
# More Strings, Structs, Project Two

# Some string review

- How are strings represented in C++?
- What is a NULL terminator?
- How much memory should you allocate for a string?
- What very important thing do you need to do when putting characters into a string?
- What's a fast way of declaring and initializing string variables?
- How do we embed a newline or a quotation mark in a string

# 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...

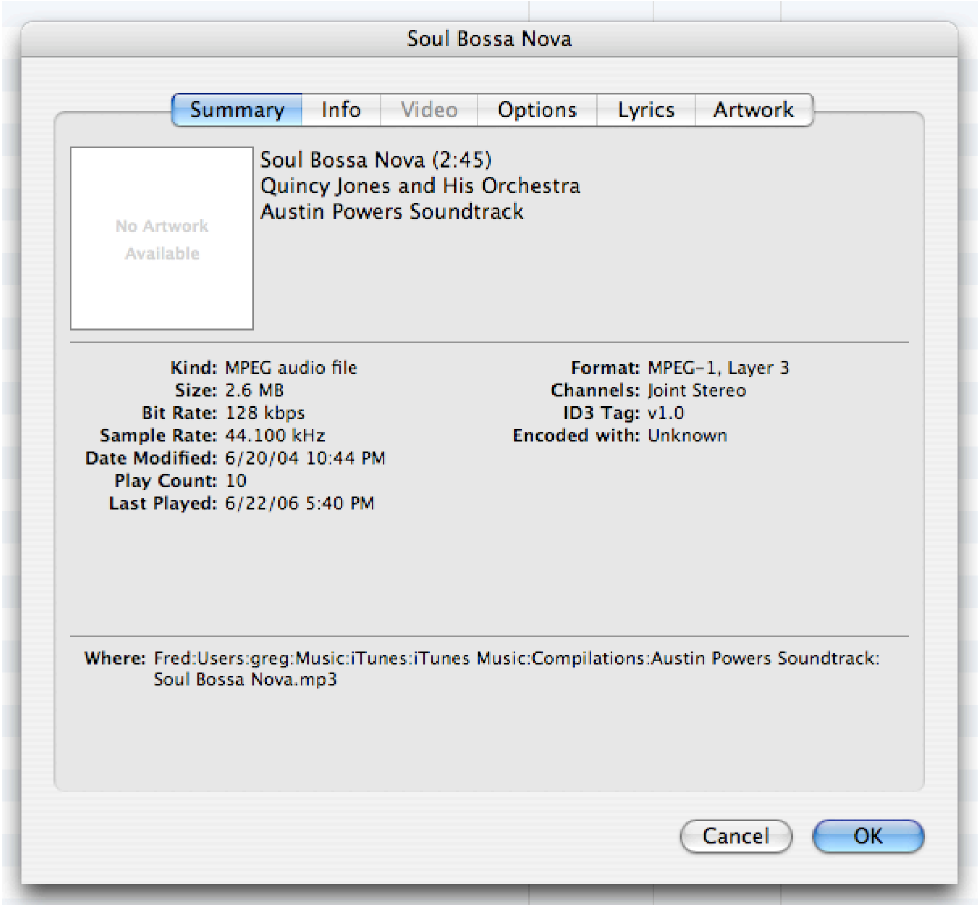
- The C standard library has many functions for working with strings:
  - formatting/modifying them
  - copying/manipulating them
  - converting them back and forth from integers, floats, etc.
  - ... and so on
- Google "string.h" and read about these if and when you need them!

# 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




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

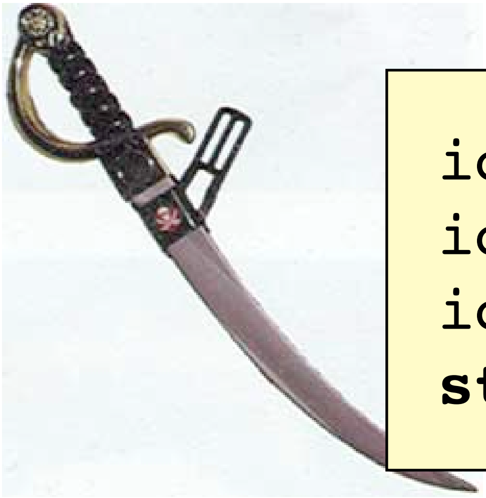
struct contents enclosed between curly brackets

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

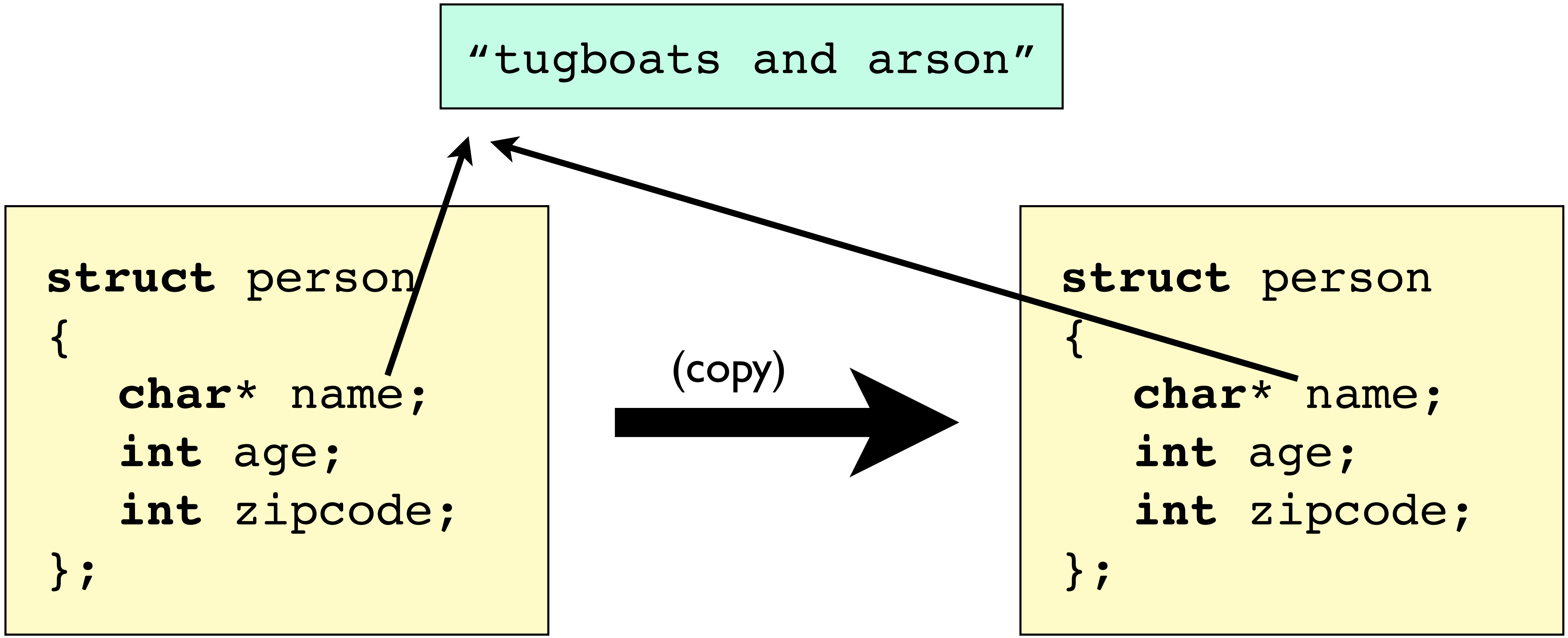
```
struct video
{
   int* frame;
   int list[10];
   int title;
};
```

```
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:

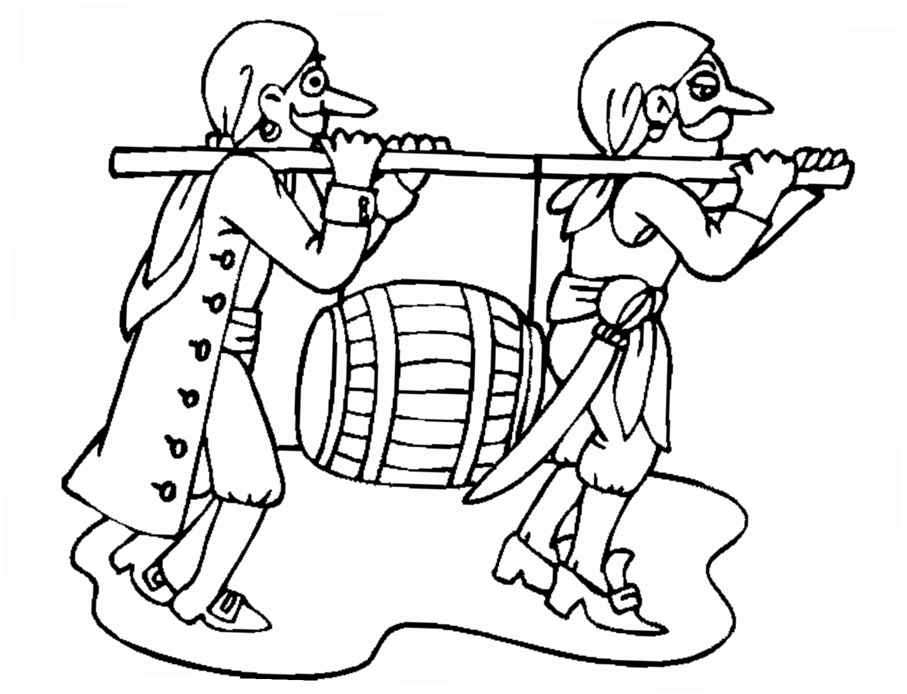


# ... back to structures

- 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!

# Project Two

Write a program that allows the user to enter words and counts their frequency

- Use an alphabetized linked list that stores the word and its count
- Whenever a word is encountered, insert it in the list (if it isn't there already) and increment its count
- At the end, print out all the words (in alphabetical order) and their frequency

# Project Two

**The tricky bits:**

- Checking if a given word is already in the list
- Inserting into the linked list (in alphabetical order)
  - ... these two can be done in one step!
- Properly cleaning up the linked list