C++ Object Oriented Programming

CSE 333 Winter 2021

Instructor: John Zahorjan

Teaching Assistants:

Matthew Arnold	Nonthakit Chaiwong	Jacob Cohen
Elizabeth Haker	Henry Hung	Chase Lee
Leo Liao	Tim Mandzyuk	Benjamin Shmidt
Guramrit Singh		

Lecture Outline

- Object Oriented Programming
- Objects
  - Data Layout
  - Methods
  - Static Dispatch
- Inheritance
  - Derived Class Layout
  - Dynamic Dispatch
- Syntatic Sugar / Software Engineering Issues

Object Oriented Programming

- Object oriented programming is a way of abstracting functionality (code) and data (variables)
  - It's a way of thinking about the abstract solution to the programming problem
- C offered close to nothing to help organize
  - The method namespace is global
    - Can use a convention: HashTable_xxx, for instance, but compiler doesn't know anything about it
    - Can define interfaces in comments in .h files, but the language doesn't enforce them
    - Variable scope is local or global, and that's about it
      - You can hide some details in xxx_priv.h files, or in the .c file itself, to make it harder for other programmers to peak under your covers

OOP and C++

- C++ provides a set of language-provided abstractions that are common to object oriented languages
  - Very similar to Java, in some ways
  - Different enough in enough ways to be trouble for a Java programmer's instincts
- Java is objects right down through execution time
- C++ is more like object-ish semantics during compilation, and even run time, except when it isn't
  - When it isn't, it's likely because you're doing something unusual

Not OOP – 1 person

```
char * pName;
char * pAddress;
int      ssn_;

void printMe(FILE* fOut) {
  fprintf(fOut, "Name: %s\nAddress: %s\nSSN: %d\n", pName, pAddress, ssn_);
}

int main(int argc, char *argv[])
{
    pName = "John Zahorjan";
    pAddress = "Home";
    ssn_ = 1;

    printMe(stdout);

    return 0;
}
```

Not OOP – 2 people

```
char * pName[2];
char * pAddress[2];
char * stateName[50];
int      ssn_[2];

void printMe(FILE* fOut, int i) {
  fprintf(fOut, "Name: %s\nAddress: %s\nSSN: %d\n", pName[i], pAddress[i], ssn_[i]);
}
int ChangeAddress( int i, const char *pNewAddress) {
…
}
int main(int argc, char *argv[])
{
    for (int i=0; i<2; i++ ) { initialize pName[i], pAddress[i], and ssn_[i]; }
    ….
    ChangeAddress(j, update);
    ….
    for (i=0; i<2; i++)
       printMe(stdout);

    return 0;
}
```

When you read (and write) the code, it tends to be control-centric.
The programming style doesn't provide any obvious association between data and code, nor between code and code (nor data and data).

OOP in a Non-Object Language

```
typedef struct ll {
  int               num_elements;  //  # elements in the list
  LinkedListNode   *head;  // head of linked list, or NULL if empty
  LinkedListNode   *tail;  // tail of linked list, or NULL if empty
} LinkedList;

LinkedList* LinkedList_Allocate(void);
void LinkedList_Free(LinkedList *list, LLPayloadFreeFnPtr payload_free_function);
int LinkedList_NumElements(LinkedList *list);
void LinkedList_Push(LinkedList *list, LLPayload_t payload);
bool LinkedList_Pop(LinkedList *list, LLPayload_t *payload_ptr);
void LinkedList_Append(LinkedList *list, LLPayload_t payload);
void LinkedList_Sort(LinkedList *list, bool ascending,
                     LLPayloadComparatorFnPtr comparator_function);
```

Language offers no direct help, we're just doing the best we can.

- Make sure we allocate full "object", not just part
  - Make sure we delete it all, not just part
- Make us identify object abstractions, rather than just names and addresses and ssns
- Make code go through interfaces (and not access data directly)

Note that this is mainly about reading and writing code, not about executing code.

Objects: Data

```
typedef struct person_t {
  char * pName;
  char * pAddress;
  int      ssn;
} Person;
```

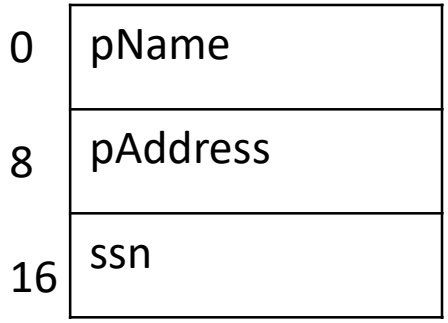
```
Class Person {
 public:
  Person(int ssn=0) { ssn_ = ssn; }
  ~Person();
   std::string  & name() { return name_; }
    std::string & address() { return address_; }

 private:
  std::string  name_;
  std::string  address_;
  int          ssn_;
};
```

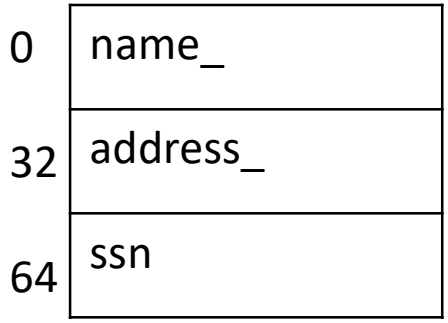
Instances: Data Layout

```
typedef struct person_t {
  char * pName;
  char * pAddress;
  int      ssn_;
} Person;
```

```
Class Person {
  std::string  name_;
  std::string  address_;
  int          ssn_;
};
```

```
Person    me;
Person * you = new Person;
me.ssn_ = 1;
you->ssn_ = 2;
```





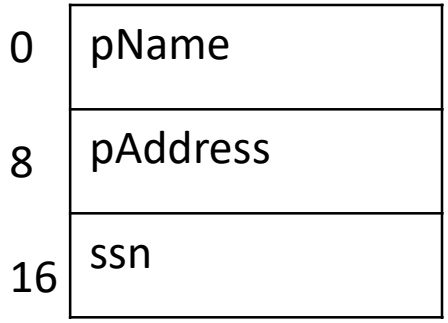
Instances: Data Access

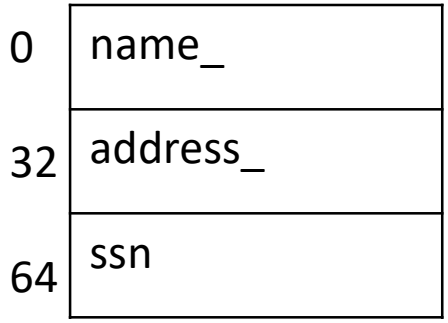
```
typedef struct person_t {
  char * pName;
  char * pAddress;
  int      ssn_;
} Person;
```

```
Class Person {
private:
  std::string  name_;
  std::string  address_;
  int          ssn_;
};
```

```
Person    me;
Person * you = new Person;
me.ssn_ = 1;
you->ssn_ = 2;
```

Compiler does what you asked.
(Not a runtime issue.)





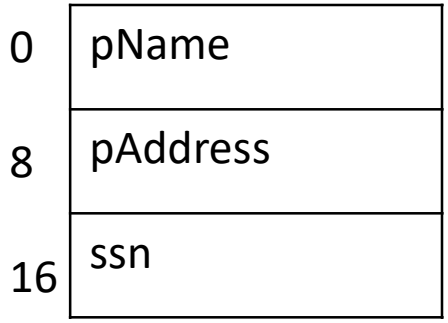
Instances: Data Access

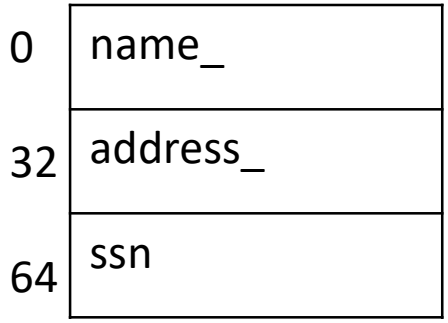
```
typedef struct person_t {
  char * pName;
  char * pAddress;
  int      ssn_;
} Person;
```

```
Class Person {
private:
  std::string  name_;
  std::string  address_;
  int          ssn_;
};
```

```
Person    me;
Person * you = new Person;
me.ssn_ = 1;
you->ssn_ = 2;
```

Compiler does what you asked.
(Not a runtime issue.)





Classes: Methods

- A class
  - Defines the layout of instances
  - A set of methods (that typically operate on instances)
- Classes provide a method to limit which code can be matched in name resolution
  - max(A &myObj, int i);
    - What if there are many methods named "max"?
  - myObj->max(i)
    - "max" must be a method of myObj's class or a base (super) class

Method Name Resolution

- When we declare classes, we must declare all methods
  - Needed by the compiler
  - Handy for the programmer

```
Class Person {
public:
   Person(int ssn=0) { ssn_ = ssn; }
   std::string & name() { return name_; }
   std::string & address() { return address_;}
private:
   std::string  name_;
   std::string  address_;
   int          ssn_;
};
```

```
Person me();
…
me->address() = “Work”;
…
```

Method Name Resolution

```
Class Person {
public:
   Person(int ssn=0) { ssn_ = ssn; }
   std::string & name() { return name_; }
   std::string & address() { return address_;}
private:
   std::string  name_;
   std::string  address_;
   int          ssn_;
};
```

```
std:string address() {
  return "I am happy to accept this award …";
}
```

```
Person me();
…
me->address() = "Work";
…
```

Inheritance: Base and Derived Classes

```
class Person {
public:
   Person(int ssn=0) { ssn_ = ssn; }
   std::string & name() { return name_; }
   std::string & address() { return address_;}
private:
   std::string  name_;
   std::string  address_;
   int          ssn_;
};
```

```
class Child : public Person {
public:
private:
   Adult * guardian_;
};
```

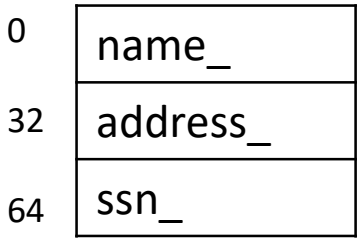
```
class Adult : public Person {
public:
private:
   std::vector<Minor&>  children_;
};
```

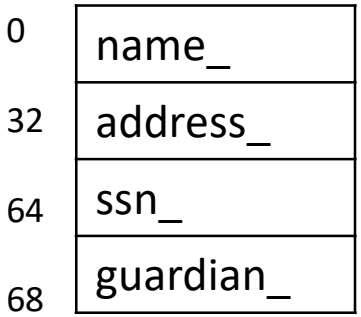
Inheritance: Data Layout

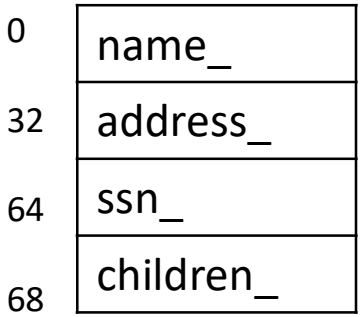
```
class Person {
public:
   Person(int ssn=0) { ssn_ = ssn; }
   std::string & name() { return name_; }
   std::string & address() { return address_;}
private:
   std::string  name_;
   std::string  address_;
   int          ssn_;
};
```

```
class Child : public Person {
public:
private:
   Adult * guardian_;
};
```

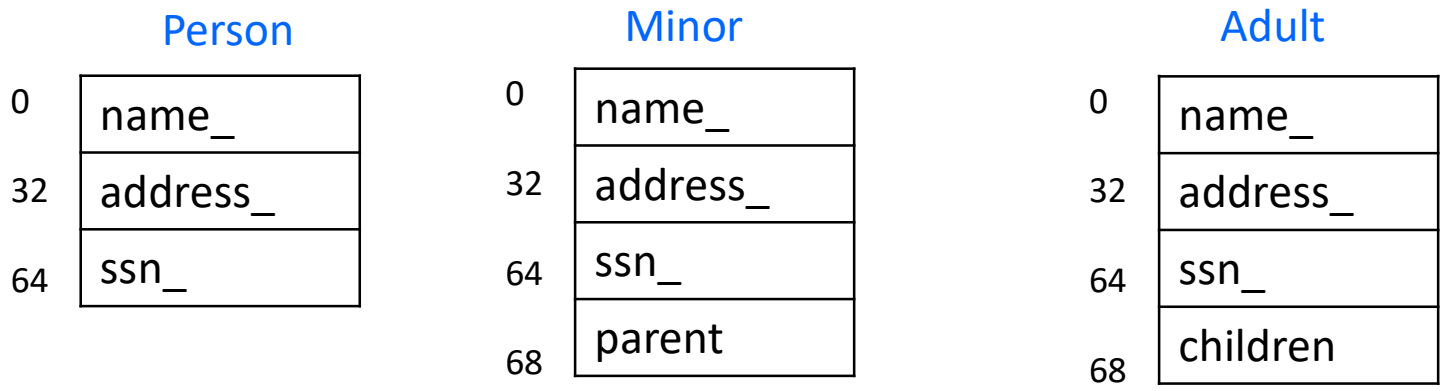
```
class Adult : public Person {
public:
private:
   std::vector<Minor&>  children_;
};
```







Subclass Data Layout



- Minor *m = new Minor();
  m -> name_ = …; // name_ is always at offset 0
  m->address_ = …; // address_ is always offset 32
  m -> parent = …; // m is known to have parent at offset 68

Offsets are determined statically (at compile time)

Person * P = m;
P->parent = …; // compile time error; Adult's don't have parent variables

C++ uses the declared type of the variable to resolve instance variable names.

Base and Derived Class Constructors

```
class Person {
public:
   Person(const std::string & name, const std::string address, int ssn ) { … }
   std::string & name() { return name_; }
   std::string & address() { return address_;}
private:
   std::string  name_;
   std::string  address_;
   int          ssn_;
};
```

```
class Child : public Person {
public:
private:
   Adult * guardian_;
};
```

```
g++ -std=c++17 test.cc
test.cc: In function 'int main(int, char**)':
test.cc:25:9: error: use of deleted function 'Child::Child()'
  25 |   Child c;
     |         ^
```

Base and Derived Class Constructors

```
class Person {
public:
   Person(const std::string & name, const std::string address, int ssn ) { … }
   std::string & name() { return name_; }
   std::string & address() { return address_;}
private:
   std::string  name_;
   std::string  address_;
   int          ssn_;
};
```

```
class Child : public Person {
public:
   Child() { guardian_ = nullptr; }
private:
    Adult * guardian_;
};
```

```
g++ -std=c++17 test.cc
test.cc: In constructor 'Child::Child()':
test.cc:19:11: error: no matching function for call to 'Person::Person()'
  19 |   Child() { guardian_ = nullptr; }
     |           ^
```

Base and Derived Class Constructors

```
class Person {
public:
   Person(const std::string & name, const std::string address, int ssn ) { … }
   …
};
```

```
class Child : public Person {
public:
  Child(const std::string & name, Person &g) {
     name_ = name;
     guardian_ = &g;
  }
…
};
```

```
g++ -std=c++17 test.cc
test.cc: In constructor 'Child::Child(const string&, const Person&)':
test.cc:19:51: error: no matching function for call to 'Person::Person()'
  19 |   Child(const std::string &name, const Person &g) {
…
test.cc:20:5: error: 'std::string Person::name_' is private within this context
  20 |     name_ = name;
```

Base and Derived Class Constructors

```
class Person {
public:
   Person(const std::string & name, const std::string address, int ssn ) { … }
   …
};
```

```
class Child : public Person {
public:
  Child(const std::string & name, Person &g) : Person(name, “”, 0) {
     guardian_ = &g;
  }
…
};
```

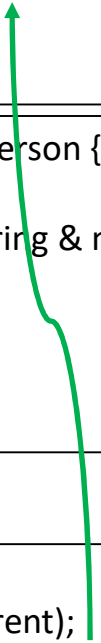
g++ -std=c++17 test.cc

Method Name Resolution: Static

```
class Person {
public:
   Person(const std::string & name, const std::string address, int ssn ) { … }
   std::string & address() { return address_; }
   …
};
```

```
class Child : public Person {
public:
  Child(const std::string & name, Person &g) : Person(name, “”, 0) {
     guardian_ = &g;
  }
…
};
```

```
Person parent(…);
Child child(“kid”, parent);
std::cout << child.address() << std::endl;
```



Name resolution starts with declared class and works up class hierarchy until a matching method is found

Method Name Resolution: Static (cont.)

```
class Person {
public:
   Person(const std::string & name, const std::string address, int ssn ) { … }
   std::string & address() { return address_; }
   …
};
```

```
class Child : public Person {
public:
  Child(const std::string & name, Person &g) : Person(name, “”, 0) {
     guardian_ = &g;
  }
 std::string & address() { return guardian_.address(); }
…
};
```

```
Person parent(…);
Child child(“kid”, parent);
std::cout << child.address() << std::endl;
```

Name resolution starts with declared class and works up class hierarchy until a matching method is found

Method Name Resolution: Static (cont.)

```
class Person {
public:
   Person(const std::string & name, const std::string address, int ssn ) { … }
   std::string & address() { return address_; }
   …
};
```

```
class Child : public Person {
public:
  Child(const std::string & name, Person &g) : Person(name, “”, 0) {
     guardian_ = &g;
  }
 std::string & address() { return guardian_.address(); }
…
};
```

```
Person parent(…);
Child child(“kid”, parent);
Person & j_doe = child;
std::cout << j_doe.address() << std::endl;
```


Name resolution starts with declared class and works up class hierarchy until a matching method is found

Method Name Resolution: Static (cont.)

```
class Person {
public:
   Person(const std::string & name, const std::string address, int ssn ) { … }
   std::string & address() { return address_; }
   …
};
```

```
class Child : public Person {
public:
  Child(const std::string & name, Person &g) : Person(name, “”, 0) {
     guardian_ = &g;
  }
 std::string & address() { return guardian_.address(); }
…
};
```

```
Person parent(…);
Child child(“kid”, parent);
Person j_doe = child;
std::cout << j_doe.address() << std::endl;
```


Name resolution starts with declared class and works up class hierarchy until a matching method is found

and

The “child” part of the data (guardian_) has been sliced (is lost) in the assignment to a Person object (j_doe)

Name Resolution: Dynamic

```
class Person {
public:
   Person(const std::string & name, const std::string address, int ssn ) { … }
   std::string & address() { return address_; }
   …
};
```

```
class Child : public Person {
public:
  Child(const std::string & name, Person &g) : Person(name, “”, 0) {
     guardian_ = &g;
  }
 std::string & address() { return guardian_.address(); }
…
};
```

```
Person parent(…);
Child child(“kid”, parent);
Person & j_doe = child;
std::cout << j_doe.address() << std::endl;
```

What if I want the method name to be resolved using the “actual type” of the object, rather than its declared type?

Have to do name resolution at run time (dynamically).

Name Resolution: Dynamic

```
class Person {
public:
   Person(const std::string & name, const std::string address, int ssn ) { … }
   virtual std::string & address() { return address_; }
   …
};
```

```
class Child : public Person {
public:
  Child(const std::string & name, Person &g) : Person(name, “”, 0) {
     guardian_ = &g;
  }
 std::string & address() { return guardian_.address(); }
…
};
```

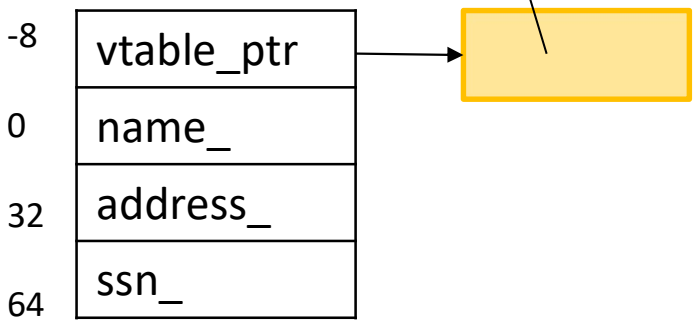
```
Person parent(…);
Child child(“kid”, parent);
Person & j_doe = child;
std::cout << j_doe.address() << std::endl;
```

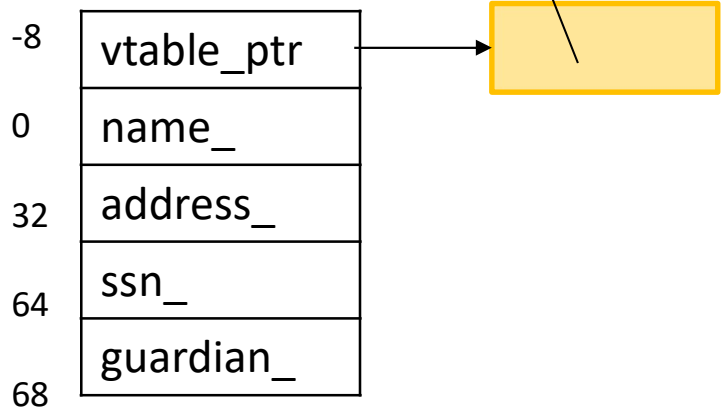
Inheritance: Dynamic Name Resolution

```
class Person {
public:
   Person(int ssn=0) { ssn_ = ssn; }
   std::string & name() { return name_; }
   virtual std::string & address() { return
address_;}
private:
   std::string  name_;
   std::string  address_;
   int          ssn_;
};
```

```
class Child : public Person {
public:
  ….
  std::string & address() { … }
private:
   Adult * guardian_;
};
```

*Person * p;*
…
p->address()



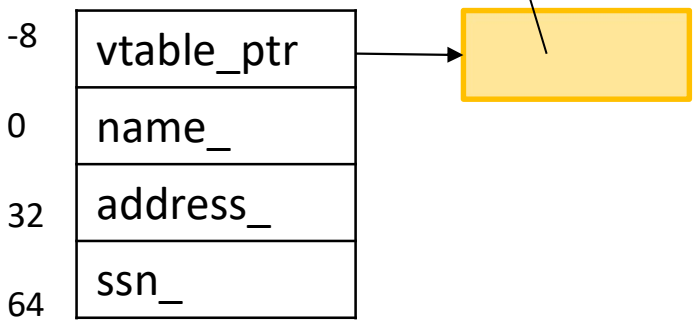


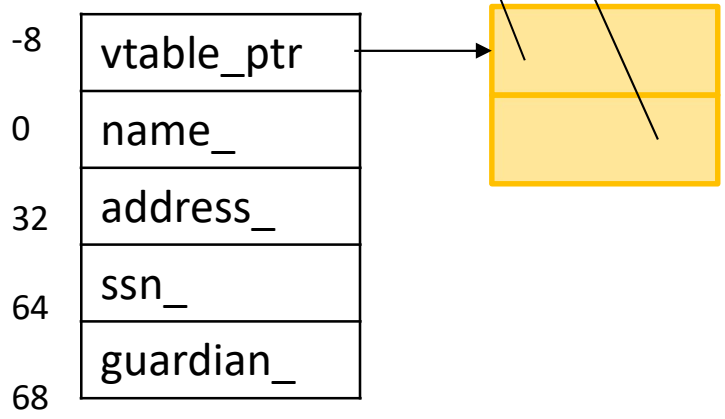
Inheritance: Dynamic Name Resolution

```
class Person {
public:
   Person(int ssn=0) { ssn_ = ssn; }
   std::string & name() { return name_; }
   virtual std::string & address() { return
address_;}
private:
   std::string  name_;
   std::string  address_;
   int          ssn_;
};
```

```
class Child : public Person {
public:
  ….
  std::string & address() { … }
  virtual void anotherVirtual(…);
private:
   Adult * guardian_;
};
```

*Person * p;*
…
p->address()





There's Much More…

- Base class implementations of virtual functions provide a kind of default, in case subclass don't specialize
- You can force them to specialize (so that there's no default)
  - Declare as `virtual someFunction(…) = 0;`
  - A derived class must have a resolvable someFunction()
    - I.e., either it implements it or some parent class below where it was declared as a "pure virtual function" does so
- Called *pure virtual functions*
- Classes that contain them are called *abstract classes*
- Abstract classes cannot be instantiated
  - Only the non-abstract subclasses can be instantiated

Example

(from https://en.cppreference.com/w/cpp/language/abstract_class)

```
struct Abstract {
   virtual void f() = 0; // pure virtual \
}; // "Abstract" is abstract

struct Concrete : Abstract {
   void f() override {} // non-pure virtual
   virtual void g(); // non-pure virtual
}; // "Concrete" is non-abstract

struct Abstract2 : Concrete {
   void g() override = 0; // pure virtual overrider
}; // "Abstract2" is abstract

int main() {
   // Abstract a; // Error: abstract class
   Concrete b; // OK
   Abstract& a = b; // OK to reference abstract base
   a.f(); // virtual dispatch to Concrete::f()
   // Abstract2 a2; // Error: abstract class (final overrider of g()
                    // is pure)
}
```