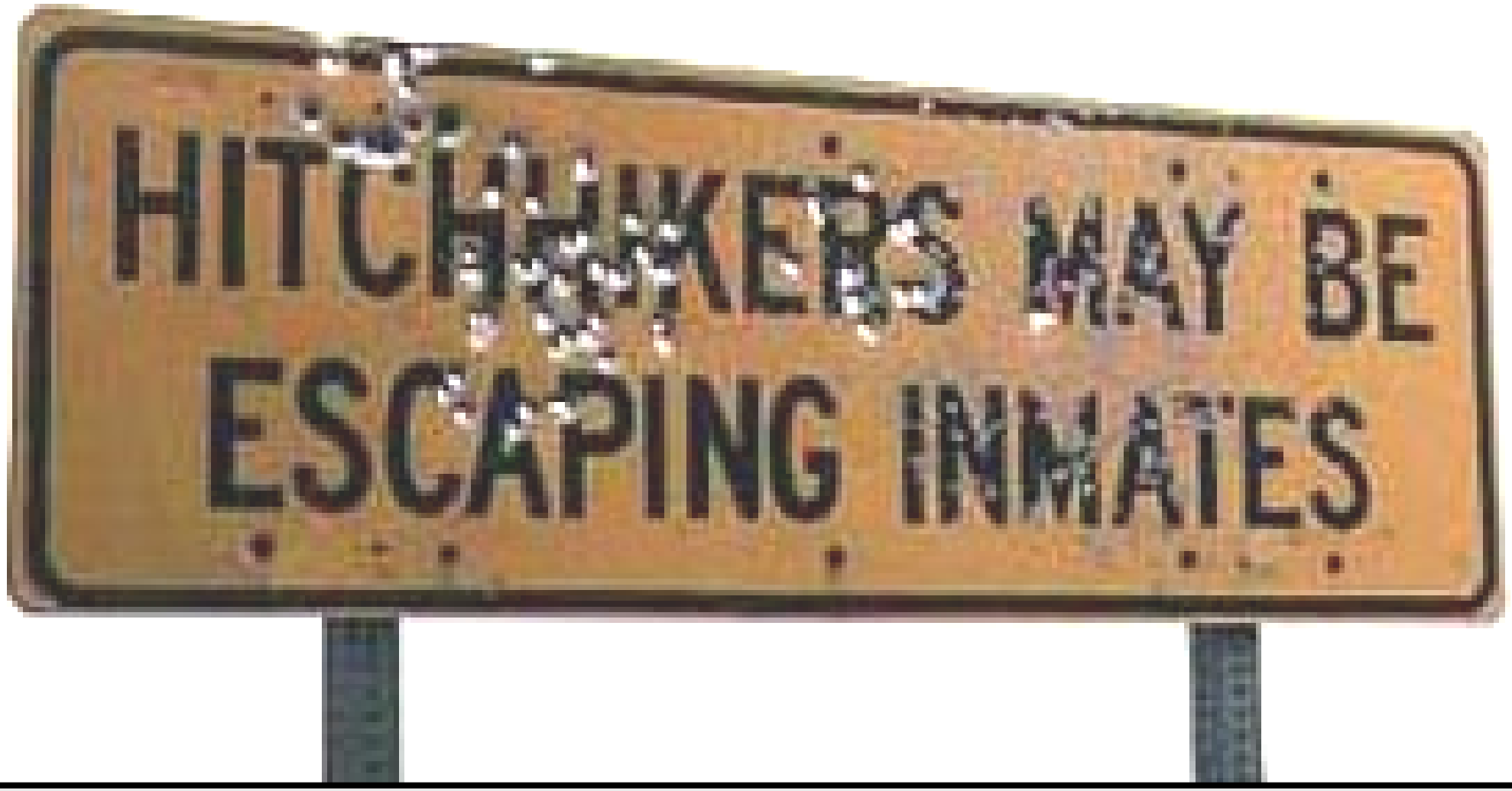


THE BASICS OF C

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

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

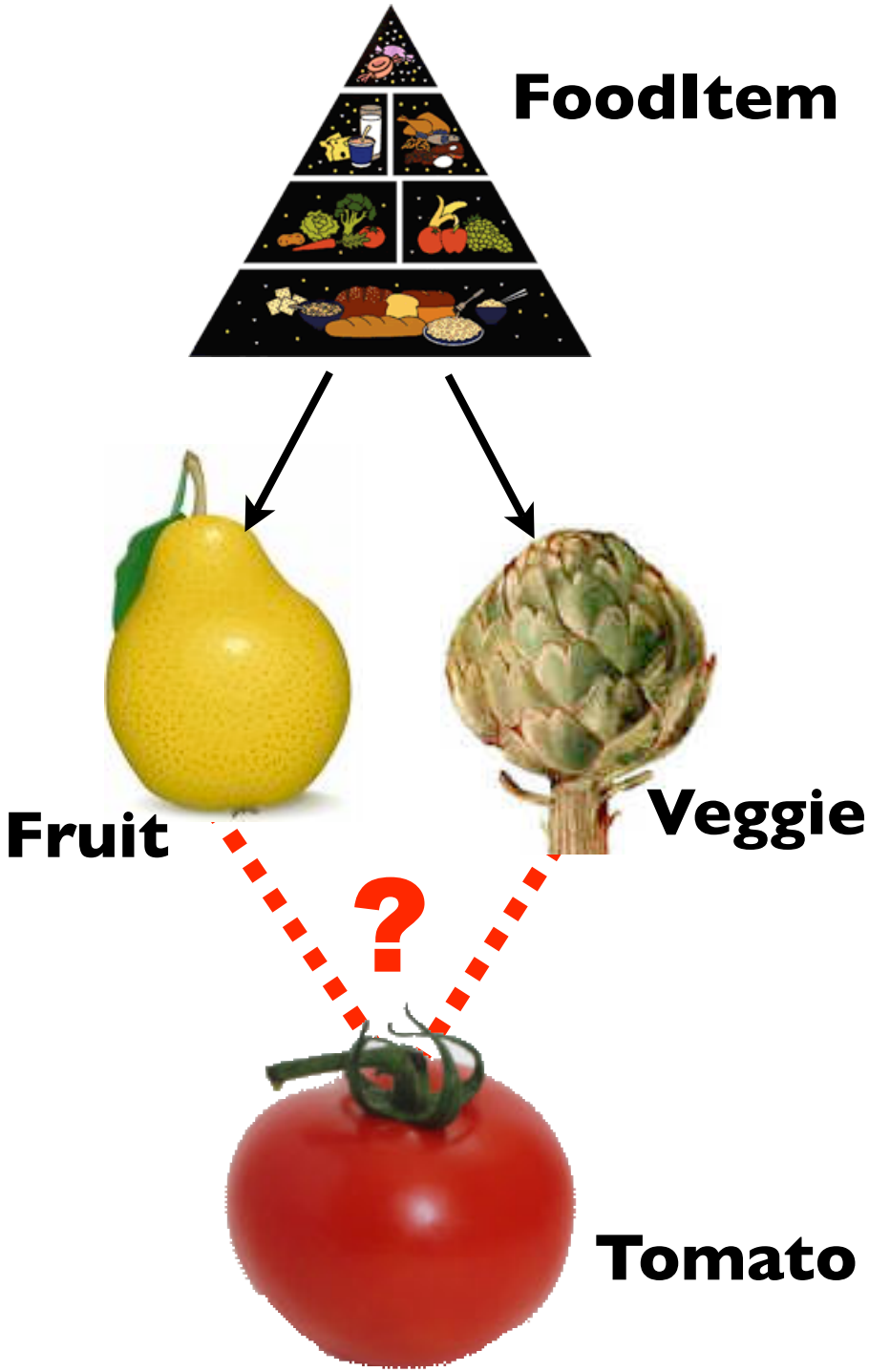
what's another way of writing this constructor?

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

What *is* this thing?

What's it pointing to?

What are the different values we could mess with here?



Nobody seems to be able to agree whether **Tomato** should be derived from **Fruit** or from **Veggie**.

How could we solve this dilemma and make everybody happy?

What would be the problems with doing this, and how might we address those?

```
#include "openfile.h"

try
{
    ifstream inFile;
    char* data = new char[500];
    openFile( inFile );
    inFile.getline( data, 500 );
    delete[] data;
}
catch(...)
{
    cout << "whoops." << endl;
}
```

is this snippet:

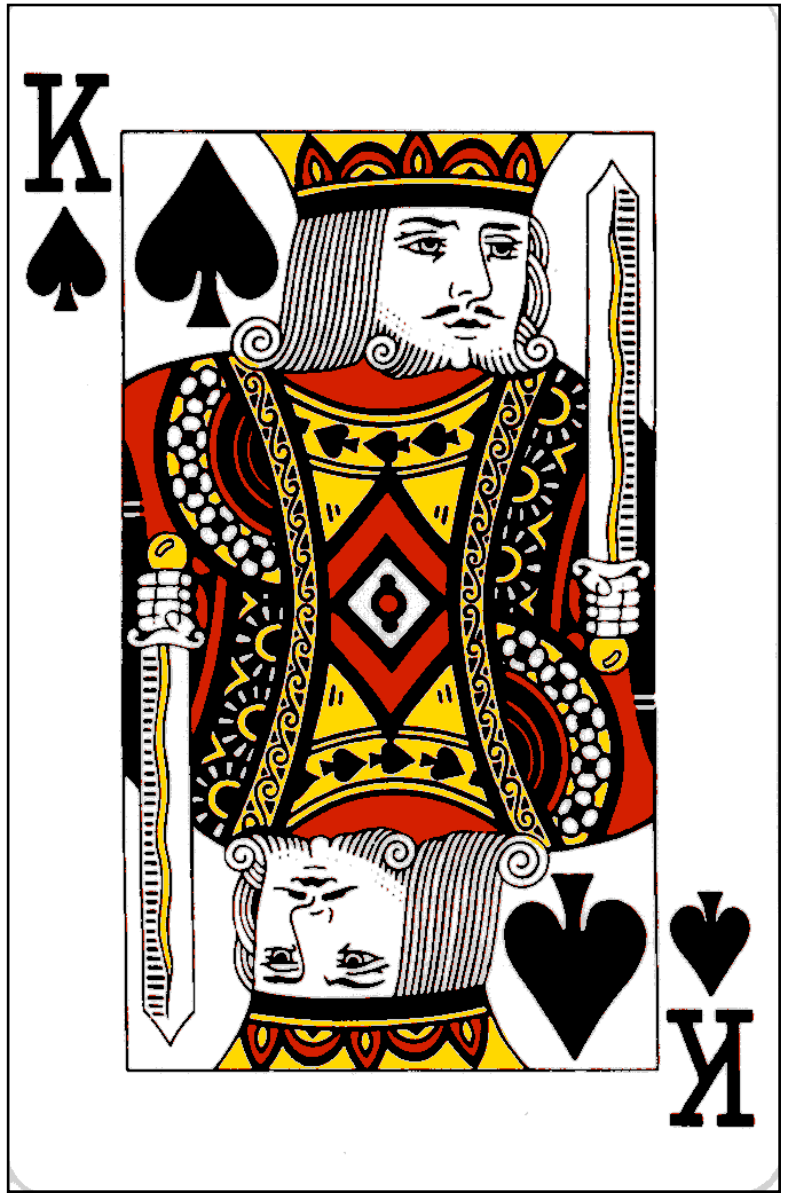
a. good?
b. not good?

why might it be not good?

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>`

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 )
{
}
```

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 largely the same

Comments

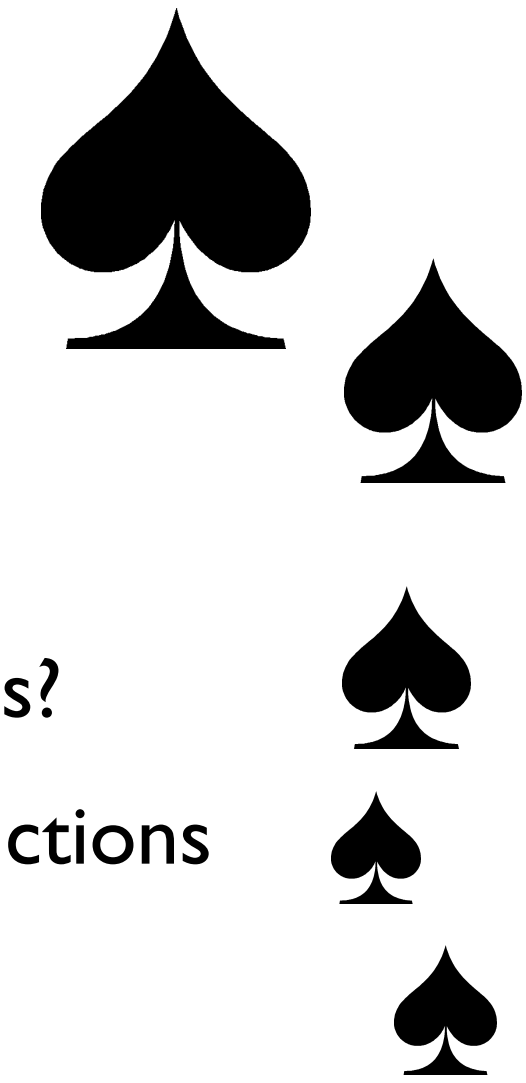
C++ allows single line comments...

```
// 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?

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*)

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

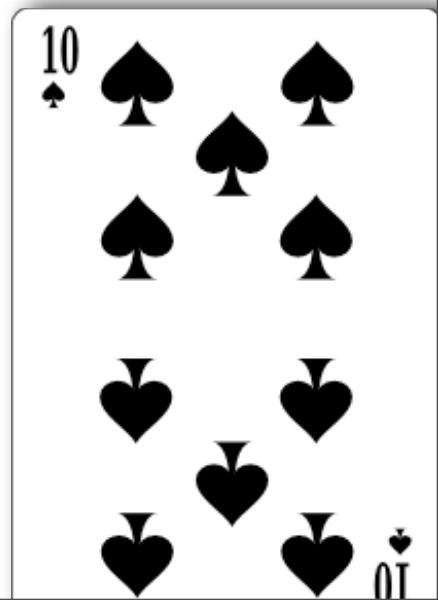
```
char* str = (char*)malloc( 50 );
```

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?