CSCI 334:
Principles of Programming Languages

Lecture 3

Data types, values, and pointers

Instructor: Dan Barowy

Williams

HW1: Due tonight by 10pm

(assignment had a typo)

(come see me if this typo bit you)

HW1: Don't forget your `Makefile`

(it's worth 30 points)

Final Exam Study Guide

`git` Tutorial

`git` Tutorial

`git clone`

Retrieves repository from {Github, wherever}

`git` Tutorial

`git add <file>`

Adds a file (to your changelist).

`git` Tutorial

`git commit -m <message>`

Commits a changelist with a message.

`git` Tutorial

`git rm <file>`

Removes a file (from your changelist)

`git` Tutorial

`git status`

Displays the status of your changelist

`git` Tutorial

`git diff`

Displays the differences between your changelist and the last committed version

`git` Tutorial

`git push`

Uploads *committed* changes back to {Github, whatever}.

`git` Tutorial

`git pull`

Downloads *latest commits* to existing cloned repository.

`git` Tutorial

See reading on website for more info.

If you're having trouble, come to office hours / TA hours.

Buffered I/O

C Primitive Data Types

These are the "atoms" of all C programs.

All of these can be stored directly in a computer's memory

`int`	at least 2 bytes
`float`	#bytes not specified as long as IEEE 754
`double`	#bytes not specified as long as IEEE 754 double
`char`	smallest addressable unit that can contain ASCII

These may not have the representation that you expect!

May vary for different compiler, architecture, OS!

C Portable Integer Types

If you need "portable" data types, see `stdint.h`

`int8_t`	8-bit signed integer
`uint8_t`	8-bit unsigned integer
`int16_t`	16-bit signed integer
`uint16_t`	16-bit unsigned integer
`int32_t`	32-bit signed integer
`uint32_t`	32-bit unsigned integer
`int64_t`	64-bit signed integer
`uint64_t`	64-bit unsigned integer

Nice huh? Everybody knows signed/unsigned, right?

For this class, ordinary primitives are fine.

C Primitive Data Types

Byte widths are not the only portability concern!

(e.g., endianness)

Take CSCI 237 for more details.

(writing truly portable C is *difficult*!)

Type Checking

If you ask C for storage of a given type,
C *gently asks* that you be consistent.

```
int a;
a = 3.2;
```

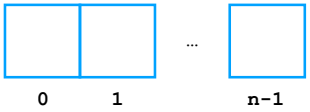
```
tc.c:3:7: warning: implicit conversion from 'double' to 'int' changes
value from 3.2 to 3 [-Wliteral-conversion]
  a = 3.2;
    ~ ^~~
1 warning generated.
```

C is a *weakly typed* language, unlike Java.

C may warn you (like above), but if you really want to do it, it will let you.

C Complex Data Types: Array

A sequence of values, stored contiguously

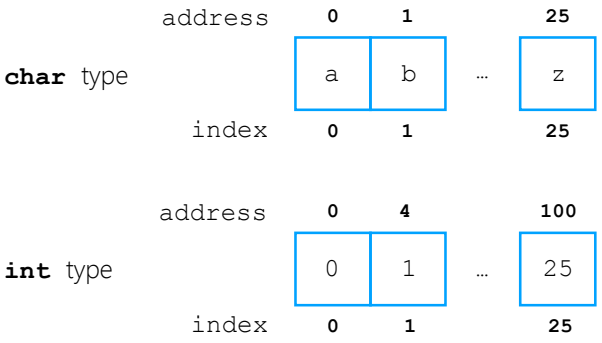


Any *type* of value can be used.

```
int arr[10];
int * arr[10];
struct point arr[10];
struct point * arr[10];
```

C Complex Data Types: Array

Amount of storage depends on *type* of value.



C Complex Data Types: Array

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

Reading:

`arr[3]` (returns 4th element)

Writing:

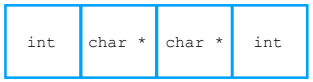
`arr[3] = 2;` (assigns to 4th element)

C Complex Data Types: Struct

A sequence of values, of heterogeneous type, stored contiguously

```
struct Account {
  int account_no;
  char *first_name;
  char *last_name;
  int balance;
};
```

"fields"



The actual storage layout varies wildly! Do not assume anything!

C Complex Data Types: Struct

```
struct Account my_account;
```

Reading:

`my_account.account_no` (returns `account_no` field)

Writing:

```
my_account.account_no = 12345678
```

(assigns to `account_no` field)

C Complex Data Types: Struct

Handy trick: `typedef`

syntax: `typedef <definition> <alias>;`

```
typedef struct Account {
  int account_no;
  char *first_name;
  char *last_name;
  int balance;
} Acc;

Acc my_account;
my_account.account_no = 12345678;
```

C Complex Data Types: Union

One value, stored in the same memory location

```
union never_do_this {
  int account_no;
  char *first_name;
  char *last_name;
  int balance;
};
```

Unions are used for special purposes.

We will never use them in this class.

You should avoid them.

C Complex Data Types Are Composable

Perfectly valid and acceptable C:

```
typedef struct Account {
  int account_no;
  char *first_name;
  char *last_name;
  int balance;
} Acc;

Acc arr[1000];
```

C Complex Data Types Are Composable

Perfectly valid and acceptable C:

```
typedef struct Account {
  int account_no;
  char *first_name;
  char *last_name;
  int balance;
  struct birthday {
    int year;
    int month;
    int day;
  }
} Acc;
```

Pointers

So simple they cause confusion.

A pointer is just an address.

```
int *ptr;
```

The type tells you the type of the value at that address.

```
int
```

Pointers

What address does `ptr` point to?

```
int *ptr;
```

Right now it points at nothing.

`ptr` is a variable, just like any other variable.

Pointers

There are two important pointer operations.

1. We can get a pointer to a value.

```
int i;
int *ptr;
ptr = &i;
```

What address does `ptr` point to?

`&` is the *address of* operator.

Pointers

There are two important pointer operations.

2. We can follow a pointer to a value.

```
int i;
int *ptr;
ptr = &i;
int j = *ptr;
```

What is `j`'s value?

`*` is the *dereference* operator.

Pointers

```
int i = 3;
int *ptr;
ptr = &i;
int j = *ptr;
```

What is `j`'s value now?

Storage Duration

This can be a tad complex.

We will focus on two: *automatic* (now) and *allocated* (next class)

Storage Duration: Automatic

```
int i = 3;
```

`i` has automatic duration, because you didn't specify anything.

C will automatically acquire (***allocate***)
and release (***deallocate***) memory for this variable.

In reality, nearly every C implementation will store `i` ***on the call stack***.

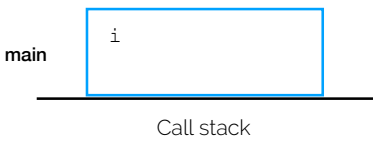
Storage Duration: Automatic

```
#include <stdio.h>

int main() {
  int i = 3;
  return i;
}
```

Call stack

Storage Duration: Automatic

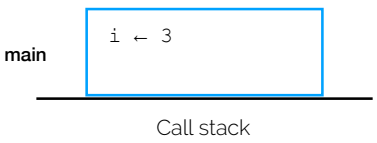


Call stack

```
#include <stdio.h>

int main() {
  int i = 3;
  return i;
}
```

Storage Duration: Automatic



Call stack

```
#include <stdio.h>

int main() {
  int i = 3;
  return i;
}
```

Storage Duration: Automatic

Call stack

```
#include <stdio.h>

int main() {
  int i = 3;
  return i;
}
```

Where does `i` get returned? How?

Storage Duration: Automatic

Call stack

```
#include <stdio.h>

int main() {
  int i = 3;
  return i;
}
```

`main`'s stack frame and all variables in it (i.e., `i`) are automatically deallocated when `main` ***goes out of scope***.

Activity

```
#include <stdio.h>

int add(int x, int y) {
  int z = x + y;
  return z;
}

int main() {
  int x = 1;
  int z = add(x, 3);
  return z;
}
```

2 →

1 →

3 →

Diagram the stack and variables when the program is at the three points.