DYNAMIC MEMORY & C-STYLE STRINGS & STRUCTURES

Static Memory

- So far we've been dealing with **static memory** - variables allocated statically, at compile time.
- Static memory is declared *on the stack*
- Static memory is very easy for the compiler to deal with:
    - amount of memory fixed at compile time
    - no chance of memory leaks
- Downside(s) of static memory?

Dynamic Memory

- **Dynamic memory** is more powerful - you don’t need to know the size until runtime
- Can be used as necessary
- Dynamic memory comes from *the heap* - a pool of memory set aside for this purpose
- Downside(s) of dynamic memory?

Dynamic Allocation

- Memory is dynamically allocated through...
- ***POINTERS!!!!!!!!*** (woo!)

introducing the **new** keyword:

```
int* foo = new int;
```

- This syntax allocates a *single int*. You can also do this for arrays:

```
int* baz = new int[50];
```

Yet Another Review:

```
int* foo = new int;
```

foo is a dynamically allocated integer.
How do we use it?

```
int* baz = new int[50];
```

baz is a dynamically allocated *array* of integers. How do we use it?

How are these two things different?

dynamic arrays

- Arrays allocated via dynamic memory are used *exactly* the same way that arrays allocated statically are.
- Only one minor difference regarding the array pointer variable - anybody remember what it is?

Some Questions

- When does the life of a *statically* allocated variable end?
- When does the life of a *dynamically* allocated variable end?

```
for( int i = 0; i < 10; i++ )
{
    int* array = new int[15];
    ...
}
```

Cleaning Up

- See the problem with the above code?
- Static variables get de-allocated right when they go out of scope - dynamic variables *need to be deleted explicitly!*
- Otherwise you get memory leaks

Memory Leaks

- When you use a pointer to dynamically allocate memory...
- ... and the pointer goes out of scope before you have *deallocated* the memory...
- Then you have a memory leak.
- These are (usually) cleaned up by the operating system after the program exits, but the program can still run out of memory while it is running

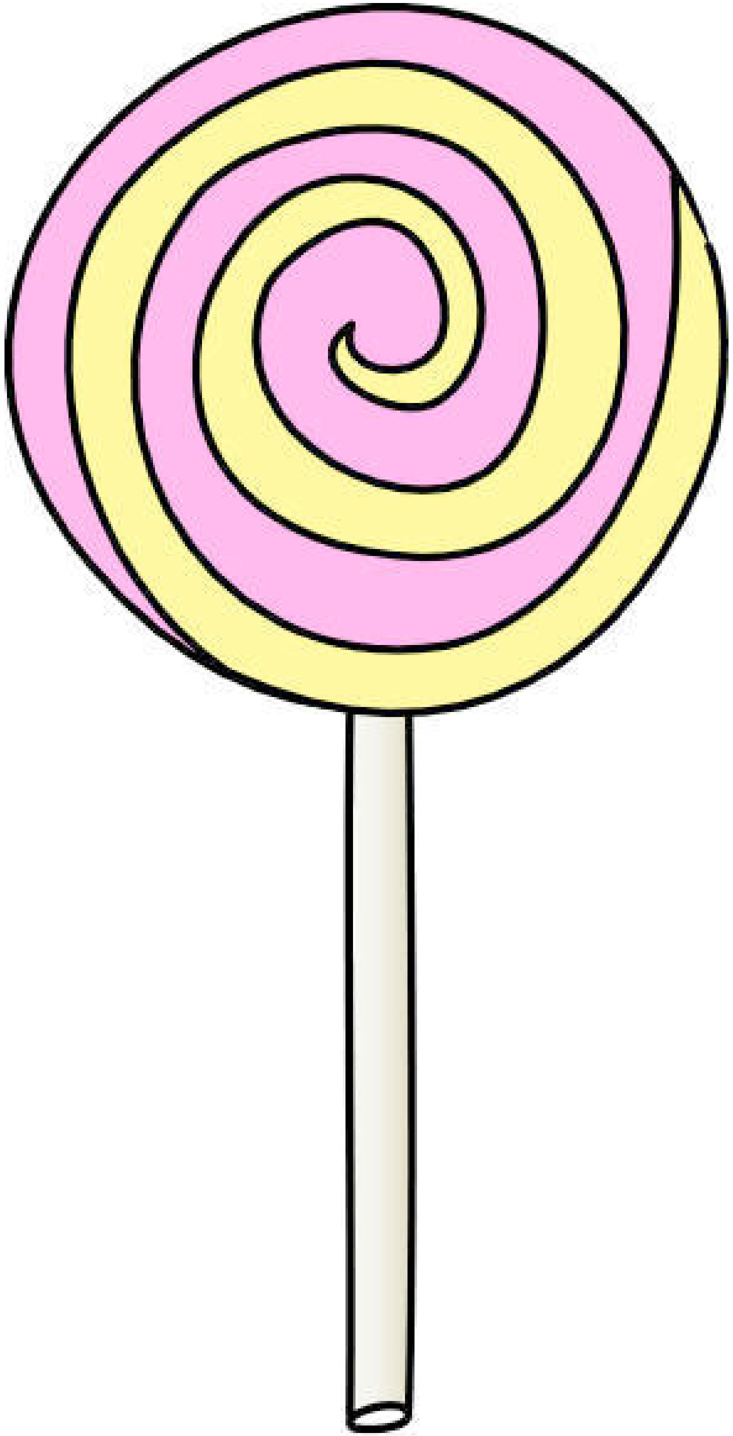
Cleaning Up

- *Single objects*, allocated with **`new`**, get cleaned up with the keyword **`delete`**:

```
int* foo = new int;
...
delete foo;
```

- *Arrays*, allocated with **`new`** and **`[]`**, get cleaned up with the keyword **`delete[]`**:

```
int* baz = new int[10];
...
delete[] baz;
```

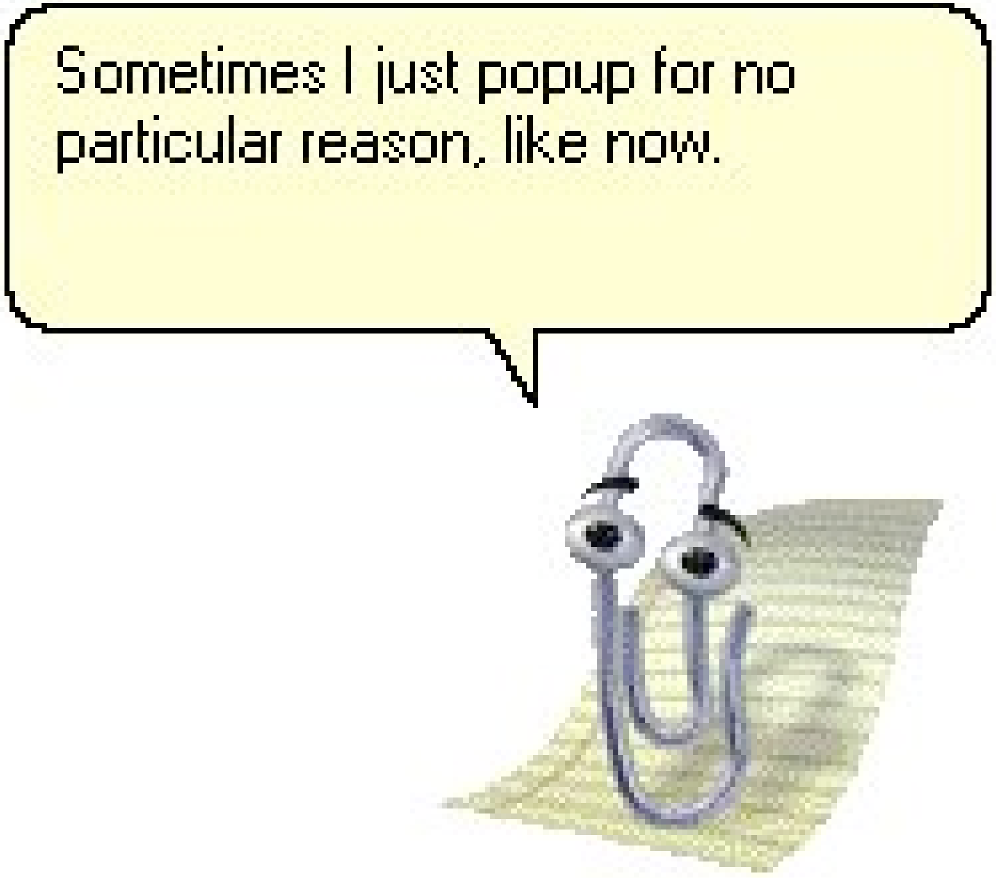


Fun with `delete`!

- What happens if we try and **`delete`** an *array* of dynamically allocated stuff?
- What if we try and **`delete`** a pointer that has been assigned the address of a static variable?
- What if we try to **`delete[]`** a pointer that has been allocated with a single **`new`**?

Useless Program Time!

Let's write a program that gets a number from the user, dynamically an array of that size, fills it with n powers of two, and prints 'em all out.



Hey! Wouldn’t it be nice if...

you could do stuff like this?

```
string word = “pickles”;
word += “ are tasty!”;
cout << word << endl;
```

You ***sure can***! Just... not today.

Today we’re learning about C-style strings,
which are quite a bit harder to use
and more annoying! Hooray!

C Strings

- It’s important to know these - you’ll come across them a lot, even when using C++
- A string in C is nothing special - just an array of `char`’s; each `char` holds a single character
- Messing with strings involves lots of nifty pointer arithmetic and manipulation

About Chars

- A character in C++ is a number (an 8-byte integer, to be exact)
- The numbers are coded using a standard mapping called ASCII: (American Standard Code for Information Interchange)
  - ‘a’ = 97, ‘b’ = 98, ‘A’ = 65, etc.
- You can find a table of these in about a gazillion places on the web

- You can assign single ASCII character values to a char using single quotes:

```
char letter = ‘A’;
cout << letter;
```

- Or you can assign a char an integer value (since it is an integer type):

```
char letter = 65;
cout << letter;
```

- You can also do arithmetic on characters:

```
char letter = ‘a’ + 2;
cout << letter;
```

Arrays of Chars

- Since a string is a sequence of characters, we can represent it as an array of characters:

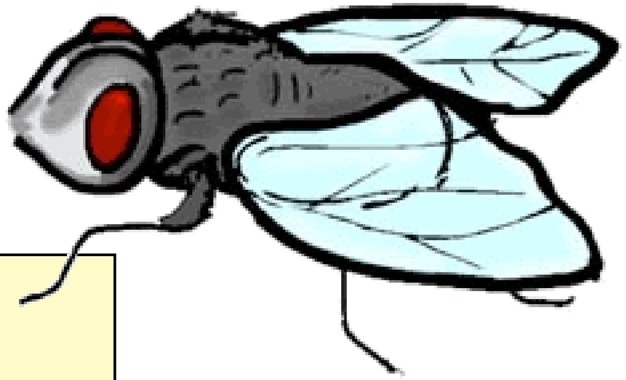
```
char turkleton[12];
```

- This array can hold up to 12 characters, as you'd expect
- This brings up the old array problem, though: how can you tell how big an array is?

null TERMINATORS

- Other than storing a separate counter variable, there’s no easy way to tell how many characters are in a string.
- The C solution to this is to have the last ‘character’ be a special character called a ***null terminator***, which has the value 0 - after this the string is considered “ended”, even if there is more following.
- There needs to be space to store the null terminator too, so each character array needs to have at least one more slot than you have characters.

C Strings



```
char turkleton[12];
```

- turk has room for only 11 actual characters, and one null terminator:

c	h	r	i	s	t	o	p	h	e	r	\0

length: 11

- Even though 11 characters will fit, you don't *need* 11 characters. Less is fine:

c	h	r	i	s	\0						

length: 5

Declaring Strings (Character Arrays)

- Because a C-style string is just a character array, we can declare it like any other array:

```
char kelso[7];
```

- If you want to pre-initialize it with numbers, that's OK too: a `char` is an integer, after all!

```
char kelso[7] = {1,2,3,4,5,6,7};
```

- More useful, though, to be able to fill it with characters...

```
char kelso[7] = {‘d’,’o’,’c’,’t’,’o’,’r’,’\0’};
```

Declaring Strings (Character Arrays)

- A shortcut in C/C++ is to use double-quotes in the initialization, instead of having to specify each character individually:

```
char kelso[7] = “doctor”;
```

- Note that we aren’t specifying the null terminator here: any string literal in C/C++ has the null terminator automatically appended.
- (A string literal means: any time you see stuff in double quotes in your source code file)

More Null Terminator Stuff

- The value of the null terminator is zero. Note that we specify it using a backslash-zero: ‘\0’
- You can embed this inside a string, too:

```
char CSBuilding[] = “MacLean\0 Hall”;
cout << CSBuilding << endl;
```

- Even though there’s more characters following “MacLean”, once a function encounters the null terminator it will stop printing

A Quick Detour:

Fun with Escape Sequences!

- Notice that we had to use '\0', instead of just '0'? Why is that?
- The backslash (\) tells the compiler that this is the start of an **escape sequence**: it means that the character following the backslash has a special meaning
- So '\0' means "null terminator", whereas '0' just means 'zero'
  - Not the *integer* zero, mind you: it means the character zero, which is actually the integer 48!

A Quick Detour:

Fun with Escape Sequences!

Some common escape sequences:

\0 → null terminator

\n → newline (like endl)

\’ → single quote

\” → double quote

\\ → an actual backslash

What does this mean?

It means that sometimes what you see isn’t what you get, and that you have to be careful with backslashes!

A Quick Detour:

Fun with Escape Sequences!

Here’s an actual chunk of (C) code that someone might write.
What’s wrong with this?

```
FILE* file = fopen(“C:\nichols\test.txt”,”r”)
```

We want these particular backslashes to be interpreted as *actual* backslashes, not escape sequences, so do it like this:

```
FILE* file = fopen(“C:\\nichols\\test.txt”,”r”)
```

On the other hand, escape sequences (newlines in particular) are often very handy, so feel free to use them:

```
cout << “I am very tired.\nI will go to sleep now.\n”;
```

Declarations:

Review / Check Yer Understanding

Which of these are valid and/or proper?

```
char bob[] = 1;
char bob[] = {1};
char bob[] = {'1', '\0'};
char bob[] = {'1', 0};
char bob[] = "hello";
char bob[] = {'h','e','l','l','o'};
char bob[30] = {'h','e','l','l','o','\0'};
char bob[3] = {'h','e','l','l','o','\0'};
```

Remember, we can also create strings dynamically:

```
char* bob = new char[50];
```

Note about Declarations

Stuff like this is nice and handy, but you **only** get to assign a string (or a group of numbers/characters) to an array when you're declaring it.

```
char janitor[20] = “fearitude”;
```

This doesn't work: (why not?)

```
char janitor[20];
janitor = “fearitude”;
```

String Functions

- We've been using `<iostream>` for a while now, but there are other libraries: a handy one for string functions is `<cstring>` or `<string.h>`
- Remember: this will include a header file, made up of function prototypes, but not the functions themselves: those get linked in later
- `<cstring>` gets you access to the old-school string functions in the C Standard Library
- ... it's important to know how these work, and what they're doing behind the scenes!

example: `strcpy`

This is a function that copies one string into another.

Here’s the prototype:

```
char *strcpy( char *dest, const char *src );
```

Here’s a sample usage:

```
char buffer[100];
strcpy( buffer, “Hi, I’m a string!” );
```

Anything bother you about this?

A Quick Detour:

Fun with Computer Security!

- When you put something into a string or array or *any* sort of data buffer, **C/C++ does not check to make sure that the data "fits"**.
- ***You*** are responsible for doing that.
- If you're not careful, `strcpy` and friends can be dangerous to use, because it will happily write past the end of the string, clobbering whatever happens to be in that memory.
- This isn't just bad programming; it can also be used to compromise your machine.

A Quick Detour:

Fun with Computer Security!

- So the moral of the story:
  - When you’re coding your own functions, make **sure** that you include code to prevent any overwriting of the buffer. (How would you do this?)
  - Use “safe” C functions (strncpy, etc) when you can instead of the “dangerous” ones (like strcpy, wgets, etc)

Anyway... string functions.

Here's the prototype of `strlen`, a function that calculates the length of a string:

```
int strlen( const char *s );
```

- strlen works by counting each character in a string until it hits a null terminator (which is not included in the count). It's a pretty simple function.
- Let's try writing our own version of strlen!

More Programming!

to tie a lot of this stuff together...

- Let's write a function kinda like strcpy, in that it copies a source buffer to a destination buffer, which we will create dynamically.
- It will include a maximum number of characters to copy (does this prevent overflow?)
- It will *only* copy characters that are either letters or a space.
- It will use lots of pointers! Hooray!

```
char* gcopy( char *dest, int maxCharsToCopy );
```

Comparing Strings

```
char one[10], two[10];

strcpy( one, "hello" );
strcpy( two, "hello" );

if( other == name )
   cout << "same";
else
   cout << "different";
```

- Say we need to compare two strings...
- Can we do it this way?
- Would <, >, <=, or >= work any better?

Comparing Strings

- The usual way to compare strings is *lexicographically* - think phone book/dictionary
- One function to do this is **strcmp**:

```
int strcmp( const char* s1, const char* s2 )
```

strcmp returns an integer that is:

< 0 when s1 < s2
0 when s1 == s2
> 0 when s1 > s2

For more information...

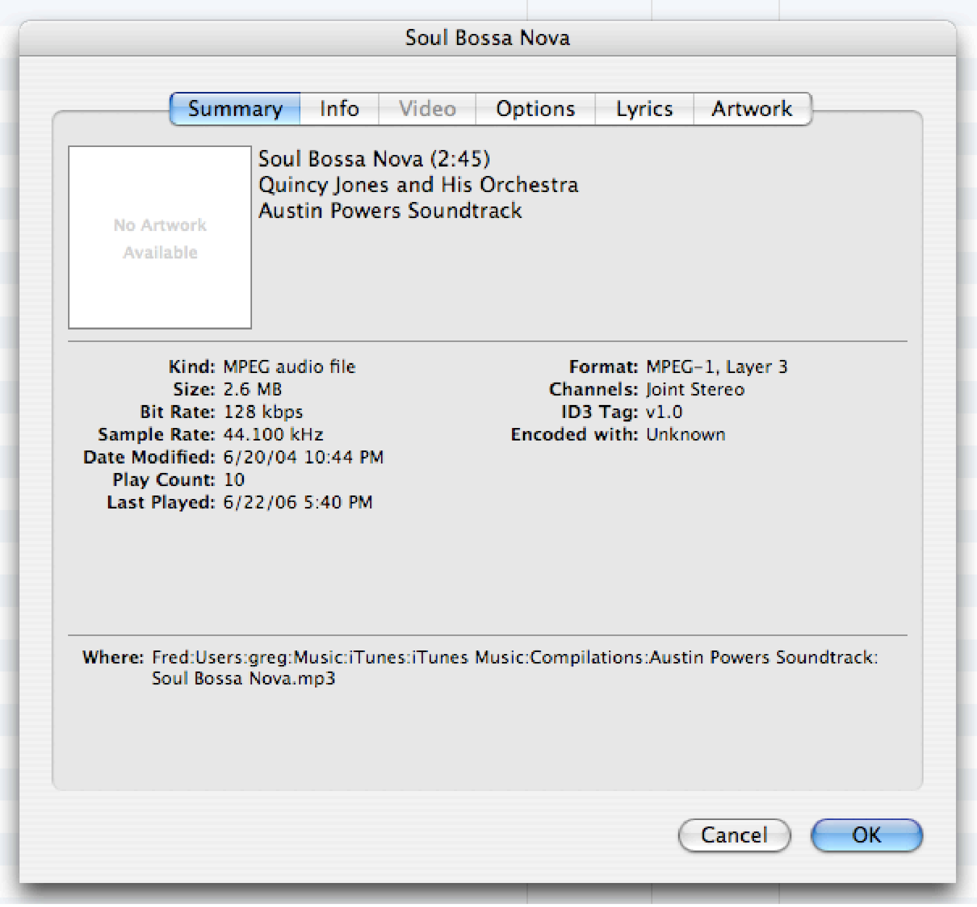
- The C standard library has many functions for working with strings:
  - formatting/modifying them
  - copying/manipulating them
  - converting them back and forth from integers, floats, etc.
  - ... and so on
- Google "string.h" and read about these if and when you need them!

So Far, We Can:

- Declare and use simple data types (int, float, char, bool, etc.)
- Use those data types in arrays
- This isn't enough, though: most complicated programs require *groups* of information, all neatly stored together

Motivation...

- Example: MP3 ID3 tags
- We might want to store name, bit rate, year, length, artist, album, etc.
- We've learned no convenient way of doing this, short of maybe declaring a variable for each item.
- This quickly becomes unworkable



Soul Bossa Nova

Soul Bossa Nova (2:45)
Quincy Jones and His Orchestra
Austin Powers Soundtrack

Kind: MPEG audio file
Size: 2.6 MB
Bit Rate: 128 kbps
Sample Rate: 44.100 kHz
Date Modified: 6/20/04 10:44 PM
Play Count: 10
Last Played: 6/22/06 5:40 PM

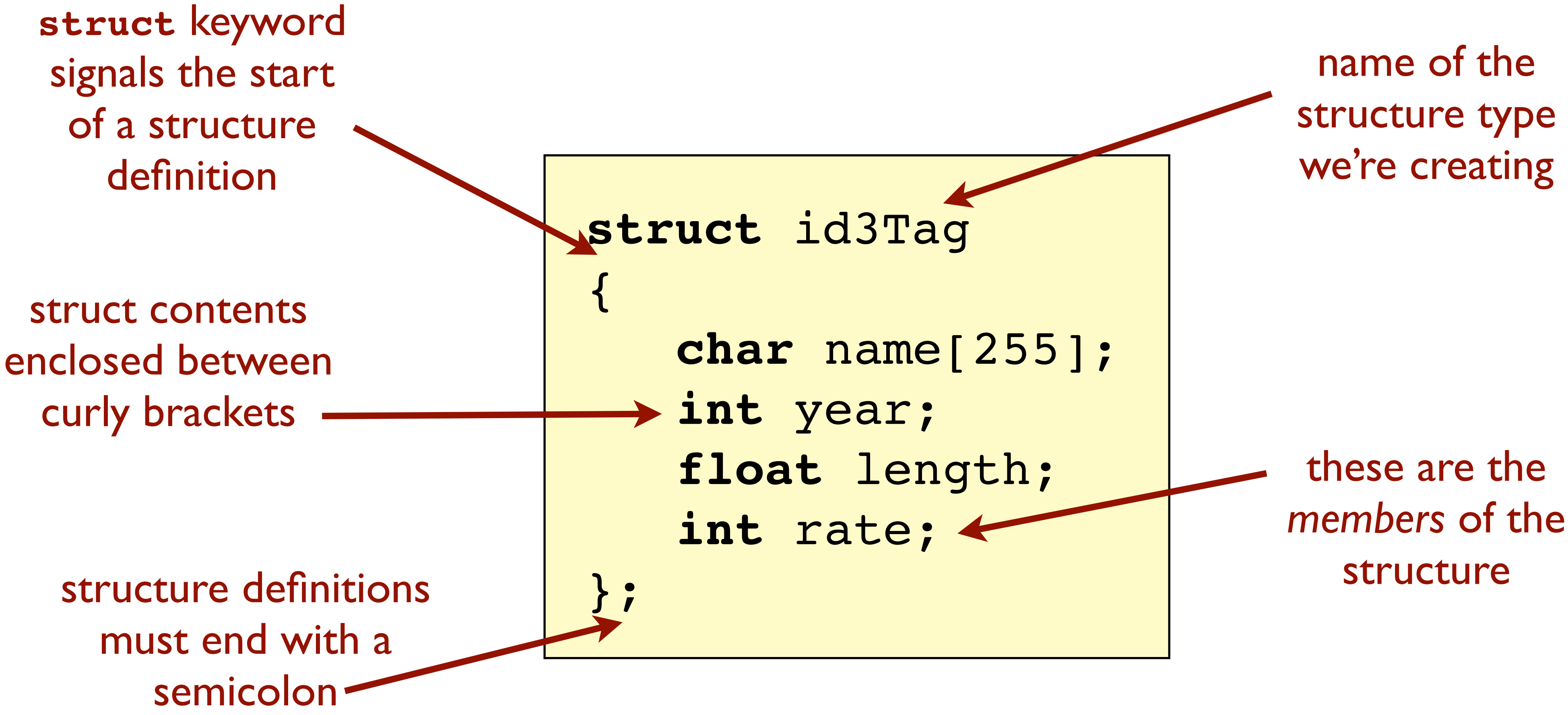
Format: MPEG-1, Layer 3
Channels: Joint Stereo
ID3 Tag: v1.0
Encoded with: Unknown

Where: Fred:Users:greg:Music:iTunes:iTunes Music:Compilations:Austin Powers Soundtrack:Soul Bossa Nova.mp3

```
char name[255];
int year;
float length;
int rate;
```

Introducing `struct`!

- ... but it makes more sense to group them all together in a single data type, which we get to define
- We can do this with a C++ concept called a structure



`struct` keyword signals the start of a structure definition

name of the structure type we're creating

struct contents enclosed between curly brackets

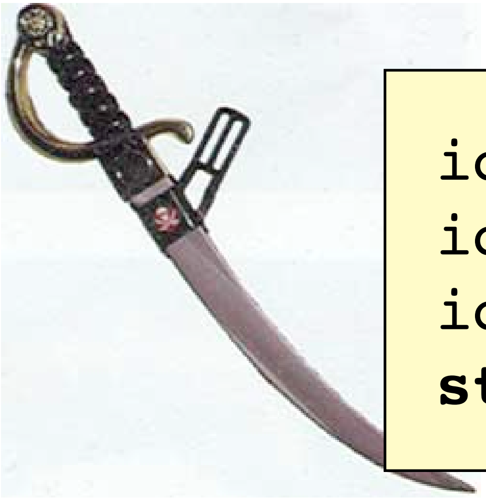
```
struct id3Tag
{
    char name[255];
    int year;
    float length;
    int rate;
};
```

these are the *members* of the structure

structure definitions must end with a semicolon

Our Very Own Data Type!

- So now we have our very own data type, called `id3Tag` that we can use - at this point `id3Tag` can be treated just like any built-in type
- We can declare variables of type `id3Tag` the same way we would with any other type:



```
id3Tag soulBossaNova;
id3Tag* ptrToSong;
id3Tag U2[50];
struct id3Tag ticketToRide;
```

- Note that we can also treat the word struct like it's part of the type - this is a holdover from C

The Rules

- Structure members can be of any type
- Arrays can be structure members
- A structure can be a member of another structure
- A structure **can't** contain an instance of itself.
- It **can**, however, contain pointers to itself.

```
struct node  // bad
{
    int payload;
    node variable;
};
```

```
struct node // OK
{
    int payload;
    node* variable;
};
```

Accessing structures

- Statically allocated structures are accessed using the dot operator (the period):

```
id3Tag soulBossaNova;
soulBossaNova.year = 1982;
cout << soulBossaNova.year << endl;

id3Tag U2[50];
strcpy( U2[5].name, “Beautiful Day” );
```

- Members of a structure can be accessed and used like regular variables, because they *are* regular variables - just grouped with others.

Accessing structures 2

- Accessing through a pointer (as with any dynamically created structure) uses a different access mechanism: the arrow (–>) operator

```
id3Tag* soulBossaNova = new id3Tag;
soulBossaNova->year = 1982;
```

- Mixing up access operators will cause a compiler error
- What would be another way of accessing the `year` member?

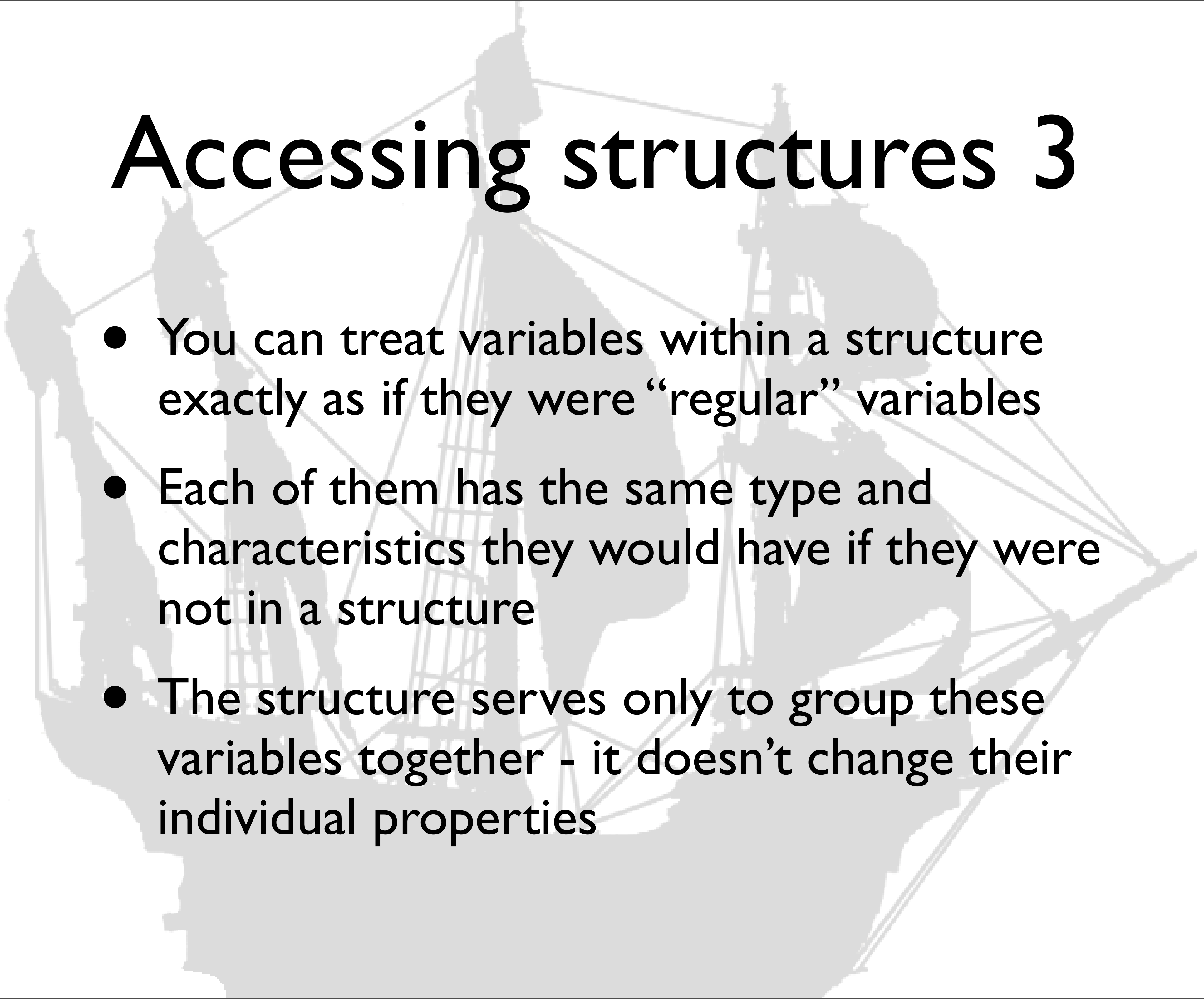
Accessing structures 2

```
id3Tag* soulBossaNova = new id3Tag;
soulBossaNova->year = 1982;
```

- Note that we're doing dynamic memory allocation here - this works the same way as it does for all the "regular" types
- This is where dynamic allocation actually gets useful (we see this more later)
- Remember, we have to clean up after ourselves:

```
delete soulBossaNova;
```

Accessing structures 3



- You can treat variables within a structure exactly as if they were “regular” variables
- Each of them has the same type and characteristics they would have if they were not in a structure
- The structure serves only to group these variables together - it doesn’t change their individual properties

Passing Structures

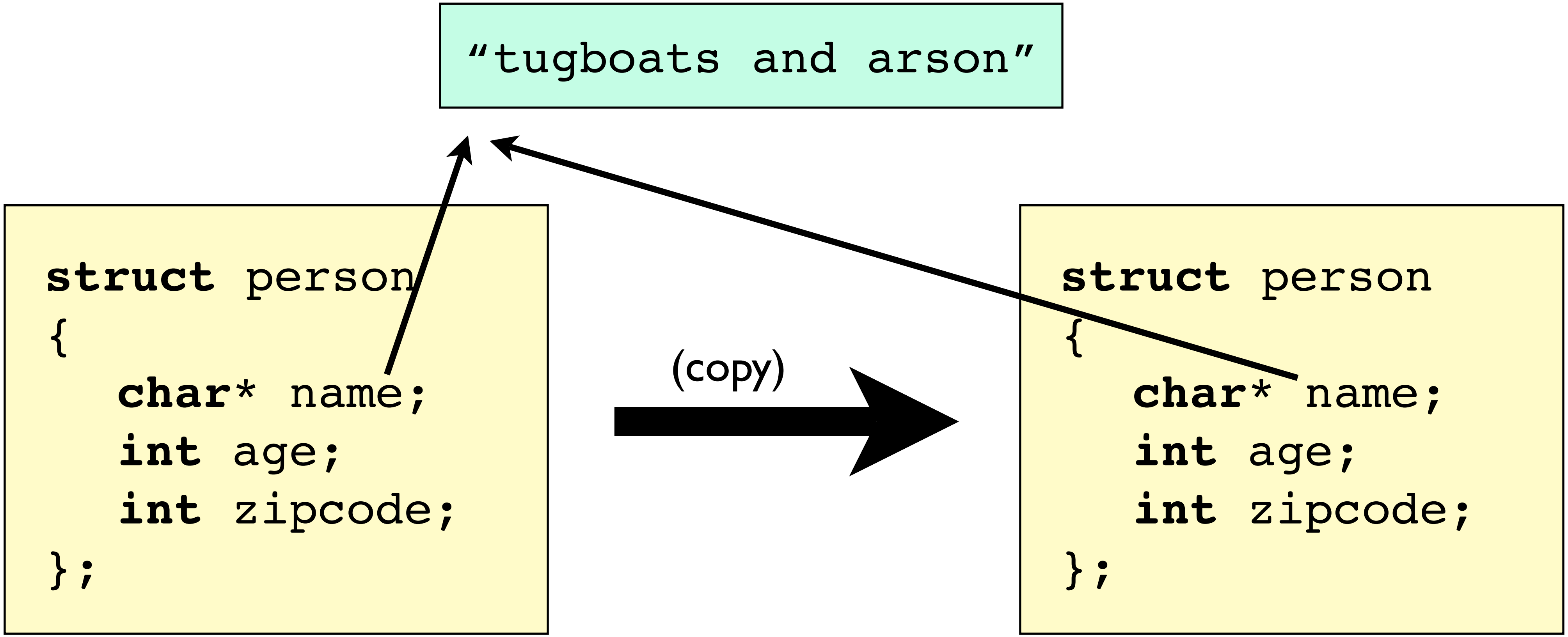
```
struct video
{
   int* frame;
   int list[10];
   int title;
};
```

```
void func( video v );
```

- A structure can be passed as a parameter to a function, just like any other type
- By default, structures are passed by value.
- When/why would you want to pass by reference instead?
- What are some potential problems in passing by value?

Passing Structures By Value

- When structures get passed by value, each member of the structure gets *copied*.
- This becomes a problem when a structure contains pointers:



`“tugboats and arson”`

```
struct person
{
   char* name;
   int age;
   int zipcode;
};
```

(copy)

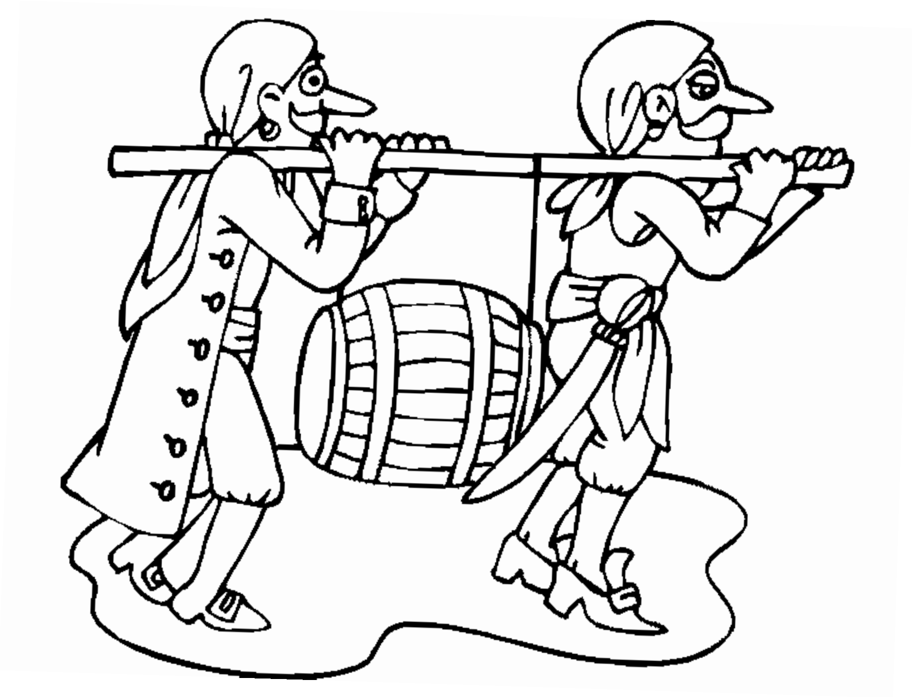
```
struct person
{
   char* name;
   int age;
   int zipcode;
};
```

... back to structures

- Structures can include pointers to other structures of the same type
- This is how we can start to create more complicated data structures: lists, trees, graphs, etc.
- An example (from a few slides back): here’s what each node of a linked list looks like:

```
struct node
{
   int payload;
   node* next;
};
```

points to another instance of the node structure



Example: Linked Lists

- Let’s make a simple linked list structure
- ... and some code that will add integers to it
- This will tie directly into your assignment!