Week 4 - Monday

CS222

Last time

- What did we talk about last time?
- Selection
- Loops
- Lab 3

Questions?

Project 2

Quotes

Unix was not designed to stop its users from doing stupid things, as that would also stop them from doing clever things.

Doug Gwyn

Bad Things

break

- The **break** command is a necessary part of the functioning of a switch statement
- But, it can also be used to jump out of a loop
- Whenever possible (i.e. always), it should not be used to jump out of a loop
  - Everyone once in a while, it can make things a little clearer, but usually not
  - Loops should have one entry point and one exit

```
for(value = 3; value < 1000; value += 2) {
        …
        if( !isPrime(value) )
                break;
}
```

```
for(value = 3; value < 1000 && isPrime(value);
    value += 2)
{
        …
}
```

continue

- The **`continue`** command is similar to the **`break`** command
- It will cause execution to jump to the bottom of the loop
- If it is a **`for`** loop, it will execute the increment
- For all loops, it will return to the top if the condition is true
- It makes things easier for the programmer up front, but the code becomes harder to follow
- The effect can be simulated with careful use of **`if`** statements

goto (a four letter word)

- A **goto** command jumps immediately to the named label
- Unlike break and continue, it is not a legal command in Java
- Except in cases of extreme (**EXTREME**) performance tuning, it should never be used
  - Spaghetti code results

```
for(value = 3; value < 1000; value += 2) {
     if( !isPrime(value) )
          goto stop;
}
printf("Loop exited normally.\n");
stop:
printf("Program is done.\n");
```

Systems Programming

System calls

- A **system call** is a way to ask the kernel to do something
- Since a lot of interesting things can only be done by the kernel, system calls must be provided to programmers via an API
- When making a system call, the processor changes from user mode to kernel mode
- There is a fixed number of system calls defined for a given system

glibc

- The most common implementation of the Standard C Library is the GNU C Library or **glibc**
- Some of the functions in the **glibc** perform systems calls and some do not
- There are slight differences between the versions of the **glibc**
  - Microsoft also has an implementation of the Standard C Library that doesn't always behave the same

Handling system errors

- There are no exceptions in C
- Instead, when a system call fails, it usually returns **`-1`**
- To find out why the system call failed
  - First, make sure you **`#include <errno.h>`**
  - Then check the value of the integer **`errno`** in your program after the system call fails
  - Use the man pages to determine what a given value of **`errno`** means
- The **`perror()`** function is often used to print errors instead of **`printf()`**
  - It sends the output to **`stderr`** instead of **`stdout`**

Error handling example

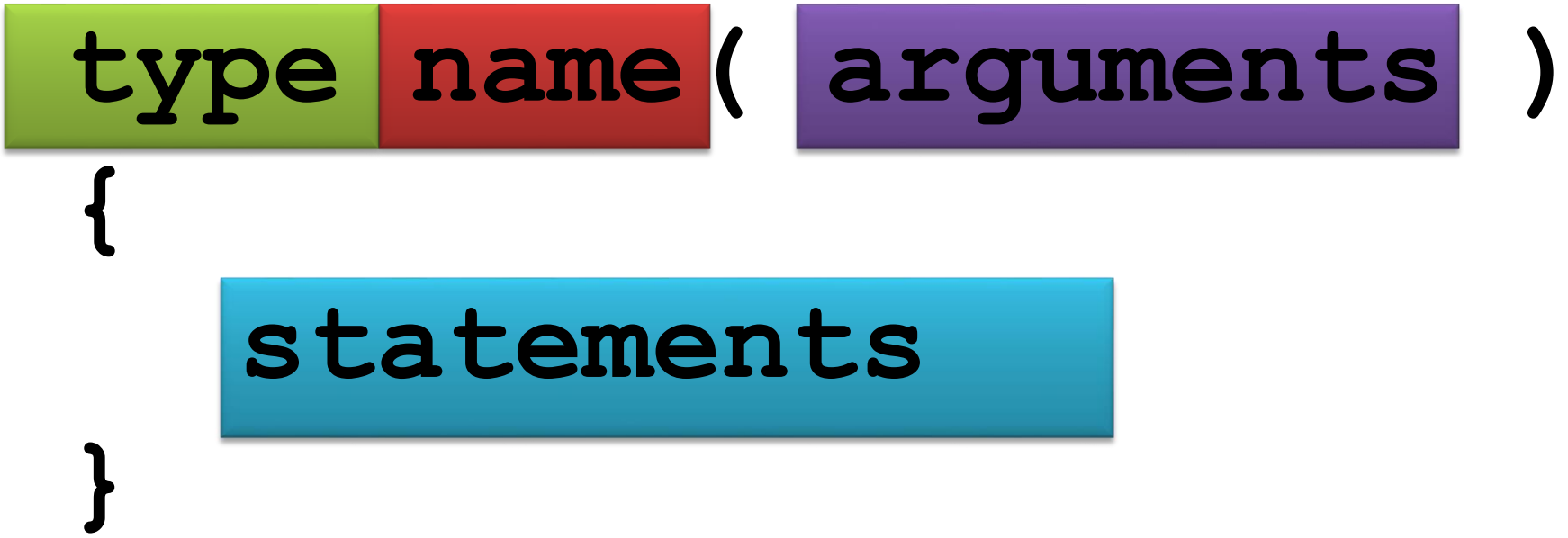
```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
int main(){
	int fd = open("eggplant.txt", O_WRONLY | O_CREAT | O_EXCL);
	if (fd == -1) {
		perror("Failure to create file: ");
		if( errno == EACCES )
			perror("Insufficient privileges\n");
		else if( errno == EEXIST )
			perror("File already exists\n");
		else
			perror("Unknown error\n");
		exit(EXIT_FAILURE);
	}
	return 0;
}
```

System types

- C has a feature called **`typedef`** which allows a user to give a new name to a type
- System types are often created so that code is portable across different systems
- The most common example is **`size_t`**, which is the type that specifies length
  - It's usually the same as **`unsigned int`**
- There are named types for process IDs (**`pid_t`**), group IDs (**`gid_t`**), user IDs (**`uid_t`**), time (**`time_t`**), and many others

Functions

Anatomy of a function definition



Differences from Java methods

- You don't have to specify a return type
  - But you **should**
  - **`int`** will be assumed if you don't
- If you start calling a function before it has been defined, it will assume it has return type **`int`** and won't bother checking its parameters

Prototypes

- Because the C language is older, its compiler processes source code in a simpler way
- It does no reasonable typechecking if a function is called before it is defined
- To have appropriate typechecking for functions, create a **prototype** for it
- Prototypes are like declarations for functions
  - They usually come in a block at the top of your source file

Prototype example

- Parameter names in the prototype are optional (and don't have to match)
- Both of the following work:

```
int root(int);
int root(int blah);
```

- You can also declare a prototype locally (inside a function), but there isn't a good reason to do so

```
#include <stdio.h>

int root(int value);
//integer square root

int main() {
   int output = root(19);
   printf("Value: %d\n", output);
   return 0;
}

int root(int value) {
      int i = 0;
      while( i*i <= value )
            i++;
      return i - 1;
}
```

Insanity

- If your method takes nothing, you should put **`void`** in the argument list of the prototype
- Otherwise, type checking is turned off for the arguments

```
double stuff();

int main()
{
   double output =
   stuff(6.4, "bang"); //legal
   return 0;
}
```

```
double stuff(void);

int main()
{
   double output =
   stuff(6.4, "bang"); //error
   return 0;
}
```

Return values

- C does not force you to return a value in all cases
  - The compiler may warn you, but it isn't an error
- Your function can "fall off the end"
- Sometimes it works, other times you get garbage

```
int sum(int a, int b)
{
      int result = a + b;
      return result;
}
```

```
int sum(int a, int b)
{
      int result = a + b;
}
```

Programming practice

- Let's write a function that:
  - Takes an unsigned integer as a parameter
  - Returns the location of the highest 1 bit in the integer (0-31) or -1 if the integer is 0

Parameter	Return Value
0	-1
2	1
3	1
2000	10
4294967295	31

More programming practice

- Let's update the function from the lab so that it can read negative integers too

```
int readInt()
{
	int c = 0;
	int i = 0;
	while( (c = getchar()) != EOF && c != '\n' )
	{
		if( c >= '0' && c <= '9')
			i = i * 10 + (c - '0');
	}
	return i;
}
```

Upcoming

Next time…

- More on functions
- Variable scope

Reminders

- Keep reading K&R chapter 4
- Start working on Project 2
  - **Get your teams sorted out by the end of the day!**