C++ Part 4

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- Suppressing value semantics
- Separating the Implementation
- Exceptions
- Static Members
- Inheritance
- Virtual Functions
- Virtual Destructors
- Const objects and methods
- Yes. We probably won't get to all these topics. So much C++, so little time.

Special Member Functions … Why

- C++ lets us define classes that can look a lot like built-in types (e.g., the std::string)
- To do a good job of this, we want our class to support *value semantics*
  - What should happen when you pass an instance of our class to a function?
  - Should you be able to return an instance by value?
  - Should you be able to assign between instances?
- Many of these special member functions let us make sure our class does the right thing when you use it like this.

The Assignment Operator

- A special function for copying one object over an existing object.
- It's called when:
  - You assign to an object.
- How is this different from the copy constructor?
  - Here, you already have a destination object; you just need to replace it.

```
SomeClass a;
SomeClass b;
...;
a = b;
```

Assignment operator.

```
SomeClass a;
SomeClass b = a;
```

Copy constructor.

List Assignment Operator

```
class List {
  ...;
  List &operator=( const List &other )
  {
    if ( this != &other ) {
      while ( head ) {
        Node *n = head;
        head = head->next;
        delete n;
      }

      Node **ptr = &head;
      for ( Node *n = other.head; n; n = n->next ) {
        *ptr = new Node;
        (*ptr)->val = n->val;
        ptr = &((*ptr)->next);
      }
      *ptr = NULL;
    }

    return *this;
  }
  ...;
};
```

Protection in case we say: a = a;

Free the old list.

Copy over the source list.

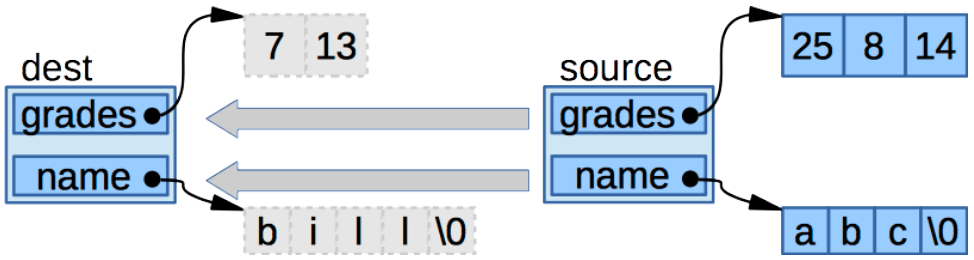
Return the resulting list by reference.
In case we want to say: a = b = c;

Looks a lot like a destructor then a copy constructor.

Compiler-Generated Assignment Operator

- If you don't define an assignment operator, the compiler will write one for you.
- What does it do?
  - It just calls the assignment operator for each field.
- For some classes, this is enough.

```
class Student {
  vector< int > grades;
public:
  string name;

  ...;
}
```



List Overloaded +

- I also overloaded + so it concatenates lists.

```
class List {
  friend List operator+( const List &a, const List &b );

  ...;
};

List operator+( const List &a, const List &b )
{
  ...;
}

List cList = aList + bList;
```

Need friendship to access List representation.

And here's the operator definition.

We can use it like this.

- Really, nothing new here, but I need it for the next topic.

Constructors and Type Conversion

- C++ has a special interpretation for one-parameter constructors.
  - We can use them to implicitly convert types.

```
class SomeClass {
  ...;
  SomeClass( int a ) {
    ...;
  }

  SomeClass( const string &a ) {
    ...;
  }
  ...;
}
```

How to build this object from an int.

How to build this object from a string.

List from Int

- Pretend we want to be able to make a List from an int.

```
class List {
  ...;
  List( int v )
  {
    head = new Node;
    head->val = v;
    head->next = NULL;
  }
  ...;
};
```

Here are the instructions.

No surprise, we can use this new constructor like this.

```
List bList = 25;
List cList = 16 + bList + 36 + 49;
```

Surprise! The compiler will also use it to be able to evaluate this expression.

Suppressing Value Semantics

- These special members help support value semantics, the ability to pass, return and assign objects by value.
- What if we don't want that? We didn't seem to need it in Java.
  - C++ has a syntax to suppress generation of these functions.

```
class List {
  ...;
  List( const List &other ) = delete;

  List &operator=( const List &other ) = delete;
  ...;
};
```

I don't want a copy constructor.

I don't want an assignment operator.

```
  List aList, bList;
  ...;
  List cList = aList;
  aList = bList;
```

Now, we can't do this.

Separating Implementation

- We need to be able to describe a class in a header file
- … while leaving the details to an implementation file.
- In the header, we can define the class, including:
  - All its fields
  - Prototypes for its methods

```
class List {
  struct Node {
    int val;
    Node *next;
  };
  Node *head;
public:
  List();
  List( const List & );
  ~List();
  ...;
}
```

Here's my representation.

I have these functions, but they're defined elsewhere.

Separating Implementation

- In the implementation file, we just need to define all the methods.

```
#include "List.h"
#include <iostream>

using namespace std;

List::List()
{
  head = NULL;
}

List::List( const List &other )
{
  Node **ptr = &head;

  ...;
```

Include my own header file first, a good policy.

I'm defining the default constructor that's part of the List class.

I'm defining the copy constructor that's part of the List class.

Exceptions in C++

- C++ supports exceptions, a general mechanism for handling errors.
- Exceptions are differentiated by type
  - Any type of value can be thrown as an exception.

```
int f( int val )
{
  if ( val <= 0 )
    throw 42;
  if ( val % 3 != 0 )
    throw "That value isn't divisible by 3";

  return val / 3;
}
```

Unlike Java, you don't have to advertise the exception types you may throw.

throw syntax, to exit this function abnormally with this value.

Exceptions in C++

- try/catch code to detect and respond to exceptions:

```
int val;
cout << "Enter an int: "
cin >> val;

try {
  int result = f( val );
  cout << result << endl;
} catch ( const char *str ) {
  cout << "Error: " << str << endl;
} catch ( int code ) {
  cout << "Internal Error: " << code << endl;
}
```

A try block to execute code that might throw an exception.

Respond to the exception, based on the type thrown.

Static Members

- In C++, you can have static member functions and fields.
  - It means the same thing as in Java: "I'm not part of an instance; I'm part of the whole class."

```
class C {
  static int x;
public:
  static void f() {
    ...
  }
}
```

There's just one of me, no matter how many instances of C you make.

You can call me without an instance.

Static Field Definition

- There's one detail. A static field is considered a declaration, not a definition.
- You need to define it in one of your implementation files.

```
class C {
  static int x;
public:
  static void f() {
    ...
  }
}
```

There's an x field, but it's not my job to allocate it.

```
static int C::x = 25;
```

That's my job.

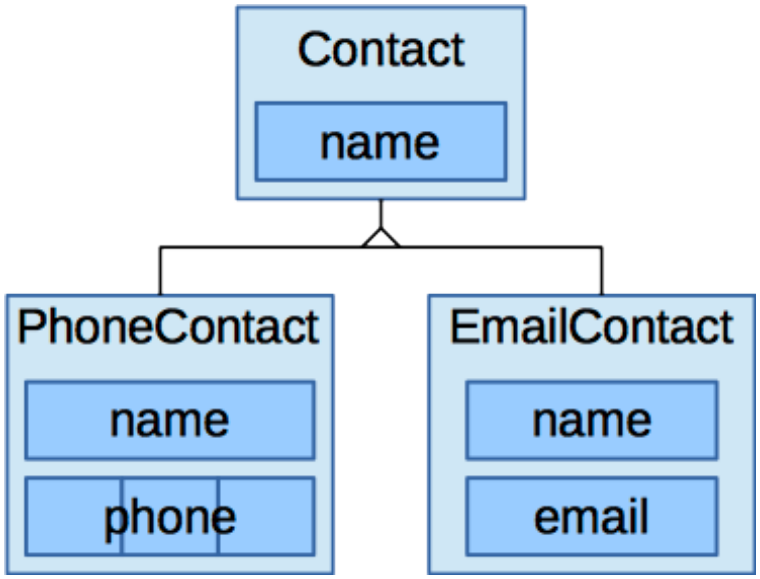
Inheritance

- Inheritance : we can define a class starting from an existing class.
  - The existing class is called the *base class*
  - The new class built from it is the *derived class*
- There are two reasons for doing this
  - Partial implementation : we can reuse any representation and behavior provided by the base class.
  - **Abstraction** : we can use the base class as a more general type, without committing to particular behavior implemented in the derived class

I think one of these is more important than the other. Can you tell which?

Contact Information

- We're going to use the contact information example from when the introduction to union
  - A base class containing just a name
  - Derived classes containing different kinds of contact information.



Inheritance Notation

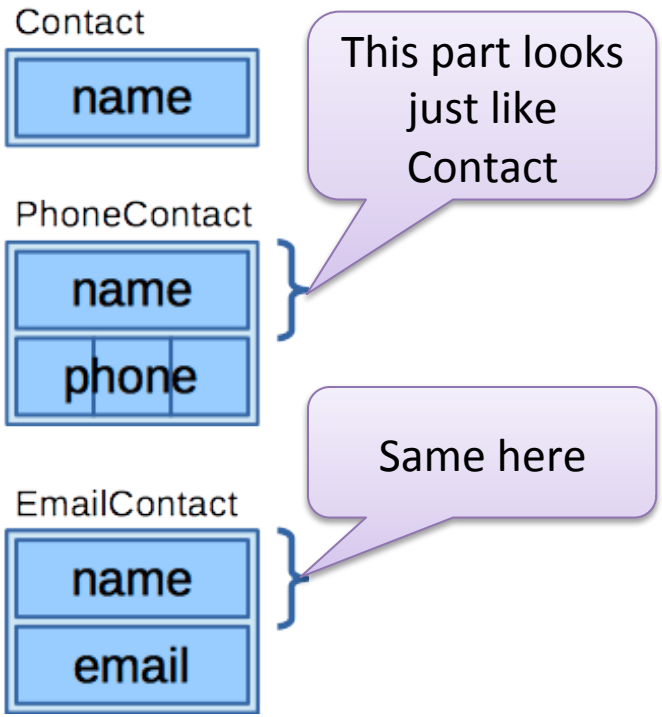
- C++ has a strange syntax for inheritance.

```
class Contact {
  char name[ 50 ];

  ...
};
```

I want everything from Contact, along with this other stuff.

```
class PhoneContact : public Contact {
  short phone[ 3 ];

  ...
};
```

```
class EmailContact : public Contact {
  char email[ 50 ];

  ...
};
```



Contact Class

```
class Contact {
  char name[ 50 ];

public:
  void setName( const char *nm )
  {
    strncpy( name, nm, sizeof( name ) - 1 );
    name[ sizeof( name ) - 1 ] = '\0';
  }

  const char *getName()
  {
    return name;
  }

  void printName()
  {
    cout << name << endl;
  }
};
```

What you'd expect, support for the contact name.

No constructor.
You could make an uninitialized Contact.
Will fix later.

PhoneContact Class

```
class PhoneContact : public Contact {
  short phone[ 3 ];

public:
  void setPhone( short a, short b, short c )
  {
    phone[ 0 ] = a;
    phone[ 1 ] = b;
    phone[ 2 ] = c;
  }

  void printPhone()
  {
    char buffer[ 20 ];
    snprintf( buffer, sizeof( buffer ), "%03d-%03d-%04d",
              phone[ 0 ], phone[ 1 ], phone[ 2 ] );
    cout << buffer << endl;
  }
};
```

We automatically get the name field and the three member functions from Contact.

EmailContact Class

```
class EmailContact : public Contact {
  char email[ 50 ];

public:
  void setEmail( const char *em )
  {
    strncpy( this->email, em, sizeof( email ) - 1 );
    email[ sizeof( email ) - 1 ] = '\0';
  }

  void printEmail()
  {
    cout << email << endl;
  }
};
```

Client Code

```
Contact ellen;
ellen.setName( "Ellen" );

PhoneContact bill;
bill.setName( "Bill" );
bill.setPhone( 919, 867, 5309 );

EmailContact mary;
mary.setName( "Mary" );
mary.setEmail( "mary@qmail.egg" );

ellen.printName();
cout << endl;

bill.printName();
bill.printPhone();
cout << endl;

mary.printName();
mary.printEmail();
cout << endl;
```

Ellen just has the functions from Contact.

Bill and Mary also have the functions from their derived classes.

What if you forget to initialize an object?

Inheritance and Constructors

- We can define constructors, to prevent creation of an uninitialized object.

```
class Contact {
  char name[ 50 ];

public:
  Contact( const char *nm )
  {
    strncpy( name, nm, sizeof( name ) - 1 );
    name[ sizeof( name ) - 1 ] = '\0';
  }
  ...
}
```

This will suppress creation of a default constructor.

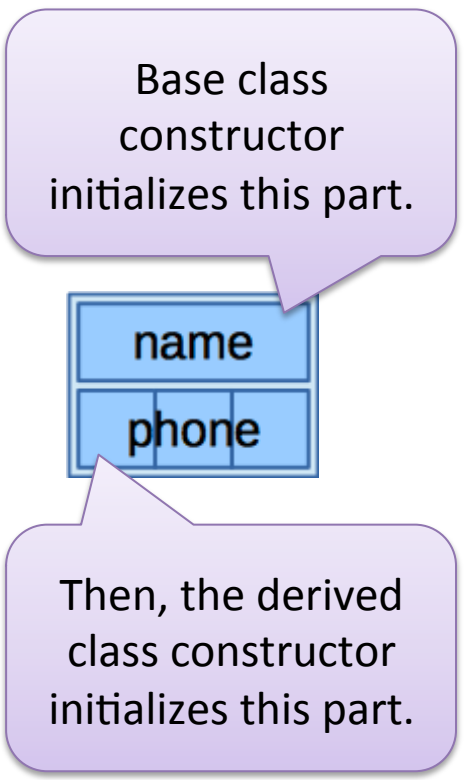
```
  Contact ellen( "Ellen" );
```

Now, client code has to supply parameters to make an object.

Inheritance and Constructors

- But, this will affect the derived classes.
  - They were using that default constructor.

```
class PhoneContact : public Contact {
  short phone[ 3 ];

public:
  PhoneContact( short a, short b, short c )
  {
    phone[ 0 ] = a;
    phone[ 1 ] = b;
    phone[ 2 ] = c;
  }

  ...
};
```



The Constructor Initializer List

- We had this problem in Java
  - Remember, we used the super() syntax to pass arguments to the base class constructor.
- Here, we use the *constructor initializer list*

```
class PhoneContact : public Contact {
  short phone[ 3 ];

public:
  PhoneContact( const char *nm, short a, short b, short c ) : Contact( nm )
  {
    phone[ 0 ] = a;
    phone[ 1 ] = b;
    phone[ 2 ] = c;
  }

  ...
};
```

Construct my base class part with this parameter.

The Constructor Initializer List

- There are times when you must use the constructor initializer list.
  - To call a non-default base class constructor
  - To initialize const or reference fields
  - To call constructors for fields that don't have default constructors.
- You can also use it to initialize other fields.

```
class SomeClass {
  const int a;
  Blob b;
  int &c;
  int x, y;

public:
  SomeClass( int v1, int v2, int v3, int x )
    : a( v1 ), b( v2 ), c( v3 ), x( x )
  {
    ...
  }
}
```

Imagine I don't have a default constructor.

You can initialize the x field like this.

Client Code Simplified

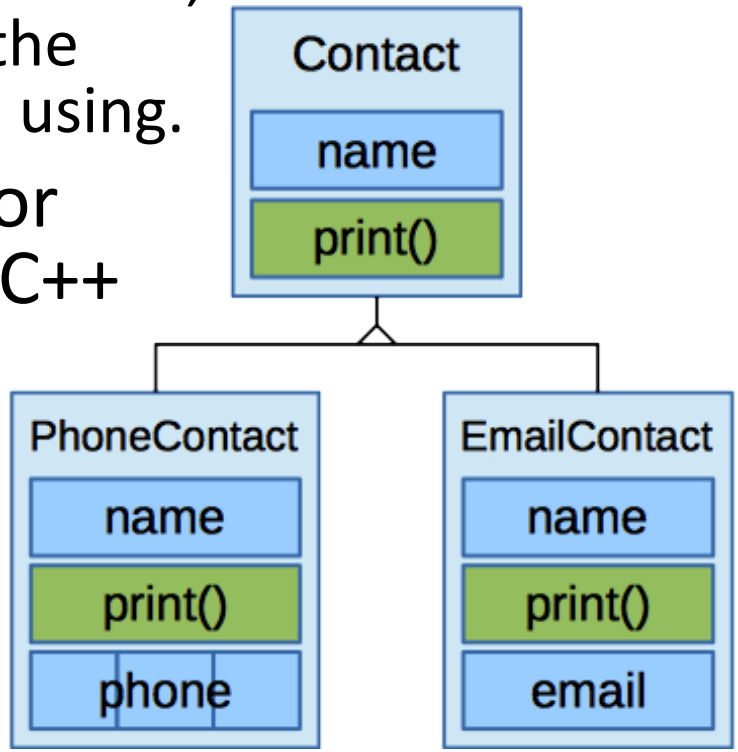
- Now, client code for our Contacts can't build uninitialized objects.

```
Contact ellen( "Ellen" );
PhoneContact bill( "Bill", 919, 867, 5309 );
EmailContact mary( "Mary", "mary@qmail.egg" );

ellen.printName();
cout << endl;

bill.printName();
bill.printPhone();
cout << endl;

mary.printName();
mary.printEmail();
cout << endl;
```

This is still kind of awkward. We have to know how to print each type of Contact.

Virtual Functions

- In C++, we can define virtual functions
  - Derived classes can override virtual functions defined in the base class
  - Then, when we call a virtual function, we get the version defined in the particular type of object we're using.
- This is the default behavior for methods in Java, but not for C++
  - Why? There's a little more overhead to call a virtual function.



Virtual Functions in Contact

- We can make sure all contacts know how to print themselves appropriately.

```
class Contact {
  ...
  virtual void print()
  {
    cout << name << endl;
  }
};
```

Derived classes can override this function.

```
class EmailContact : public Contact {
  ...
  void print() override
  {
    Contact::print();
    cout << email << endl;
  }
};
```

Here's my version of this function.

C++11, optional but a good idea.

Client Code More Simplified

- Now, client code doesn't have to worry about how to print each type of contact.

```
Contact ellen( "Ellen" );
PhoneContact bill( "Bill", 919, 867, 5309 );
EmailContact mary( "Mary", "mary@qmail.egg" );

ellen.print();
cout << endl;

bill.print();
cout << endl;

mary.print();
cout << endl;
```

All types of contacts know how to print themselves.

Consider, this first contact isn't very useful.

Abstract Classes

- The base class, Contact, isn't useful by itself; it doesn't actually store contact information.
- But, we still need it:
  - It provides shared implementation of the name field.
  - It specifies functionality all contacts should have.
- I wish we could keep Contact for inheritance but prevent client code from instantiating it.
- … and we can. We can make it an abstract class.
  - Abstract Class : a class you can't instantiate, but you can use its type to refer to its derived classes
  - Concrete Class : a class you can instantiate

Pure Virtual Functions

- In C++, we can make a class abstract by giving it a pure virtual function.
  - A virtual function with no default definition.
  - A function that has to be implemented in the derived class.

```
class Contact {
  ...
  virtual void print() = 0;
};
```

Here's how you mark a function as pure virtual.

```
class EmailContact : public Contact {
  ...
  void print() override
  {
    ...
  }
};
```

If you want to be able to instantiate this class, you must define this function.

Using Abstraction

- Here's a typical use of abstraction

```
vector< Contact * > contacts;

contacts.push_back( new PhoneContact( "Bill", 919,
                                      867, 5309 ) );
contacts.push_back( new EmailContact( "Mary",
                                      "mary@qmail.egg" ) );

contacts.push_back( new Contact( "Ellen" ) );

for ( Contact *c : contacts )
  c->print();

for ( Contact *c : contacts )
  delete c;
```

A list of pointers to contacts (of any type)

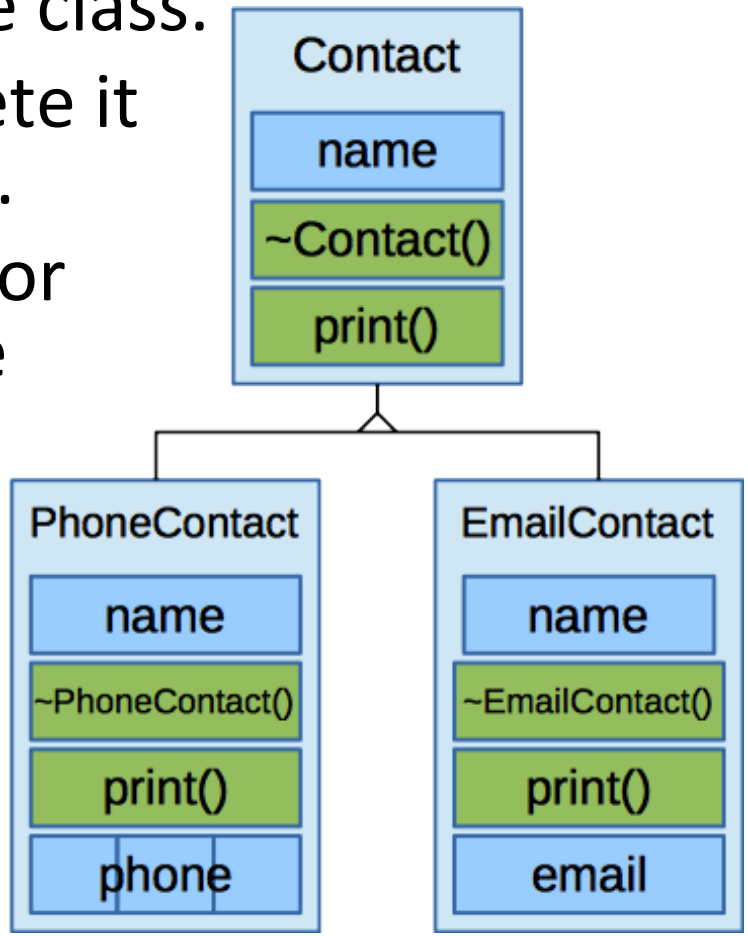
We can print them without worrying about their specific types.

But, you might have yourself a problem here. What distructor will you get?

Virtual Destructor

- If an object has virtual functions then you plan to use it via a pointer to its base class.
- Maybe you even plan to delete it via a pointer to its base class.
- You need to get the destructor for the particular type you're pointed to.
- You need a virtual destructor.

good band name?



Virtual Destructor

- I've changed the Contact implementation to highlight the need for a destructor.

```
class Contact {
  char *name;

public:
  Contact( const char *nm )
  {
    name = new char [ strlen( nm ) + 1 ];
    strcpy( name, nm );
  }

  virtual ~Contact()
  {
    delete [] name;
  }
  ...
};
```

Now the name is a pointer to a char array.

Allocated during construction.

Freed during destruction.

Let derived classes override this.

Virtual Destructor

- EmailContact also needs a non-trivial destructor:

```
class EmailContact : public Contact {
  char *email;

public:
  EmailContact( const char *nm, const char *em ) : Contact( nm )
  {
    email = new char [ strlen( em ) + 1 ];
    strcpy( email, em );
  }

  ~EmailContact() override
  {
    delete [] email;
  }
  ...
};
```

Here also, something to free.

So, client code will get this destructor.

Which automatically calls the base class destructor when its done.

Using a Virtual Destructor

- Now, this code will work:

```
vector< Contact * > contacts;

contacts.push_back( new PhoneContact( "Bill", 919,
                                      867, 5309 ) );
contacts.push_back( new EmailContact( "Mary",
                                      "mary@qmail.egg" ) );

for ( Contact *c : contacts )
  c->print();

for ( Contact *c : contacts )
  delete c;
```

We'll get the right destructor for the run-time type of this Contact.

Const Objects

- Consider, we never need to change a contact once we put it on our list.
- So, we could define our list like this:

```
vector< const Contact * > contacts;

contacts.push_back( new PhoneContact( "Bill", 919,
                                      867, 5309 ) );
contacts.push_back( new EmailContact( "Mary",
                                      "mary@qmail.egg" ) );

for ( Contact *c : contacts )
  c->print();

for ( Contact *c : contacts )
  delete c;
```

But now this code won't work.

Const Objects

- No problem, we just need to change our iteration syntax to use const.

```
vector< const Contact * > contacts;

contacts.push_back( new PhoneContact( "Bill", 919,
                                      867, 5309 ) );
contacts.push_back( new EmailContact( "Mary",
                                      "mary@qmail.egg" ) );

for ( const Contact *c : contacts )
  c->print();

for ( const Contact *c : contacts )
  delete c;
```

But now this call won't work. What if print changes the object it's called on?

Const Member Functions

- We need a way to say "This function doesn't change the object."
- And we have it, the *const member function*.

```
class EmailContact : public Contact {
  char *email;
public:
  ...
  void print() const override
  {
    // Print our name and our email.
    cout << getName() << " " << email << endl;
  }
};
```

I guess this should be const too.

Const-ness

- Const and const member functions are the C++ mechanism for immutable objects
  - Any variable can be const, effectively saying "You can't change me while I live."
  - But, then we can only call const member functions on the variable.

```
string x = "abc";
const string y = "xyz";

vector< int > aList = list;
const vector< int > bList = list;
```

You can modify me.

But not me.

Same here.