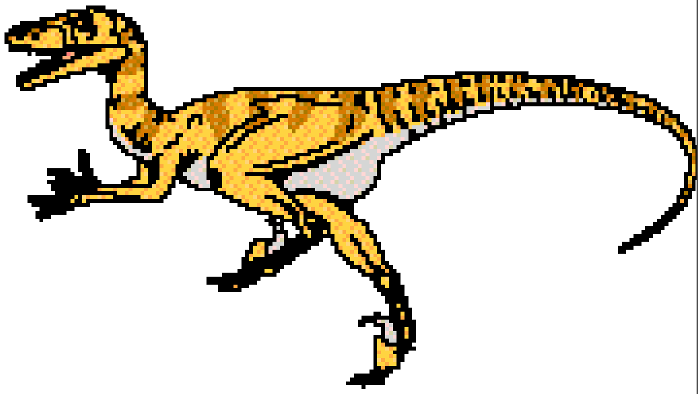
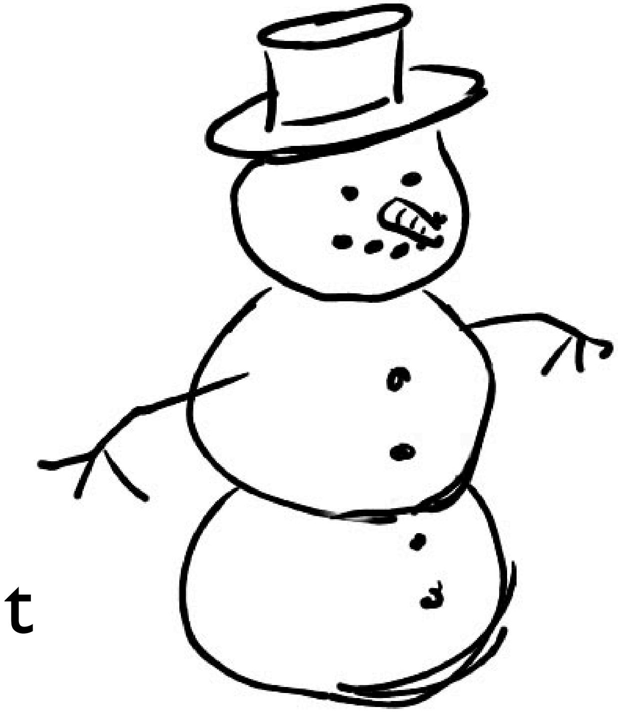
THE BASICS OF C

Review

- What does **dynamic_cast** let you do? Why is it sometimes preferable to C-style casting?
- How do you throw an exception?
- How do you catch an exception?
- Once an exception is caught, where does execution pick back up?



The Basics



- Compiling is a multi-stage process
- In the *first* stage, the code gets sent through the **preprocessor**
- The preprocessor handles the code before the actual compiling process starts
- Once the preprocessor has handled (and maybe changed) the code, the compiler gets to compile that code

Preprocessor Uses

- There are typically three uses for the preprocessor:
  - *code* - include a code file, skip chunks of code, conditionally include code, etc.
  - *constants* - define constants
  - *macros* - typically, small “functions” that are expanded at compile time
- Preprocessor typically start at the left edge of the screen, and always start with the **#** symbol (know any?)

#include

- The #include statement is actually a preprocessor directive
- It tells the compiler to “paste” the included file in place of the #include statement
- The compiler “sees” it as one long file

```
#include <iostream>
```

Constants

- We can use the **#define** directive like this:

```
#define PI 3.14159
```

- Now every time PI is used in that source file, it will be replaced with 3.14159
- This is often used for defining constants (like this one!)
- By convention, #define'd constants are uppercase

#define

- #define works like this:

```
#define [name] [value]
```

- ... but [value] means "anything to the end of the current line"

- Be ye careful:

```
#define PI 3.14  // I like pie!

x = PI + 1;
```

In other words...

- PI (or whatever) is going to get replaced with *exactly* what is in the #define directive

```
#define PI_PLUS_ONE 3.14159 + 1

x = PI_PLUS_ONE * 5
```

- What is wrong with this? What could be done to fix it?

... and one more thing...

- It's possible to #define a name *without* giving it a value.

```
#define GREG_WAS_HERE
```

- GREG_WAS_HERE is now defined, but doesn't have a value
- This can be useful in conjunction with another set of directives, as we'll see later

Conditional Compilation

- The preprocessor can be used to determine if a chunk of code will ever make it to the compiler
- There's a whole set of conditional directives:
  - #if, #elif, #else, #ifdef, #ifndef

#if

- The #if statement takes a numerical argument that evaluates to **true** if the argument is non-zero.
- Every #if block must end with an #endif

DATA

```
#if 3*4
void doStuff()
{
    // does stuff
}
#endif
```

this can be a simple numerical expression - but it can't use any variables or functions - why?

what happens if the condition evaluates to zero?

#if commenting

- The #if statement can be a fast way to "comment out" large blocks of code:

```
#if 0
void doStuff()
{
}

void doMoreStuff()
{
}
#endif
```

- The code between the **#if 0** and **#endif** never gets to the compiler
- From the compiler's perspective, it's as if that code doesn't exist!

a few of

The Others

... #else and #elif

```
#if X == 1
  printf( "one\n" );
#elif X == 2
  printf( "two\n" );
#else
  printf( "three\n" );
#endif
```

- **#else** is an else; **#elif** stands for else-if
- They work pretty much like you'd expect
- The entire block still needs to end with **#endif**

#ifdef

- The **#ifdef** directive is like **#if**...
- Instead of checking a numerical value, it checks to see if the argument is *defined*

```
#ifdef INC_DOSTUFF

void doStuff()
{
}

#endif
```

this checks to see if INC_DOSTUFF was defined, either with or without a value

for this to work, there would need to be a **#define** INC_DOSTUFF earlier in the code

One Application...

```
// data.h
class data
{
    int x;
};
```

```
// stuff.h
#include “data.h”
```

```
// main.cpp
#include “data.h”
#include “stuff.h”
```

- We touched on this earlier in the semester...
- It’s easy to accidentally include the same header file multiple times
- data.h is getting pulled into main.cpp directly, *and* via stuff.h
- What is the problem with this?

Include Guards

- We can use the preprocessor to make sure the same header only gets included *once* per source file:

```
#ifndef DATA_H
#define DATA_H

class data
{
    int x;
};

#endif
```

#ifndef - is true if the argument is *not* defined

if DATA_H is not #defined, then it has never been included; include it and then #define it so it won't be #included again

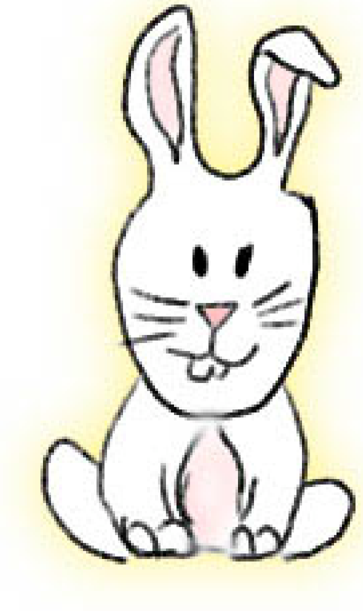
Macros

- The other major use of the preprocessors is to define *macros*
- A macro is a #define that can accept arguments:

```
#define MACRO_NAME(arg1, arg2, ...) [code to expand]
```

- Macros aren't of any particular type
- They get "expanded" directly into the code

Tricksy Macros



- A simple example:

```
#define MULT(x, y) x * y
```

- We'd use the macro like this:

```
int z;
z = MULT(3 + 2, 4 + 2);
```

- What would you expect this to expand to? What *does* it expand to? How do we fix this?

How ‘bout this one?

- Another simple macro:

```
#define ADD_FIVE(a) (a) + 5
```

- But are problems is we use it like this:

```
int x = ADD_FIVE(3) * 3;
```

- What would you expect this to expand to? What *does* it expand to? How do we fix this?

One more...

- There's a weird trick you can do, using the bitwise exclusive-or to swap two variables
- Here's a macro to implement that:

```
#define SWAP(a, b) a ^= b; b ^= a; a ^= b;
```

- Sometimes this works fine:

```
int a = 5, b = 10;
SWAP( a, b );
```

- When would this *not* work fine? How would we fix it?

Why Macros Suck

- By now you may have realized why people hate using macros:
  - They're picky
  - They often have unintended consequences
  - They aren't typesafe
- Macros were used a lot in C - what is often used instead in C++?

Multiline Macros

- In C/C++, a backslash at the end of the line means "extend this line onto the next line"
- We can use this to make macros easier to read and write
- For instance, we could rewrite the swap macro to look like this:

```
#define SWAP(a, b)  {               \
                        a ^= b;  \
                        b ^= a;  \
                        a ^= b;  \
                    }
```

```
class data
{
   public:
       data();
   private:
       float foo;
       bool isOrd;
       float quux;
}
```

```
data::data()
{
   foo = 5;
   isOrd = true;
   quux = -23.34;
}
```

what's another way of writing this constructor?

```
int*** ptr;
```

What *is* this thing?

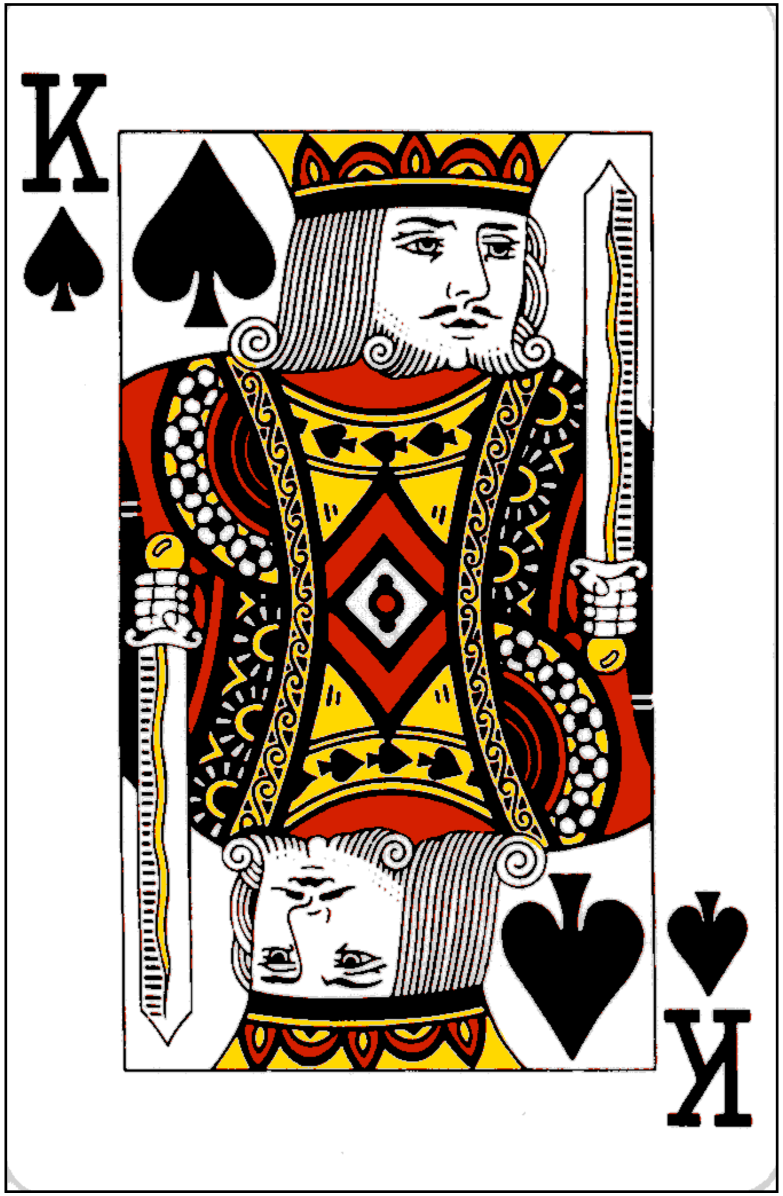
What's it pointing to?

What are the different values we could mess with here?

All About **C**

- Why does this matter?
- Lots of C++ code is actually C code in disguise!
- Everything you can do in C, you can do in C++.
- And vice versa: everything you can do in C++, you can do in C.
  - ... but sometimes it's harder

The Basics



- Designed mainly for efficiency and portability
- Less concerned about programmer niceties:
  - Less type-safe, for example
- Less “behind the scenes” stuff

C Files

- C files usually have a **.c** extension (as opposed to **.cpp**)
  - Sometimes this is important - the extension tells the compiler how to deal with a file
- Like C++, header files have a **.h** extension
- In C++, standard header files usually have *no* extension - `#include <iostream>`
- In C, even the standard header files have **.h** extensions - `#include <stdio.h>`

C Standard Library

- Most of the “built-in” functionality of C comes from functions that are part of the **C Standard Library**
- We’ve used some of this...
- These functions are declared in many different header files:
  - stdio.h, stdlib.h, math.h, string.h, ...

bool

- The **bool** type is new to C++ - there is no boolean type in C
- Instead, all comparisons are of type **int**
- We used this in C++ sometimes too:
    - zero means false, non-zero means true

Struct Variables

```
struct aPoint
{
    int x, y;
};
```

- In C++, once you've declared as structure, you can instantiate it with only the structure name:

```
aPoint a;
```

- In C, the *full* typename is **struct aPoint** - aPoint alone is not enough

```
struct aPoint a;
struct aPoint* pt;
```

Declarations

- C++ lets you declare variables anywhere you want in the code
- In C, declaration statements must be the *first* statements in a block (like a function)

```
doStuff();

for( int i = 0; i < 10; ++i )
{
}
```

bad

good

```
int i;

doStuff();
for( i = 0; i < 10; ++i )
{
}
```

```c
#define SPRING 0
#define SUMMER 1
#define FALL   3
#define WINTER 4
```

enum

- an **enum** is a way of setting up a bunch of named constants
- You might see this done like the code snippet above...
- An alternate way (often used in C) is to use an enum

```c
enum season { SPRING, SUMMER, FALL, WINTER };
```

```
enum season { SPRING, SUMMER, FALL, WINTER };
```

```
enum size { SMALL = 3, MEDIUM = 7, LARGE };
```

- The value of enum constants start at zero and increment each time you move down the list
- Alternatively, some or all constants can be explicitly given a value
- The compiler will convert enum ▶ int, but not int ▶ enum
- Same declaration rules as a struct: **size s;** works in C++. In C it must be **enum size s;**

Type Casting

- C and C++ both support this form of typecasting:

```
int bob = (int)3.14159;
```

- C++ also gives you constructor-style casting:

```
int bob = int(3.14159);
```

- This does not work in C.
- Implicit conversions are mostly the same

Comments

C++ allows single line comments...

```
// this is a comment
doStuff();
```

C only allows comments delimited by **/*** and ***/**, which can be multi-line

```
/* this is a comment,
   and it can go on for
   quite a while */
doStuff();
```

Function Stuff

- C has **no** function overloading
  - What does this mean?
  - How would you work around this?
- Also: **no** default arguments for functions
  - What does this mean?
- In C functions do *not* have to be declared...
  - as long as they are of type **int func()**

Operator Overloading

- In C, there is no operator overloading
- This usually isn't that big of a deal, though...
- What *is* operator overloading, exactly?
- How would you implement something equivalent?

References

- Reference types (int& a, etc.) are new to C++, and didn't exist in C.
- Why does this not matter much?

How do we rewrite this code without using references?

```
void swap( int& x, int& y )
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

iostreams

- In C, there are no iostreams
  - no ifstream, ofstream, cin, cout...
- Instead, there are the functions declared in **<stdio.h>**
- There are several different I/O functions but we're going to focus on just a few of them

printf

```
printf( format, arg, arg, arg ... )
```

- printf is how you print stuff to the screen
- printf can handle a variable number of arguments
- The first argument to printf is the **format string**
- The format string tells printf the *type* of all the forthcoming arguments, and sometimes the *formatting*
- ... or it can just contain regular text

Format String

- The format string can contain regular text, complete with escape sequences

```
printf( “my name is bob\n” );
```

- The types are specified via codes called *type specifiers*, which start with the % character

```
printf( “%i\n”, 42 );
```

- The character that follows the **%** sign tells printf what the type the argument is going to be
- Some common type specifers:
  - **%i** or **%d** = integer
  - **%u** = unsigned integer
  - **%s** = string (character array, NULL terminated)
  - **%f** = floating point
  - **%c** = character

```
printf( “%s’s favorite number is %d!\n”,
         person->name, person->favNum );
```

More Format Strings

- The type specifier can sometimes contain formatting information:

```
printf( “[%d]\n”, 17 );
                                        [17]
printf( “[%5d]\n”, 17 );
                                        [   17]
printf( “[%05d]\n”, 17 );
                                        [00017]
```

- There are a bunch of these, depending on the type specifier - look ‘em up if you’re curious

More I/O

- There are specialized version of the printf function:
  - sprintf - prints the output into a string
  - fprintf - prints the output to a file
- Also input functions:
  - The scanf family - gets output *from* something - a file, a string, the keyboard

void pointers

- So far, every time we've talked about pointers, the pointer has a *type*
  - **int** pointers point to **int**s, etc.
- C has many functions (mainly I/O and memory functions) that deal with chunks of data of *unknown* type
- When a function needs a pointer to data that could be *any* type, it uses a **void***
(a *void pointer*)

void pointers

- C++ tries to be much more typesafe than C
  - More careful about what conversions are allowed
- In C++, implicit casting of void* pointers is not allowed
  - What does this mean?

Example:

```
int fwrite( const void* buffer, int size,
            int count, FILE* stream );
```

- The **fwrite** function writes a block of bytes out to a file, without regard to what *kind* of data its writing
- Any kind of data can be turned into a void*, so we can call fwrite with any kind of data

Dynamic Memory Allocation

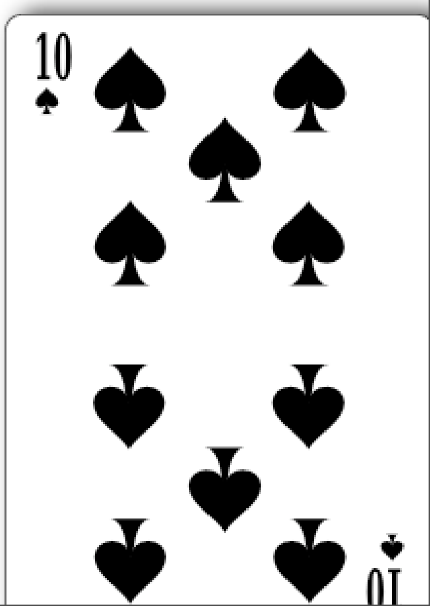
- C has no new/delete operators
- Instead, dynamic memory allocation is handled by a function named **malloc**, which takes the number of bytes needed as a parameter
- **malloc** returns a **void***, which then needs to be cast to the correct type

```
char* str = (char*)malloc( 50 );
```

allocate a character array for how many characters?

Freeing Dynamically Allocated Memory

- In C++, for every **new**, there has to be a **delete** or we get memory leaks
- In C++, for every **malloc**, there has to be a **free**
- free is a function called on a pointer to the allocated memory (just like delete):




```
char* str = (char*)malloc( 50 );
...
free( str );
```

Dynamic Memory Allocation

- In C++, we can request a certain number of a certain type:

```
Cow* array = new Cow[10];
```

- ... and the compiler figures out exactly how many bytes of memory are needed
- In C, we need to know how many bytes we need before calling malloc!
- So we have to be able to figure out exactly how many bytes a **Cow** structure takes up in memory

Introducing: **sizeof**

- **sizeof** is a C/C++ operator that returns the number of bytes something takes
- We can call sizeof with a type:

```
printf( “%d\n”, sizeof(int) );
```

- or we can call it with an *instance* of a type:

```
int bob = 196;
printf( “%d\n”, sizeof(bob) );
```

- How would we allocate an array of 10 **cow**s?