```
------------------------------------------------------------------
pizza.cpp: simple inheritance example
------------------------------------------------------------------

#include <...>
using namespace std;

class pizza {
  public:
    pizza();
    virtual ~pizza();

    virtual void print_type()=0;
    virtual void print_ingredients();
    virtual void print_newline();

  protected:
    string *cheese;
    string *toppings;
};

pizza::pizza() {
  cheese = NULL;
  toppings = NULL;
}

pizza::~pizza() {
  if (cheese)   delete cheese;
  if (toppings) delete toppings;
}

void pizza::print_ingredients() {
  if (cheese)   cout << " " << *cheese;
  if (toppings) cout << " " << *toppings;
}

void pizza::print_newline() {
  cout << "\n";
}

------------------------------------------------------------------

class cheese_pizza : public pizza {
  public:
    cheese_pizza();
    ~cheese_pizza() { ; }

    void print_type();
};

cheese_pizza::cheese_pizza() {
  cheese = new string("mozzarella");
  toppings = NULL;
}

void cheese_pizza::print_type() {
  cout << setw(18) << "cheese pizza:";
}

------------------------------------------------------------------

class special_pizza : public pizza {
  public:
    special_pizza(pizza *);
    virtual ~special_pizza();

    void print_type() { W->print_type(); }
    void print_ingredients();

  protected:
    string *addon;
    pizza *W;
};

special_pizza::special_pizza(pizza *n_W) {
  addon = NULL;
  W = n_W;
}

special_pizza::~special_pizza() {
  if (addon) delete addon;
  delete W;
}

void special_pizza::print_ingredients() {
  W->print_ingredients();
  if (addon) cout << " " << *addon;
}

------------------------------------------------------------------

struct addon_pesto : public special_pizza {
  addon_pesto(pizza *);
  ~addon_pesto() { ; }
};

addon_pesto::addon_pesto(pizza *n_W) : special_pizza(n_W) {
  addon = new string("pesto");
}
```

```
int main() {
  srand(time(NULL));

  const int N = 20;      // number of orders
  const int NPT = 1;     // number of pizza types
  const int NAT = 1;     // number of addon types

  pizza *thepie;
  queue<pizza *> orders;

  for (int i=0; i<N; i++) {
    int pizza_type = rand() % NPT;
    switch (pizza_type) {
    case 0:
      thepie = new cheese_pizza;
      break;
    }

    int special_type = rand() % (NAT+1);
    switch (special_type) {
    case 0:
      orders.push(thepie);
      break;
    case 1:
      orders.push(new addon_pesto(thepie));
      break;
    }
  }

  while (!orders.empty()) {
    thepie = orders.front();
    orders.pop();

    thepie->print_type();
    thepie->print_ingredients();

    int price = 8;
    if (dynamic_cast<special_pizza *>(thepie)) {
      if (dynamic_cast<addon_pesto *>(thepie))
        price += 1;
    }
    cout << " $" << price;

    thepie->print_newline();

    delete thepie;
  }
}
```

---

TODO:

Make a pepperoni_pizza class based on the cheese_pizza class. Change the cheese and create the topping as pepperoni.

Make an addon_olive class based on the addon_pesto class.

Update the main program to handle pepperoni_pizza objects and addon_olive objects incl switch-cases and price handling.

```
unix> ./pizza
  pepperoni pizza: provolone pepperoni olive $10
  pepperoni pizza: provolone pepperoni pesto $9
  pepperoni pizza: provolone pepperoni olive $10
  pepperoni pizza: provolone pepperoni $8
  pepperoni pizza: provolone pepperoni pesto $9
  pepperoni pizza: provolone pepperoni $8
     cheese pizza: mozzarella $8
     cheese pizza: mozzarella pesto $9
     cheese pizza: mozzarella $8
  pepperoni pizza: provolone pepperoni pesto $9
     cheese pizza: mozzarella $8
  pepperoni pizza: provolone pepperoni olive $10
  pepperoni pizza: provolone pepperoni $8
  pepperoni pizza: provolone pepperoni olive $10
  pepperoni pizza: provolone pepperoni $8
     cheese pizza: mozzarella olive $10
     cheese pizza: mozzarella pesto $9
     cheese pizza: mozzarella pesto $9
  pepperoni pizza: provolone pepperoni $8
     cheese pizza: mozzarella olive $10
```

---

```
-----------------------------------------------------------------
factory_method.cpp: Factory (store) generates products (pizzas)
using a member function (store::order).
-----------------------------------------------------------------

#include <...>
using namespace std;

-----------------------------PRODUCT-----------------------------

class pizza {
  public:
    pizza() { ; }
    virtual ~pizza() { ; }

    void print_name() { cout << pizza_name; }
    void print_newline() { cout << "\n"; }

  protected:
    string pizza_name;
};

-----------------------------------------------------------------

struct cheese_pizza1 : public pizza {
  cheese_pizza1() { pizza_name = "cheese pizza 1"; }
};

struct pepperoni_pizza1 : public pizza {
  pepperoni_pizza1() { pizza_name = "pepperoni pizza 1"; }
};

-----------------------------FACTORY-----------------------------

class store {
  public:
    store() { ; }
    virtual ~store() { ; }

    void print_name();

    pizza *order(const string &type) { return product(type); }

  protected:
    virtual pizza *product(const string &)=0;

    string store_name;
};
```

```
void store::print_name() {
  cout << setw(12)
       << setfill('.')
       << left
       << store_name
       << ": "
       << setfill(' ');
}

-----------------------------------------------------------------

struct store_Knoxville : public store {
  store_Knoxville() { store_name = "Knoxville"; }

  pizza *product(const string &);
};

pizza *store_Knoxville::product(const string &type) {
  pizza *thepie = NULL;

  if (type == "cheese")
    thepie = new cheese_pizza1();
  else
  if (type == "pepperoni")
    thepie = new pepperoni_pizza1();

  return thepie;
}

-----------------------------------------------------------------

TODO:

Add more products, e.g., cheese_pizza2, pepperoni_pizza2, and
more stores, e.g., store_OakRidge. Update the main program on
the next page to include the extra store. Try changing what
store makes what pizzas.

-----------------------------------------------------------------

Hint: Once a collection of products exist, it is easy to change
which products a factory makes. In order words, we can modify
the types of pizza sold by a store without having to change any
other code.

-----------------------------------------------------------------
```

```
struct info {
  info(int, pizza *);

  int store_number;      // who made pizza
  pizza *thepie;         // what type was it
};

info::info(int n_storenumber, pizza *n_thepie) {
  store_number = n_storenumber;
  thepie = n_thepie;
}

int main() {
  srand(time(NULL));

  int N = 10;     // number of orders
  int NS = 2;     // number of stores
  int NP = 2;     // number of pizza types

  store *storelist[NP];

  storelist[0] = new store_Knoxville();
  storelist[1] = new store_Knoxville();

  queue<info> orders;

  for (int k=0; k<N; k++) {
    int store_number = rand() % NS;
    store *thestore = storelist[store_number];

    int pizza_type = rand() % NP;
    pizza *thepie = NULL;

    switch (pizza_type) {
    case 0:
      thepie = thestore->order("cheese");
      break;
    case 1:
      thepie = thestore->order("pepperoni");
      break;
    }

    if (thepie)
      orders.push(info(store_number, thepie));
  }

  while (!orders.empty()) {
    info next_order = orders.front();
    orders.pop();

    int store_number = next_order.store_number;
    store *thestore = storelist[store_number];

    pizza *thepie = next_order.thepie;

    thestore->print_name();
    thepie->print_name();
    thepie->print_newline();

    delete thepie;
  }

  for (int i=0; i<NS; i++)
    delete storelist[i];
}
```

---

Hint: The store list can hold a mix of stores as long as they are all derived from the same base store class.

Hint: The order queue has no knowledge of stores and products the sell. All products look the same to it as long as they are all derived from the same base product class.

Hint: Note that main program doesn't know what the objects represent. It merely executes functions associated with objects

```
unix> ../factory_method
Knoxville...: pepperoni pizza 1
OakRidge....: cheese pizza 2
Knoxville...: cheese pizza 1
Knoxville...: pepperoni pizza 1
OakRidge....: cheese pizza 2
Knoxville...: pepperoni pizza 1
OakRidge....: cheese pizza 2
Knoxville...: cheese pizza 1
OakRidge....: cheese pizza 2
Knoxville...: pepperoni pizza 1
```

```
------------------------------------------------------------------
abstract_factory.cpp: Control the types of products produced by the
factories using recipes.
------------------------------------------------------------------

#include <...>
using namespace std;

-----------------------------PRODUCT--------------------------

class pizza {
  public:
    pizza() { ; }
    virtual ~pizza() { ; }

    void print_name() { cout << pizza_name; }
    void print_newline() { cout << "\n"; }

  protected:
    string pizza_name;
};

-----------------------------RECIPE---------------------------

class recipe {
  public:
    recipe() { ; }
    virtual ~recipe() { ; }

    virtual string get_recipe()=0;
};

struct recipe1 : public recipe {
  string get_recipe() { return " 1"; }
};

-----------------------------PRODUCT--------------------------

struct cheese_pizza : public pizza {
  cheese_pizza(recipe *R) {
    pizza_name = "cheese pizza" + R->get_recipe();
  }
};

struct pepperoni_pizza : public pizza {
  pepperoni_pizza(recipe *R) {
    pizza_name = "pepperoni pizza" + R->get_recipe();
  }
};
```

```
-----------------------------FACTORY--------------------------

class store {
  public:
    store() { ; }
    virtual ~store() { ; }

    void print_name();

    pizza *order(const string &type) { return product(type); }

  protected:
    virtual pizza *product(const string &)=0;

    recipe *store_recipe;
    string store_name;
};

void store::print_name() { ... }

struct store_Knoxville : public store {
  store_Knoxville() {
    store_recipe = new recipe1();
    store_name = "Knoxville";
  }

  pizza *product(const string &);
};

pizza *store_Knoxville::product(const string &type) {
  pizza *thepie = NULL;

  if (type == "cheese")
    thepie = new cheese_pizza(store_recipe);
  else
  if (type == "pepperoni")
    thepie = new pepperoni_pizza(store_recipe);

  return thepie;
}

------------------------------------------------------------------

TODO:

Add more recipes, products and stores. Have different stores use
different recipes.

------------------------------------------------------------------
```

```
See factory_method.cpp for the driver code

struct info { ... }

main() { ... }

-----------------------------------------------------------------

Example:

After making the Knoxville store use recipe1, adding a recipe2
and an Oak Ridge store that uses it, and making minor modifications
to the main function that ties the code together, we have this.

unix> ../abstract_factory
Knoxville...: cheese pizza 1
OakRidge....: pepperoni pizza 2
Knoxville...: cheese pizza 1
Knoxville...: cheese pizza 1
OakRidge....: cheese pizza 2
OakRidge....: pepperoni pizza 2
Knoxville...: cheese pizza 1
Knoxville...: pepperoni pizza 1
Knoxville...: pepperoni pizza 1
Knoxville...: cheese pizza 1

-----------------------------------------------------------------

Hint: See how easy it is to change products by changing recipes
without having to change any other code. See how easy it is to
add more factories and modify which recipes they use to create
their products. See how the main function is completely agnostic
about such changes.

-----------------------------------------------------------------

TODO:

Modify the code to allow the main function to control which recipe
a store uses when it's instantiated. That is, pass an argument to
the store constructor when then makes a recipe decision accordingly.

-----------------------------------------------------------------
```