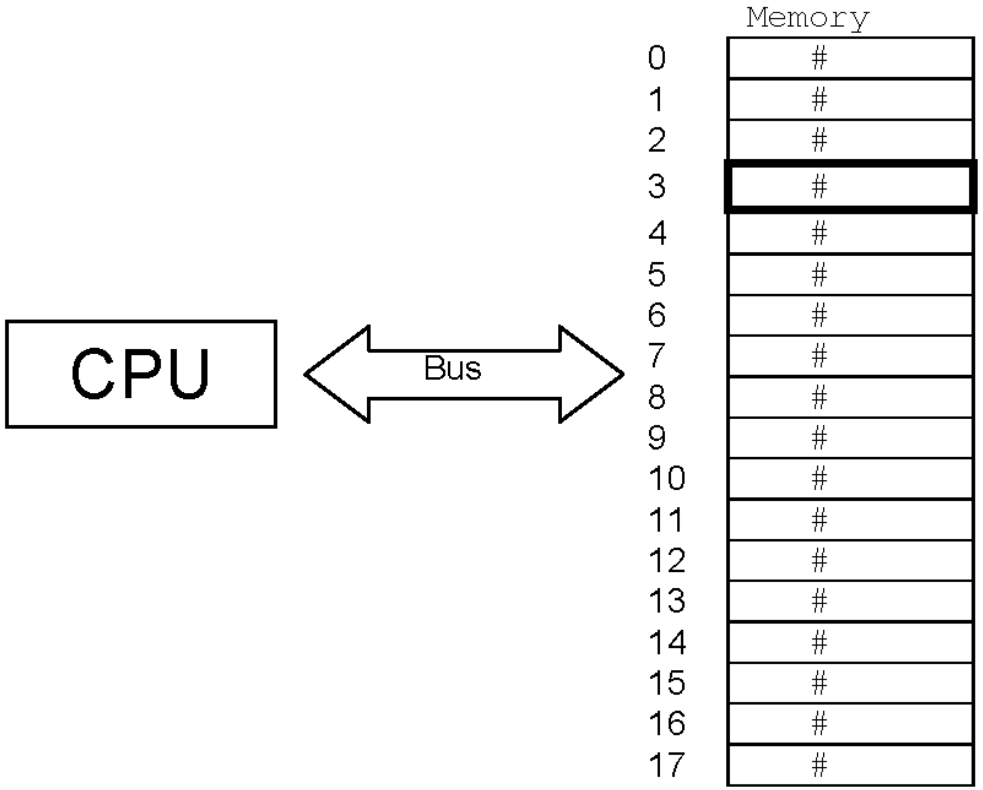


Image courtesy of Paul Pospisil

Allocating Memory: Example

```
int main() {
    int i = 42;
    int j;
    char c = 'K';
    double d = 3.14159;
    // ...
    return 0;
}
```

> *The memory addresses are **integers**, though usually hexadecimal (base 16)*

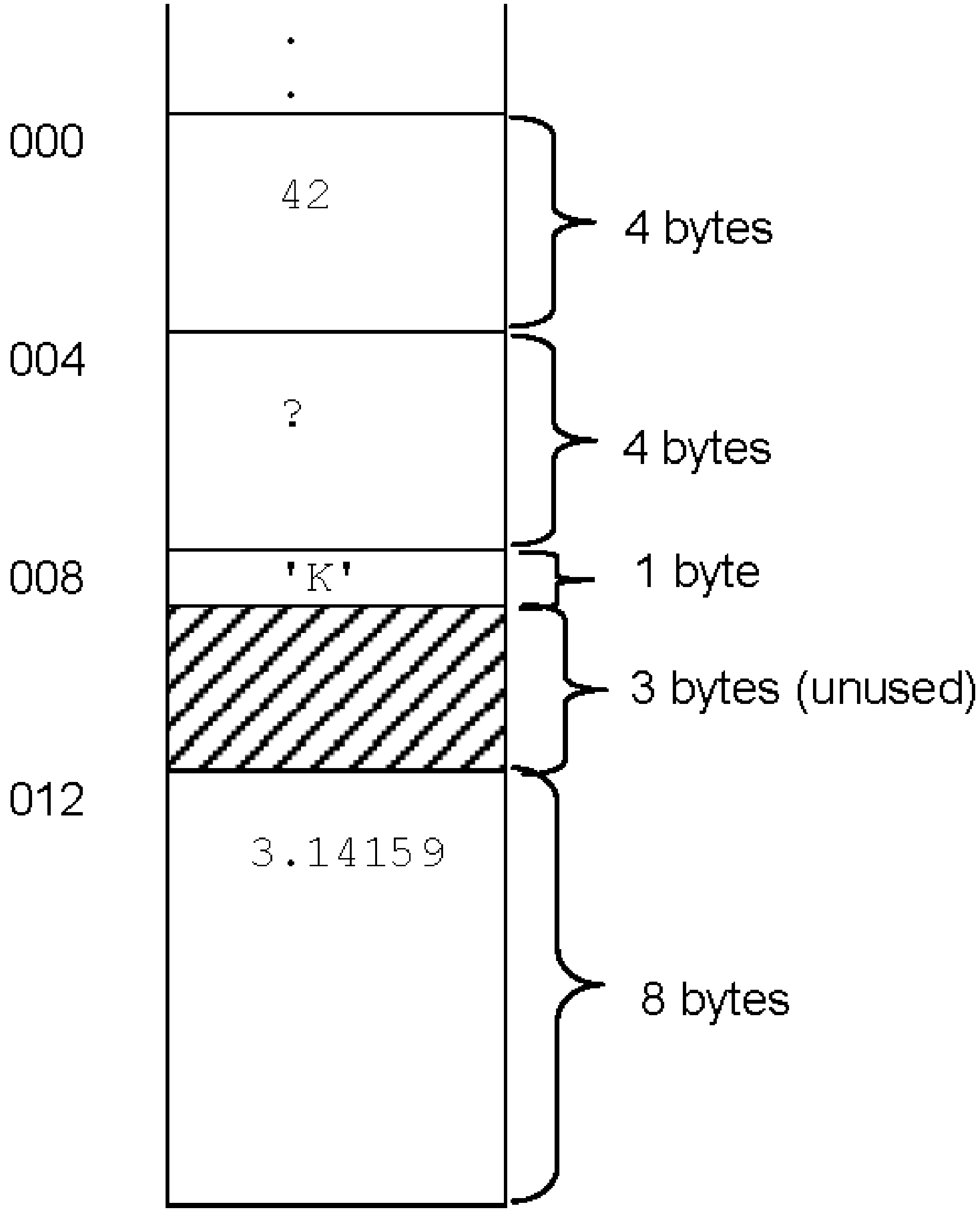
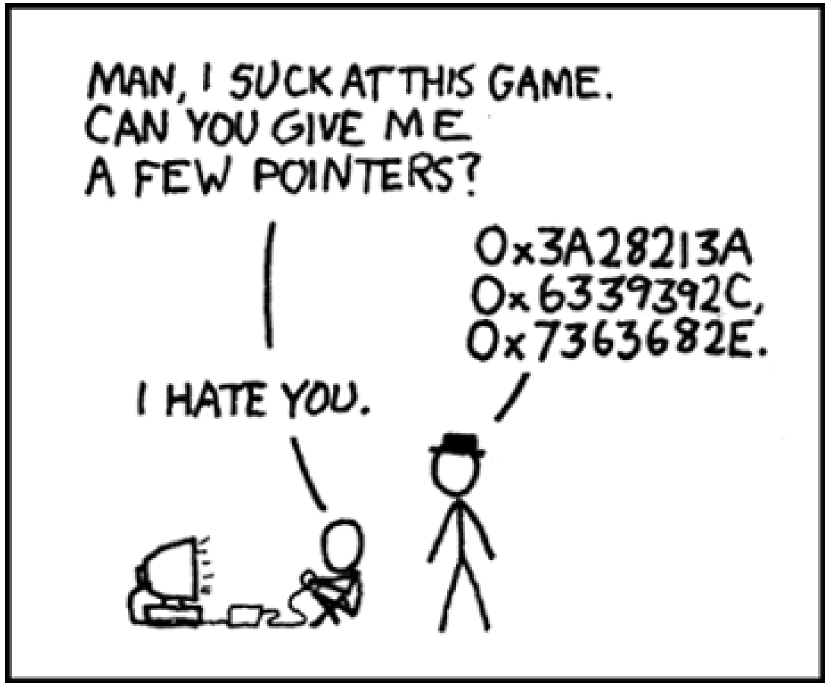


Image courtesy of Paul Pospisil

The pointer type

- A **pointer** is a variable that stores the **address** of another variable
- The **value** of a pointer doesn't make sense on its own
- Memory addresses are integers, but pointers are a specific type, such as:
    - Pointer to an `int`
    - Pointer to a `char`
    - Pointer to a `BillInfo` struct



https://xkcd.com/138

Declaring pointers

- Passing by reference uses `&`, but this is the **address-of** operator
    - `int &x = y;` is a compile time error
- The actual syntax:

  ```
  type *variable_name;
  ```

  where `type` is the type of the variable being pointed to
- Example:

  ```
  int x; // a normal integer variable
  int *p; // a pointer to an integer
  ```

The `*` is not part of the type!

While C++ allows the `*` to be placed anywhere between the type and the variable, you have to be very careful:

```cpp
char *cptr; // pointer to a char
char* cptr2; // also a pointer to a char

int *ptr1, *ptr2; // Two pointers to ints
int* ptr3, ptr4; // ptr3 is a pointer to an int, ptr4 is an int
```

- Keeping the `*` next to the variable name helps to keep things straight

What happens when we declare a pointer?

- Like other variable declarations:
  - Memory is allocated
  - The value is **uninitialized** (random garbage)
- Regardless of the type, memory allocated to a pointer is the size of an `int`
- The random garbage may or may not point to a valid memory address

> *When a program runs, it is given its own isolated memory space. While you might get segmentation faults by accessing invalid memory locations, you won't bork your system or break another program.*

Side tangent: Segmentation fault

- A **segmentation fault** or "segfault" might happen if you:
  - Try to access memory that doesn't belong to you
  - Try to write to read-only memory
  - Try to get the value of unallocated memory
- Basically, any time you mess with memory that isn't your own, you might see:

  ```
  Segmentation fault (core dumped)
  ```

- "Core dump" is a reference to old-school magnetic memory cores
- You can **backtrace** in `gdb` to find the offending code

Initializing pointers

- We can initialize pointers to point to a specific memory address:

```c
int *p = 0x7ffeeb6b4a4c;
```

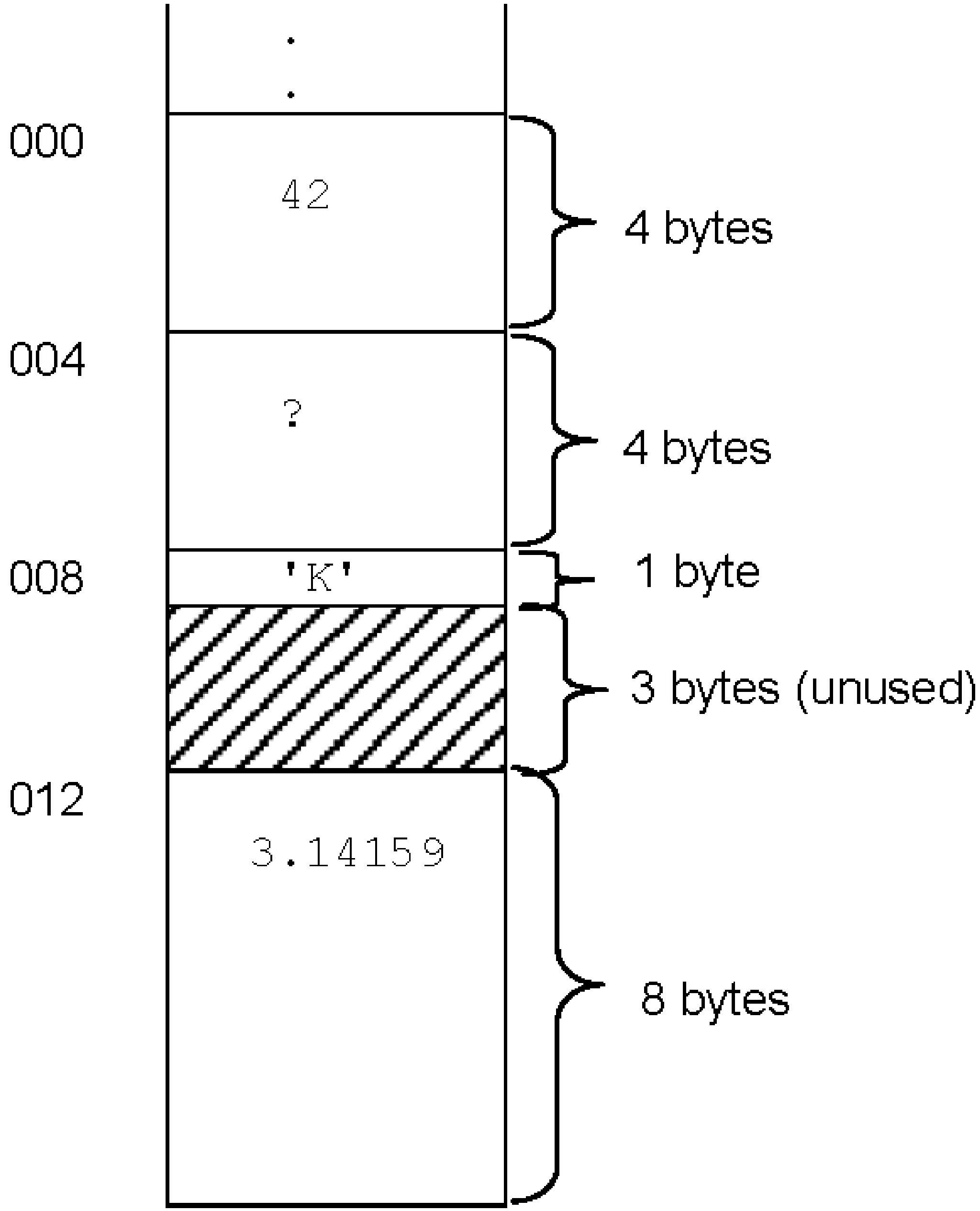
- But this is pretty much useless
- The only useful predefined pointer value is `NULL`, which is **falsy**

```c
int *iptr = NULL;
char *cptr = NULL;
```

> *In C++ 11, `NULL` was replaced with `nullptr` to avoid some ambiguity*

The `&` operator

- We've already seen the **address-of** operator, used for pass-by-reference
- Recalling the diagram to the right:
  - `&i` evaluates to `000`
  - `&j` evaluates to `004`
  - `&c` evaluates to `008`
  - `&d` evaluates to `012`
- We now have **valid addresses** that can be assigned to pointer variables!



Assigning addresses to pointers

Consider the following:

```
int i = 42;
int *iptr = &i;
iptr = &j;
```

What just happened??

> *Let's draw a diagram!*

Now add on:

```
int *iptr2;
iptr2 = iptr;
```

Dereferencing pointers

- Okay, we've declared and initialized pointers, but who cares?
- What we really want to manipulate is the **value** at that memory location
- The **dereference** operator `*` makes this happen

```
int i = 42;
int *iptr = &i;
cout << *iptr << endl; // prints 42
*iptr = 0;
cout << i << endl; // prints 0
```

- This is the same symbol used in the declaration, but it's a different operator!
- `&` gives the address, `*` gives the value - kind of like the inverse of each other

Some Pointer Gotchas

- Dereferencing `NULL` is **undefined behavior**
  - Good idea to check for `NULL` before dereferencing
  - `if (iptr) { ... }`
- Beware the precedence and associativity
  - Most operators are left-to-right, but `*` is right-to-left
  - This means that `*iptr++` is equivalent to `*(iptr++)`
  - This is **not** the same as `(*iptr)++`!
- `++` on a pointer is valid - it increments the **address** by the size of the type

Syntax Soup: exercise

Given the following, fill in the table to the right:

```c
int x = 24;
int *iptr = &x;
char c;
char *cptr = &c;
```

Expression	Type	Expression	Type
`x`		`c`	
`iptr`		`cptr`	
`&x`		`&c`	
`*iptr`		`*cptr`	

> *Take a few minutes to try to answer this (in groups or independently), then we'll go through the solution together*

Coming up next

- Lab tomorrow: pointers tutorial
- Lecture: pointers + arrays, functions, and structures
- **Assignment 2 due Friday, March 1**
- **Midterm: Wednesday, March 6** 🎉