bigger picture: connecting to engineering big software systems

c++ classes
goals
background
modules

next time: constructors & destructors
access control

another common use of function pointer tables
(jump tables) in C

abstract interface to 'device'
- interrupt controllers (how behaves differently)
- file operations
- device drivers

what kind of 'file'?

Saturday 4-6
extra office hours for me / review

P12... shift to Tuesday now
The organization of data and function specialization that we examined in the last two lectures forms the core of data and function inheritance in C++.

What are those? If parent has a field, so does the child (data inheritance). If function can operate on a parent, it can operate on a child (function inheritance).

Ways to make programming big systems easier:

Avoid replicating code — when one type of structure is just a special kind of another type, most of the data and behavior (functions) are identical.

Simplify usage — in example, we can write functions that work for all references in the library; we don't always want to do things the same way for more specialized references (books, conference papers, etc.), so we allow specialization (replacement) of functions (using function pointer tables in C/c++)

Simplify extensibility — only write/change behavior that is new/different — reuse other code & data whenever possible.
background

50's software systems large enough that they are divided into modules to make reasoning about them easier.

late 60s & 70s
object-oriented languages developed
(Simula, Smalltalk)

1972
David Parnas: information hiding
- module defines an interface (functions)
- how functions are implemented and what information (fields, data structures) are used should be hidden from other modules

79 C with classes

80s C++ takes off — blends benefits of object-oriented design with clarity & performance of C.
Typically, a module is organized around one of more data structures.

Recall that a data structure is:
- a struct with fields [sometimes >1]
- related static data
- interface functions that operate on one or more instances
- internal/implementation functions
- initialization/teardown routines for an instance

Module also includes initialization/teardown routines for the module.

\[
\text{C++ smallest module: a class}
\]

\[
\text{class name : public parent class }
\]

\[
\text{// fields declared - same as a struct}
\]

\[
\text{// associated functions declared (interface and implementation)}
\]

\[
\text{// static data related to class also appear here}
\]

\[
\text{// initialization/teardown routines for an instance}
\]

\[
\text{Let's examine each in more detail --}
\]
Fields — identical to structs

- add a line that looks like a variable declaration
  
  ```
  int x;
  double y;
  player* p;
  ```

- now each instance has a field named x, y, or p
- layout in memory matches order in class definition
- fields in a class come after fields in parent class (identical to how we drew for C)

Functions — syntax introduced to make easier

Often a function takes a pointer to an instance

Function declared in class definition using style you know is a *member function* or method

```
int memberfunction (char x, double* y);
```
In other words, if we have the class: `ALPHA`, with member function declared:

```
int memberfunc (char x, double y);
```

The real implemented in same way as C function, with clear translation to assembly & calling convention is:

```
int memberfunc (ALPHA* this, char x, double y);
```

Note: first argument has a name (not chosen by you): "this"

Using a member function: like a field:

```
ALPHA a;
ALPHA* ptr;

a.memberfunc ('A', NULL); // this is &a
ptr->memberfunc ('Z', NULL); // this is ptr
```

What if I don't want an implicit pointer to class?

Add 'static' to front of function declaration in class definition called a class function.
Invoking a class function:

Everything in a class definition is implicitly prefixed with the class name (avoid conflicts in naming).

But often compiler can figure out which thing you want:

`alpha a;
alpha::member_func(...)
^ the function defined in ALPHA
\_ \_ a is an ALPHA

Often not true for class functions.

In ALPHA

\texttt{static void class_func (int g2);} 

To use: new namespace operator – join pieces of namespace together

\texttt{ALPHA::class_func (42);}
Inheritance

Mentioned already for data

```java
class ALPHA : public BETA
```

An ALPHA has all fields of a BETA
(Data inheritance)

We can also invoke any member function
of BETA on an ALPHA
(Function inheritance)

If BETA and ALPHA both define a function,
the compiler will call the one corresponding
to the current type.

```
BETA b1; ALPHA a1; // compiler approves
b1 = &a1; // casts implicitly
```

If you want the behavior that we implemented in C

- using function pointer tables
- add "virtual" before
- myfunc declaration in BETA

ALPHA recommended, but optional