Opgave 1

a) Nee. De methode **`bestaat()`** is *pure virtual*. (Dat zie je aan de **`= 0`** achter de declaratie.) De klasse is dus abstract.
Je kunt het mogelijk maken door de **`= 0`** weg te halen en een implementatie te schrijven van de methode.

b) Een *namespace* is een container die *identifiers* en *symbolen* bevat; **`namespace std`** bevat onder andere de definities van **`vector`**, **`cout`** en **`endl`**.
De compiler zal deze dus niet herkennen als **`using namespace std`** ontbreekt.

c) Omdat **`kas`** een *pointer* is.

d) Alles waar **`new`** voor staat, moet ergens een **`delete`** hebben. Er is dus een geheugenlek voor het object **`kas`**, en voor de twee laatste objecten die aan de **`kas`** worden toegevoegd. (Eén object van de klasse **`Euro`** en één object van de klasse **`Gulden`**.)

e)

```
void Kassa::stort(Munt* muntje)
{
	inhoud.push_back(muntje);
}
```

f)

```
int Kassa::aantalMunten()
{
	return inhoud.size();
}
```

g)

```
int Kassa::aantalGeldig()
{
	int geldig = 0;
	for (unsigned int i = 0; i < inhoud.size(); i++){
		if (((Munt*)inhoud[i])->bestaat())
		{
			geldig++;
		}
	}
	return geldig;
}
```

h) Die bepaalt dat de methode *pure virtual* is. De klasse is daardoor abstract.

i)

```
void Kassa::stort(Munt* muntje)
{
	muntje->waarde = 0;
	inhoud.push_back(muntje);
}
```

Opgave 2

a) Nee. Dat zou wel nodig zijn als de attributen pointers waren naar objecten van (subklassen van) Gerecht. Maar in dit geval is het zo dat de objecten worden gecreëerd door aanroep van hun default constructor.

b) Als soep een pointer naar een object van de klasse Voorgerecht is, dan zou die ook naar NULL kunnen wijzen.

c) Als de objecten soep, dagschotel en ijs worden vervangen door *vectoren* met pointers naar die objecten, dan kan elk van die vectoren 0 of meer objecten bevatten.

d) Let op: het attribuut prijs is protected. Dat kun je dus niet benaderen vanuit Diner. Een public methode getPrijs() is nodig op het private attribuut te benaderen.
Syntactisch gezien kun je ook het attribuut public maken, maar dat is geen correct gebruik van het OO-principe *inkapseling*.

```
public:
double getPrijs() {return prijs;};
};

double Diner::bepaalDinerPrijs(){
      double prijs = 0;

      prijs += soep.getPrijs();
      prijs += dagschotel.getPrijs();
      prijs += ijs.getPrijs();

return 0;
};
```

Opgave 3

a) Omdat std::cout wordt gebruikt, en die is gedefinieerd in iostream.

b) order[i].aantal mag niet worden aangeroepen omdat aantal een private attribuut is.
Je zou het attribuut public kunnen maken (niet zo netjes) of vervangen door een methode die de waarde van het attribuut retourneert.

c) Nee, dat is niet fout.
De precompiler vervangt de code door wat er achter de #define staat.
Hierdoor wordt bijvoorbeeld het statement

```
Bestelling order[ORDERGROOTTE];
```

veranderd is

```
Bestelling order[5;];
```

Dat levert een compiler error op.

d) Nee, het programma bevat geen geheugenlekken.
Er wordt nergens geheugen dynamisch aangemaakt.
Statisch geheugen

e) zie code hieronder
- er komt een * achter Bestelling bij de declaratie
- er is een new nodig om het geheugen te creëren
- er komen -> i.p.v. .
- er is een delete nodig om het geheugen vrij te geven

```
Bestelling* order[ORDERGROOTTE];
for (int i = 0; i < ORDERGROOTTE; i++)
{
	order[i] = new Bestelling();
	// Voer de orders in
	order[i]->prijs = PRIJS;
	order[i]->aantal = i + 1;
}
for (int i = 0; i < ORDERGROOTTE; i++)
{
	// Voer de orders in
	cout << order[i]->prijs << " " << order[i]->aantal << endl;
}
for (int i = 0; i < ORDERGROOTTE; i++)
{
	delete order[i];
}
```