CSE 1325

Week of 10/12/2020

Instructor : Donna French

Constructors

A **constructor** is a special kind of class member function that is automatically called when an object of that class is instantiated.

Constructors are typically used to initialize member variables of the class to appropriate default or user-provided values, or to do any setup steps necessary for the class to be used.

Unlike normal member functions, constructors have specific rules for how they must be named:

- Constructors must have the same name as the class (with the same capitalization)

Initializing Objects with Constructors

```
My bank account's name is
Enter a new name for the bank account CSE 1325's Bank Account
My bank account has been renamed CSE 1325's Bank Account
```

Our initial bank account name prints out as blank because, when our object `MyBankAccount` was created, `name` was initialized to the empty string.

What if you want the freshly created object to have a name already?

A class can define a **constructor** that specifies *custom initialization* for objects of that class.

Initializing Objects with Constructors

Constructor

special member function

must have the same name as the class

have no return type (not even void)

C++ requires a constructor call when *each* object is created

ideal point in program to initialize an object's data members

can have parameters

A constructor's *parameter list* specifies pieces of data required to initialize an object

usually public

Default Constructors

A constructor that takes no parameters (or has parameters that all have default values) is called a default constructor.

The default constructor is called if no user-provided initialization values are provided.

Default Constructor

```
Account MyBankAccount;
```

No braces to the right of the object's name causes the default constructor to be called.

In any class that does *not* explicitly define a constructor, the compiler provides a default constructor with no parameters.

```
class Date
{
  public :
     std::string getDateMMDDCCYY(void)
     {
          std::string s_month{std::to_string(month)};
          std::string s_day{std::to_string(day)};
          std::string s_year{std::to_string(month)};
          return (s_month.size() == 1 ? "0" : "") + s_month +
                 (s_day.size() == 1 ? "0" : "") + s_day +
                  s_year;
     }

  private :
     int month;
     int day;
     int year;
};
```

```
int main(void)
{
   Date today;

   cout << today.getDateMMDDCCYY();

   return 0;
}
```

Default constructor with no parameters created by compiler

```
student@cse1325:/media/sf_VM$ ./constructorDemo.e
327641473398668327 64student@cse1325:/media/sf_VM$
```

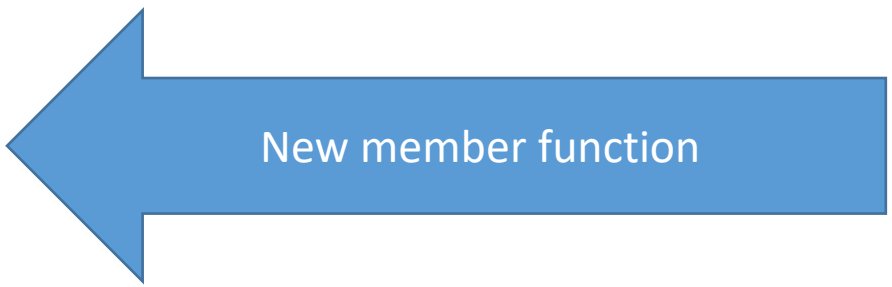
The default constructor does *not* initialize the class's fundamental-type data members, but *does* call the default constructor for each data member that's an object of another class.

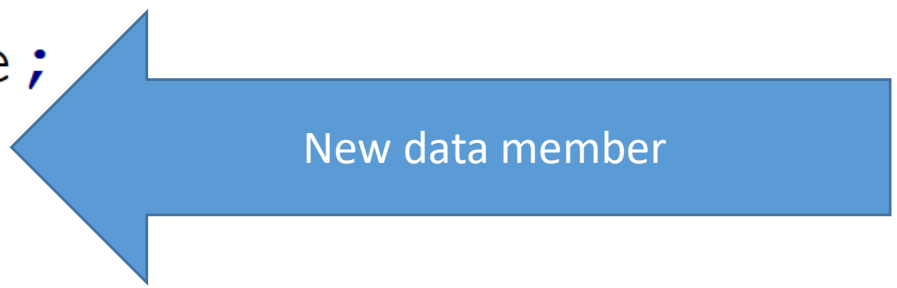
```
5  class Account
6  {
7      public :
8          std::string getName() const
9          {
10             return name;
11         }
12
13     private :
14         std::string name;
15 };
16
17 using namespace std;
18
19 int main(void)
20 {
21     Account MyAccount;
22
23     cout << "My account's name is \"" << MyAccount.getName() << "\"" << endl;
24
25     return 0;
26 }
```

The default constructor does *not* initialize the class's fundamental-type data members, but *does* call the default constructor for each data member that's an object of another class.

```
student@cse1325:/media/sf_VM$ ./Account1Demo.e
My account's name is ""
```

```
class Account
{
    public :
        std::string getName() const
        {
            return name;
        }
        int getBalance() const
        {
            return balance;
        }

    private :
        std::string name;
        int balance;
};
```





```
int main(void)
{
    Account MyAccount;

    cout << "My account's name is \"" << MyAccount.getName() << "\""
         << " and the balance is " << MyAccount.getBalance() << endl;

    return 0;
}
```

```
student@cse1325:/media/sf_VM$ ./Account1Demo.e
My account's name is "" and the balance is 844865152
```

The default constructor does *not* initialize the class's fundamental-type data members, but *does* call the default constructor for each data member that's an object of another class.

```
class Account
{
    public :
        Account()
        {
            balance = 0;
        }
        std::string getName()  const
        {
            return name;
        }
        int getBalance() const
        {
            return balance;
        }

    private :
        std::string name;
        int balance;
};
```

We provided a default constructor so the compiler does not create one for us.

```
student@cse1325:/media/sf_VM$ ./Account1Demo.e
My account's name is "" and the balance is 0
```

```
class Account
{
    public :
        Account()
        {
            balance = -100;
            name = "MyPaycheck";
        }
        std::string getName() const
        {
            return name;
        }
        int getBalance() const
        {
            return balance;
        }

    private :
        std::string name;
        int balance;
};
```

```
student@cse1325:/media/sf_VM$ ./Account1Demo.e
My account's name is "MyPaycheck" and the balance is -100
```

```
class Account
{
    public :
        Account()
        {
            balance = -100;
            name = "MyPaycheck";
        }
        Account(int startingBalance, std::string newName="Bank Account")
        {
            balance = startingBalance;
            name = newName;
        }
```

```
Account MyAccount;
Account YourAccount{1000};
Account HisAccount{1,"His Account"};
```

```
class Account
{
    public :
        Account()
        {
            balance = -100;
            name = "MyPaycheck";
        }
        Account(int startingBalance=0, std::string newName="Bank Account")
        {
            balance = startingBalance;
            name = newName;
        }
```

What happens if we try to give the balance parameter a default when the name already has a default?

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 Account1Demo.cpp -o Account1Demo.o
Account1Demo.cpp: In function ‘int main()’:
Account1Demo.cpp:36:10: error: call of overloaded ‘Account()’ is ambig
uous
  Account MyAccount;
          ^~~~~~~~~
Account1Demo.cpp:13:3: note: candidate: Account::Account(int, std::__c
xx11::string)
   Account(int startingBalance=0, std::string newName="Bank Account")
   ^~~~~~~
Account1Demo.cpp:8:3: note: candidate: Account::Account()
   Account()
   ^~~~~~~
makefile:14: recipe for target 'Account1Demo.o' failed
make: *** [Account1Demo.o] Error 1
```

```
class Account
{
  public :
    Account(std::string accountName)
      : name{accountName}
    {
    }

    void setName(std::string accountName)
    {
      name = accountName;
    }

    std::string getName() const
    {
      return name;
    }
  private :
    std::string name;
};
```

constructor has one `string` parameter called `accountName`

member-initializer list

Member initializers appear between a constructor's parameter list and the left brace that begins the constructor's body.

The member initializer list is separated from the parameter list with a colon (:).

Each member initializer consists of a data member's *variable name* followed by {} containing the member's *initial value*.

`name` is initialized with the parameter `AccountName`'s value.

If a class contains more than one data member, each member initializer is separated from the next by a comma.

The member initializer list executes before the constructor's body executes.

Member Initializer List

Variables in the initializer list are not initialized in the order that they are specified in the initializer list.

They are initialized in the order in which they are declared in the class.

For best results, the following recommendations should be observed:

1) Don’t initialize member variables in such a way that they are dependent upon other member variables being initialized first (in other words, ensure your member variables will properly initialize even if the initialization ordering is different).

2) 2) Initialize variables in the initializer list in the same order in which they are declared in your class. This isn’t strictly required so long as the prior recommendation has been followed, but your compiler may give you a warning if you don’t do so and you have all warnings turned on.

```
class Account
{
  public :
    Account(std::string accountName)
      : name{accountName}
    {
    }

    void setName(std::string accountName)
    {
      name = accountName;
    }

    std::string getName() const
    {
      return name;
    }
  private :
    std::string name;
};
```

The constructor and member function `setName()` both have a parameter called `accountName`.

Both of them are local to their functions.

Their scope means that they are not visible to each other.

Default Constructor

In class `Account`, the class's default constructor calls class string's default constructor to initialize the data member name to the empty string. An uninitialized fundamental-type variable contains an undefined ("garbage") value.

If you define a custom constructor for a class, the compiler will *not* create a default constructor for that class.

Once we created the constructor, we can no longer use the default constructor.

```
g++ -c -g -std=c++11 TestBankAccount.cpp -o TestBankAccount.o
TestBankAccount.cpp: In function 'int main()':
TestBankAccount.cpp:14:10: error: no matching function for call to 'Account::Acc
ount()'
  Account MyBankAccount;
```

```
student@cse1325:/media/sf_VM$ more makefile
#Donna French
#makefile for C++ program
SRC = TestBankAccount.cpp
OBJ = $(SRC:.cpp=.o)
EXE = $(SRC:.cpp=.e)

CFLAGS = -g -std=c++11

all : $(EXE)

$(EXE): $(OBJ)
        g++ $(CFLAGS) $(OBJ) -o $(EXE)

$(OBJ) : $(SRC)
        g++ -c $(CFLAGS) $(SRC) -o $(OBJ)

student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 TestBankAccount.cpp -o TestBankAccount.o
g++ -g -std=c++11 TestBankAccount.o -o TestBankAccount.e
student@cse1325:/media/sf_VM$ ./TestBankAccount.e
My bank account's name is Bank Account with Lots of Money
Enter a new name for the bank account My Bank Account
My bank account has been renamed My Bank Account
student@cse1325:/media/sf_VM$
```

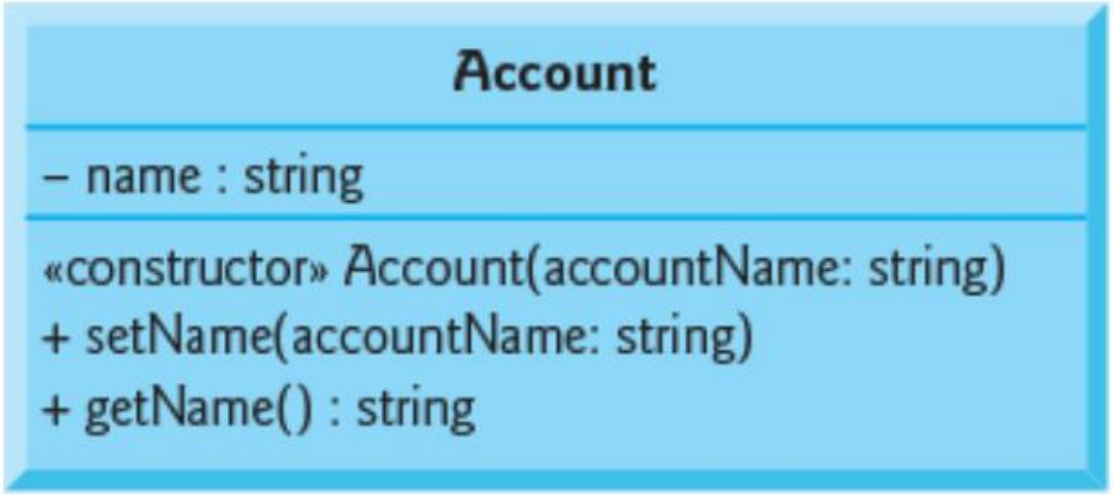
Class Diagram with a Constructor

Like operations, class diagrams show constructors in the *third* compartment of a class diagram.

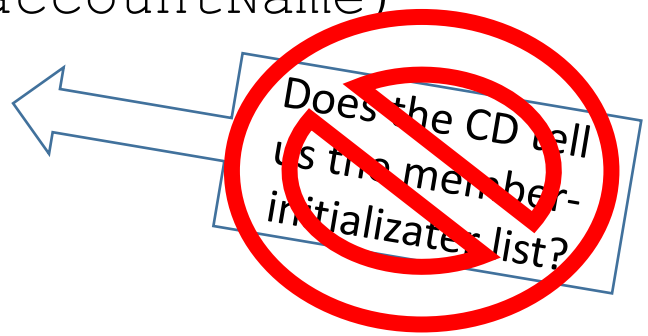
To distinguish a constructor from the class's operations, the class diagram requires that the word "constructor" be enclosed in **≪ and ≫** and placed before the constructor's name. It's customary to list

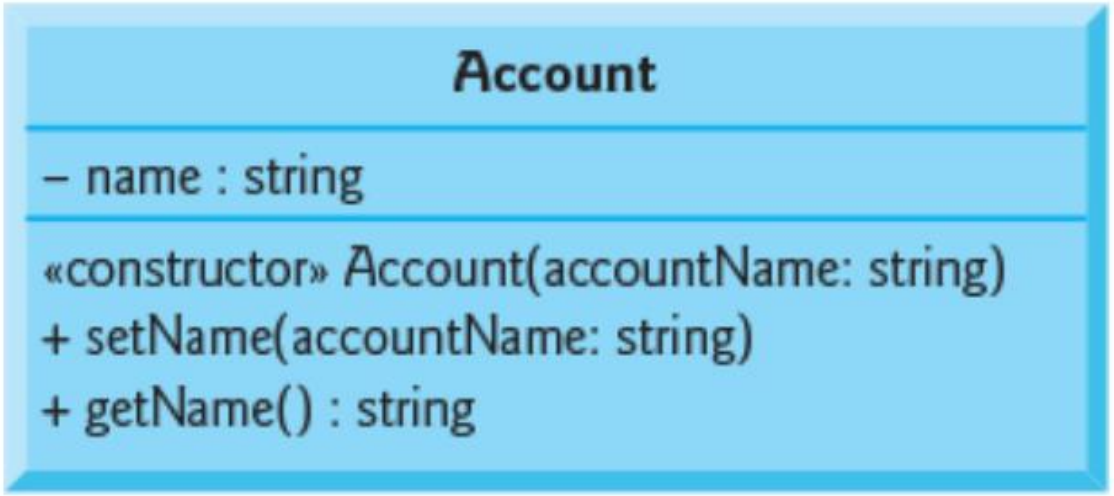
constructors *before* other operations

in the third compartment.



```
class Account
{
  public :
    Account(std::string accountName)
      : name{accountName}
    {
    }

    void setName(std::string accountName)
    {
    }

    std::string getName()
    {
    }
  private :
    std::string name;
};
```



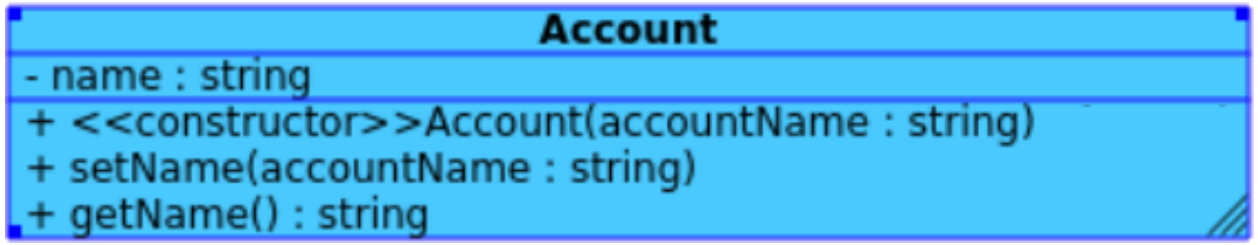


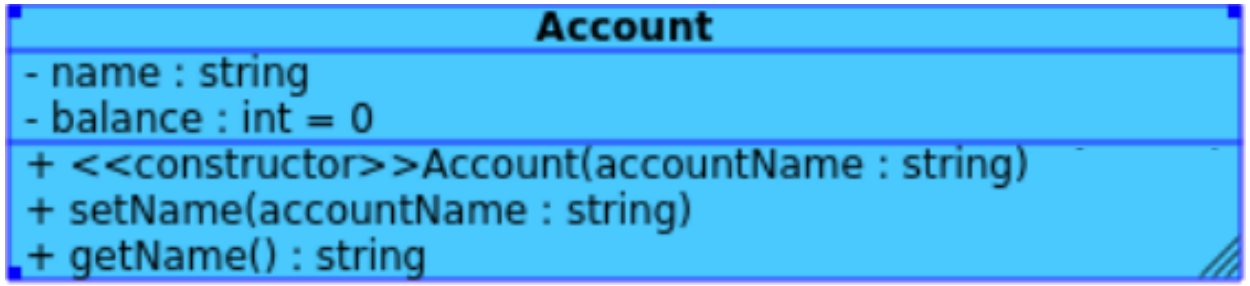
```
class Account
{
  public :
    Account(std::string accountName)
      : name{accountName}
    {
    }

    void setName(std::string accountName)
    {
      name = accountName;
    }

    std::string getName() const
    {
      return name;
    }
  private :
    std::string name;
};
```

```
class Account
{
  public :
    Account(std::string accountName)
      : name{accountName}
    {
    }

    void setName(std::string accountName)
    {
      name = accountName;
    }

    std::string getName() const
    {
      return name;
    }
  private :
    std::string name;
    int balance{0};
};
```

```
class Account
{
  public :
    Account(std::string accountName)
      : name{accountName}
    {
    }

    void setName(std::string accountName)
    {
      name = accountName;
    }

    std::string getName() const
    {
      return name;
    }
  private :
    std::string name;
    int balance{0};
};
```

`balance` is now a variable available to any member function of class `Account`





```
class Account
{
  public :
    Account(std::string accountName, int InitialBalance)
      : name{AccountName}
    {
      if (InitialBalance > 0)
        balance = InitialBalance;
    }

    void setName(std::string accountName)
    {
      name = accountName;
    }

    std::string getName() const
    {
      return name;
    }
  private :
    std::string name;
    int balance{0};
};
```

Constructor validates that the passed in `InitialBalance` is greater than zero. If it is, the account's balance to the passed in balance;

otherwise, it stays 0 – the default initial value set in the class's definition.

Account.h

```
#include <iostream>
#include <string>

#include "Account.h"

using namespace std;

int main()
{
  string NewAccountName;

  Account MyBankAccount{"Bank Account with Lots of Money", 1000000};

  cout << "My bank account's name is " << MyBankAccount.getName();

  cout << "\nEnter a new name for the bank account ";
  getline(cin, NewAccountName);
  MyBankAccount.setName(NewAccountName);

  cout << "My bank account has been renamed " << MyBankAccount.getName() << endl;

  return 0;
}
```

We started with the default constructor.

We then created our own constructor with one parameter to set `name` to the passed in value.

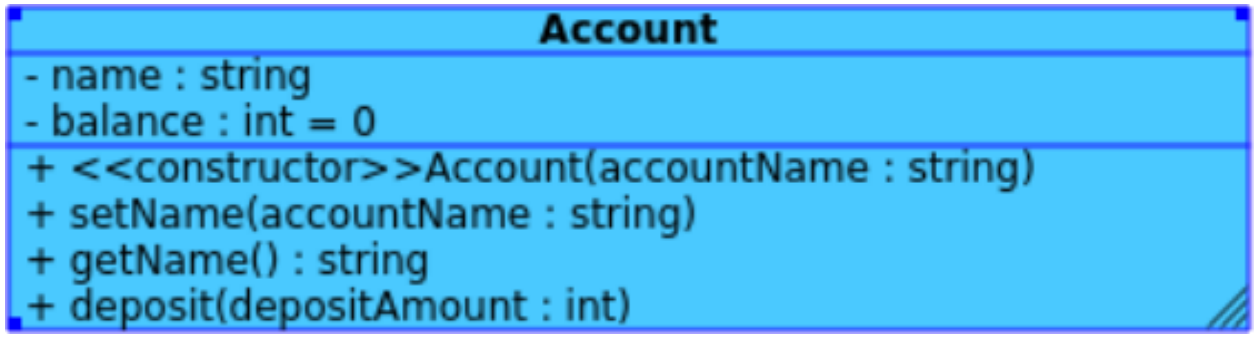
Then, we add a passed in `balance` to the constructor.

TestBankAccount.cpp

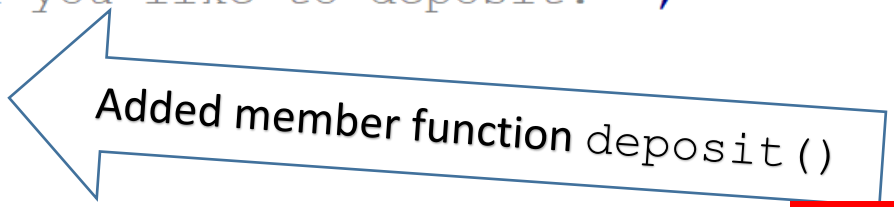
Adding a new member function- deposit

```
3  class Account
4  {
5      public :
6          Account(std::string AccountName, int InitialBalance)
7              : name{AccountName}
8          {
9              if (InitialBalance > 0)
10                 balance = InitialBalance;
11         }
12
13         void setName(std::string AccountName)
14         {
15             name = AccountName;
16         }
17
18         std::string getName() const
19         {
20             return name;
21         }
22
23         void deposit(int depositAmount)
24         {
25             if (depositAmount > 0)
26             {
27                 balance += depositAmount;
28             }
29         }
30
31     private :
32         std::string name;
33         int balance{0};
34 };
```

```
void deposit(int depositAmount)
{
    if (depositAmount > 0)
    {
        balance += depositAmount;
    }
}
```



```cpp
#include <iostream>
#include <string>

#include "Account.h"

using namespace std;

int main()
{
    string NewAccountName;
    int deposit;

    Account MyBankAccount{"Bank Account with Lots of Money", 1000000};

    cout << "My bank account's name is " << MyBankAccount.getName();

    cout << "\nEnter a new name for the bank account ";
    getline(cin, NewAccountName);
    MyBankAccount.setName(NewAccountName);

    cout << "My bank account has been renamed " << MyBankAccount.getName() << endl;

    cout << "How many dollars would you like to deposit? ";
    cin >> deposit;
    MyBankAccount.deposit(deposit);

    return 0;
}
```



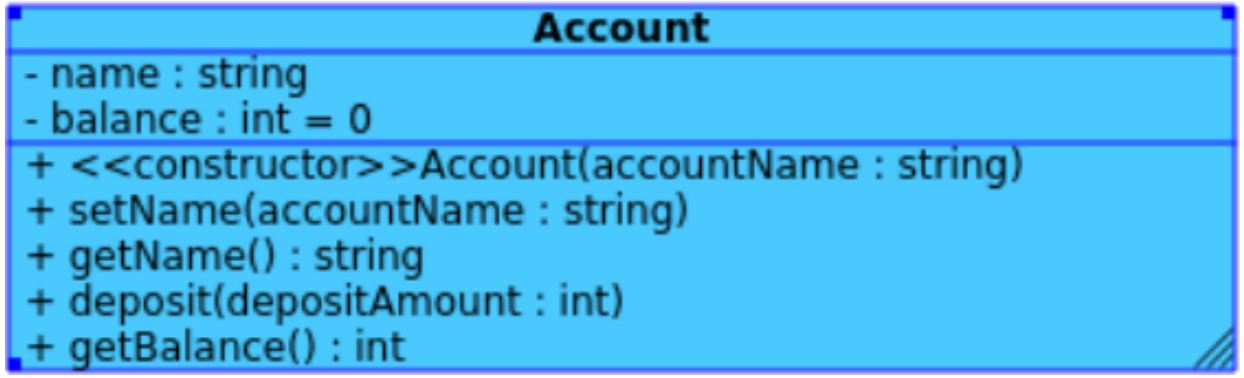
TestBankAccount.cpp

Adding a new `const` member function - `getBalance`

```
3   class Account
4   {
5       public :
6           Account(std::string AccountName, int InitialBalance)
7               : name{AccountName}
8           {
9               if (InitialBalance > 0)
10                  balance = InitialBalance;
11          }
12
13          void setName(std::string AccountName)
14          {
15              name = AccountName;
16          }
17
18          std::string getName() const
19          {
20              return name;
21          }
22
23          void deposit(int depositAmount)
24          {
25              if (depositAmount > 0)
26              {
27                  balance += depositAmount;
28              }
29          }
30
31          int getBalance const(void)
32          {
33              return balance;
34          }
35
36      private :
37          std::string name;
38          int balance{0};
39  };
```

```
int getBalance const(void)
{
        return balance;
}
```

`getBalance` is declared `const`, because in the process of returning the balance, the function does not, and should not, modify anything in the `Account` object.



Account.h

```
31
32
33
34
```

```
int getBalance (void) const
{
    return balance
}
```

student@cse1325: /media/sf_VM

File Edit Tabs Help

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 TestBankAccount.cpp -o TestBankAccount.o
In file included from TestBankAccount.cpp:6:0:
Account.h: In member function ‘int Account::getBalance() const’:
Account.h:34:3: error: expected ‘;’ before ‘}’ token
   }
   ^

makefile:15: recipe for target 'TestBankAccount.o' failed
make: *** [TestBankAccount.o] Error 1
student@cse1325:/media/sf_VM$
```

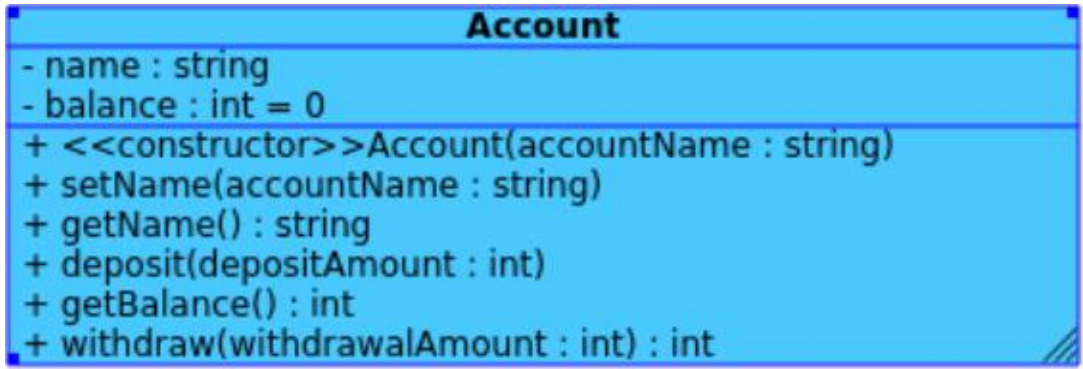
```cpp
#include <iostream>
#include <string>

#include "Account.h"

using namespace std;

int main()
{
    string NewAccountName;
    int deposit;

    Account MyBankAccount{"Bank Account with Lots of Money", 1000000};

    cout << "My bank account's name is " << MyBankAccount.getName();

    cout << "\nEnter a new name for the bank account ";
    getline(cin, NewAccountName);
    MyBankAccount.setName(NewAccountName);

    cout << "My bank account has been renamed " << MyBankAccount.getName() << endl;

    cout << "How many dollars would you like to deposit? ";
    cin >> deposit;
    MyBankAccount.deposit(deposit);

    cout << "\n\nYour balance is " << MyBankAccount.getBalance() << endl;

    return 0;
}
```

TestBankAccount.cpp

Adding a new member function - `withdraw`

```
3   class Account
4   {
5       public :
6           Account(std::string AccountName, int InitialBalance)
7               : name{AccountName}
8           {
9               if (InitialBalance > 0)
10                  balance = InitialBalance;
11          }
12
13          void setName(std::string AccountName)
14          {
15              name = AccountName;
16          }
17
18          std::string getName() const
19          {
20              return name;
21          }
22
23          void deposit(int depositAmount)
24          {
25              if (depositAmount > 0)
26              {
27                  balance += depositAmount;
28
29              }
30          }
31
32          int withdraw(int withdrawalAmount)
33          {
34              if (withdrawalAmount <= balance)
35              {
36                  balance -= withdrawalAmount;
37                  return 1;
38              }
39              else
40              {
41                  return 0;
42              }
43          }
44
45          int getBalance (void) const
46          {
47              return balance;
48          }
49
50      private :
51          std::string name;
52          int balance{0};
53  };
```

```
int withdraw(int withdrawalAmount)
{
  if (withdrawalAmount <= balance)
  {
      balance -= withdrawalAmount;
      return 1;
  }
  else
  {
    return 0;
  }
}
```



```
cout << "\n\nHow many dollars will be withdrawn? ";
cin >> withdrawal;
if (MyBankAccount.withdraw(withdrawal))
{
   cout << "The new balance is " << MyBankAccount.getBalance() << endl;
}
else
{
   cout << "Insufficient funds" << endl;
}
```

```
int withdraw(int withdrawalAmount)
{
   if (withdrawalAmount <= balance)
   {
      balance -= withdrawalAmount;
      return 1;
   }
   else
   {
      return 0;
   }
}
```

```
int PrintMenu(void)
{
  int Choice = 0;

  cout << "0. Exit menu" << endl;
  cout << "1. Get Account Name" << endl;
  cout << "2. Change Account Name" << endl;
  cout << "3. Check Current Balance" << endl;
  cout << "4. Deposit Funds" << endl;
  cout << "5. Withdraw Funds" << endl;
  cout << "\nEnter choice " << endl;
  cin >> Choice;
  getchar();

  return Choice;
}
```

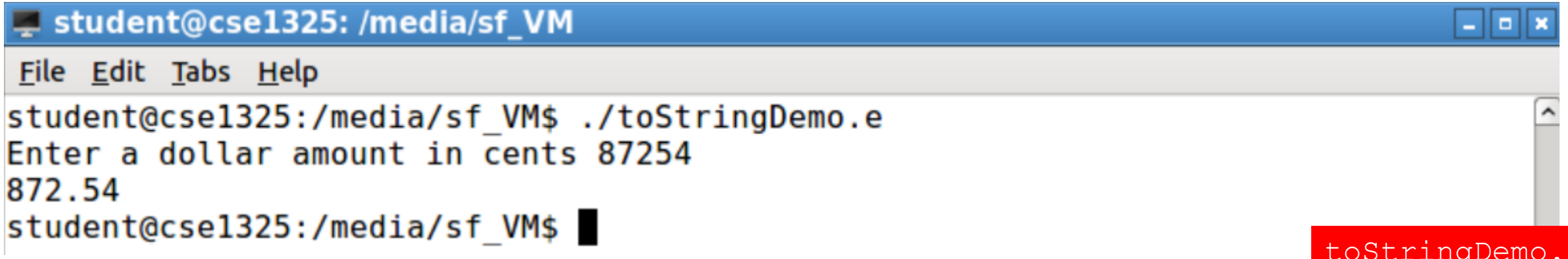
```
0. Exit menu
1. Get Account Name
2. Change Account Name
3. Check Current Balance
4. Deposit Funds
5. Withdraw Funds

Enter choice
```

```
do
{
    Choice = PrintMenu();

    switch (Choice)
    {
        case 1 :
            cout << "\n\nBank account's name is " << MyBankAccount.getName() << endl;
            break;
        case 2 :
            cout << "\n\nEnter a new name for the bank account ";
            getline(cin, NewAccountName);
            MyBankAccount.setName(NewAccountName);
            cout << "\n\nBank account has been renamed " << MyBankAccount.getName() << endl;
            break;
        case 3 :
            cout << "\n\nThe current balance is " << MyBankAccount.getBalance() << endl;
            break;
        case 4 :
            cout << "\n\nHow many dollars will be deposited? ";
            cin >> deposit;
            MyBankAccount.deposit(deposit);
            cout << "The new balance is " << MyBankAccount.getBalance() << endl;
            break;
        case 5 :
            cout << "\n\nHow many dollars will be withdrawn? ";
            cin >> withdrawal;
            if (MyBankAccount.withdraw(withdrawal))
            {
                cout << "The new balance is " << MyBankAccount.getBalance() << endl;
            }
            else
            {
                cout << "Insufficient funds" << endl;
            }
            break;
        default :
            cout << "Goodbye!" << endl;
    }

    cout << "\n\n";
}
while (Choice);
```

BankAccountMenu.cpp

to_string()

```
long amount;

cout << "Enter a dollar amount in cents ";
cin >> amount;

std::string dollars{std::to_string(amount / 100)};

std::string cents{std::to_string(std::abs(amount % 100))};

cout << dollars + "." + (cents.size() == 1 ? "0" : "") + cents << endl;
```



toStringDemo.cpp

```cpp
#include <string>
#include <cmath>

class DollarAmount
{
    public :
        DollarAmount(long value) : amount{value}
        {
        }

        long getAmount(void)
        {
            return amount;
        }

        void add(DollarAmount ObjectToAdd)
        {
            amount += ObjectToAdd.amount;
        }

        void subtract(DollarAmount ObjectToSubtract)
        {
            amount -= ObjectToSubtract.amount;
        }

        std::string displayMoney() const
        {
            std::string dollars{std::to_string(amount / 100)};
            std::string cents{std::to_string(std::abs(amount % 100))};
            return dollars + "." + (cents.size() == 1 ? "0" : "") + cents;
        }
    private :
        long amount{0};
};
```

DollarAmount.h

```cpp
#include <iostream>
#include <string>

#include "DollarAmount.h"

using namespace std;

void PrintLine(DollarAmount DAObject1, DollarAmount DAObject2)
{
    cout << "DAObject1 amount in pennies " << DAObject1.getAmount() << endl;
    cout << "DAObject1 amount in dollars and cents " << DAObject1.displayMoney() << "\n\n";

    cout << "DAObject2 amount in pennies " << DAObject2.getAmount() << endl;
    cout << "DAObject2 amount in dollars and cents " << DAObject2.displayMoney() << endl;
}

int main()
{
    DollarAmount DAObject1{12345}; // $123.45 (represents dollar amounts in whole number of pennies
    DollarAmount DAObject2{655};   // $6.55

    PrintLine(DAObject1, DAObject2);

    cout << "\n\nAdding DAObject2 to DAObject1...\n\n" << endl;
    DAObject1.add(DAObject2);

    PrintLine(DAObject1, DAObject2);

    cout << "\n\nSubtracting DAObject1 from DAObject1...\n\n" << endl;
    DAObject1.subtract(DAObject1);

    PrintLine(DAObject1, DAObject2);

    return 0;
}
```

TestDollarAmount.cpp

Constructor

```
DollarAmount DAObject1{12345}; // 123.45
DollarAmount DAObject2{655};   // 6.55


class DollarAmount
{
   public :
      DollarAmount(long value) : amount{value}
      {
      }
   private :
      long amount{0};
```

getAmount() Member Function

```
cout << "DAObject1 amount in pennies " << DAObject1.getAmount() << endl;

class DollarAmount
{
   public :
      long getAmount(void)
      {
         return amount;
      }
```

```
DAObject1 amount in pennies 12345
```

displayMoney() Member Function

```
cout << "DAObject1 amount in dollars and cents "
     << DAObject1.displayMoney() << "\n\n";

class DollarAmount
{
   public :
      std::string displayMoney() const
      {
         std::string dollars{std::to_string(amount / 100)};
         std::string cents{std::to_string(std::abs(amount % 100))};
         return dollars + "." + (cents.size() == 1 ? "0" : "") + cents;
      }
```

```
DAObject1 amount in dollars and cents 123.45
```

Member functions `add()` and `subtract()`

```
DAObject1.add(DAObject2);
DAObject1.subtract(DAObject1);

class DollarAmount
{
   public :
      void add(DollarAmount ObjectToAdd)
      {
         amount += ObjectToAdd.amount;
      }

      void subtract(DollarAmount ObjectToSubtract)
      {
         amount -= ObjectToSubtract.amount;
      }
```

```
student@cse1325: /media/sf_VM
File  Edit  Tabs  Help
DAObject1 amount in pennies 12345
DAObject1 amount in dollars and cents 123.45

DAObject2 amount in pennies 655
DAObject2 amount in dollars and cents 6.55


Adding DAObject2 to DAObject1...


DAObject1 amount in pennies 13000
DAObject1 amount in dollars and cents 130.00

DAObject2 amount in pennies 655
DAObject2 amount in dollars and cents 6.55


Subtracting DAObject1 from DAObject1...


DAObject1 amount in pennies 0
DAObject1 amount in dollars and cents 0.00

DAObject2 amount in pennies 655
DAObject2 amount in dollars and cents 6.55
```

```
DAObject1{12345}; // 123.45
DAObject2{655};   // 6.55
```

TestDollarAmount.cpp

`friend` Function and `friend` Classes

A **friend function** of a class is a non-member function that has the right to access the public *and* non-public class members.

A **friend function** is a function that can access the private members of a class as though it were a member of that class.

Standalone functions, entire classes or member functions of other classes may be declared to be *friends* of another class.

Declaring a `friend`

To declare a non-member function as a `friend` of a class, place the function prototype in the class definition and precede it with the keyword `friend`.

The `friend` declaration(s) can appear *anywhere* in a class and are not affected by access specifiers public or private (or protected, which we discuss later).

```
#include <iostream>

using namespace std;

class ClassABC
{
   public :
      int getZ() const
      {
         return Z;
      }
   private :
      int Z{0};
};

int main()
{
   ClassABC def;

   cout << "def.Z after instantiation: " << def.getZ() << endl;
}
```

Let's add a friend function called setZ that can update private data member Z

friend1Demo.cpp

```
class ClassABC
{
  friend void setZ(ClassABC&, int);

  public :
    int getZ() const
    {
      return Z;
    }
  private :
    int Z{0};
};
```

```
void setZ(ClassABC& ghi, int newvalue)
{
  ghi.Z = newvalue;
}

int main()
{
  ClassABC def;

  cout << "def.Z after instantiation: "
       << def.getZ() << endl;
  setZ(def, 8);
  cout << "def.Z after call to setZ
          friend function: "
       << def.getZ() << endl;
}
```

friend2Demo.cpp

Properties of `friend`

Friendship is *granted, not taken*—for class B to be a friend of class A, class A must *explicitly* declare that class B is its friend.

Friendship is not *symmetric*—if class A is a friend of class B, you cannot infer that class B is a friend of class A.

Friendship is not *transitive*—if class A is a friend of class B and class B is a friend of class C, you cannot infer that class A is a friend of class C.

Friendship is not inherited.

Rules of `friend`ship

Even though the prototypes for `friend` functions appear in the class definition, `friend`s are not member functions.

Member access notions of private, protected and public are not relevant to `friend` declarations, so `friend` declarations can be placed anywhere in a class definition.

Place all friendship declarations first inside the class definition's body and do not precede them with any access specifier.

Why make friends?

Because everyone should have a friend…

Why does C++ have friend functions?

Allows functionality to be extracted from a class and kept in a non member function for use by multiple classes.

When a database changes, the indexes must be updated. This process can be kept in a friend function rather than a class member function. This type of generic function can then be reused across different database tables.

Functions that are used solely for testing a class can be made friends in the class being tested. This allows for the test code to change without changing the class itself and gives the test function access to the class's private data.

One way to keep classes from inheriting functionality.

Review from 1320

Pointers to Structures

In C, it is possible to declare a pointer to any type

This includes pointers to structures.

```
struct tshirt
{
    char  size[5];
    char  color[10];
    char  design[100];
    char  fittype;
    float price;
    int   inventory;
};
```

```
struct tshirt MyTShirts = {"M","BLUE","DISNEY",'W',29.99,1};
struct tshirt *tshirtptr;
tshirtptr = &MyTShirts;

printf("MyTShirts.design\t%s\n",MyTShirts.design);
printf("(*tshirtptr).design\t%s\n\n", (*tshirtptr).design);
```

```
MyTShirts.design        DISNEY
(*tshirtptr).design     DISNEY
```

Review from 1320

Pointers to Structures

In C, it is possible to declare a pointer to any type

This includes pointers to structures in arrays.

```
struct tshirt DCComicsTShirts[5] = {{"XS","BLACK","BATMAN",'Y',12.99,198},
                                    {"S","BLUE","SUPERMAN",'M',24.99,34},
                                    {"M","RED","WONDER WOMAN",'W',27.99,87},
                                    {"L","YELLOW","AQUAMAN",'M',26.99,65},
                                    {"XL","GREEN","GREEN LANTERN",'Y',15.99,81}
                                    };

struct tshirt *tshirtarrayptr;
tshirtarrayptr = &DCComicsTShirts[3];

printf("DCComicsTShirts[3].design\t%s\n", DCComicsTShirts[3].design);
printf("(*tshirtarrayptr).design\t%s\n", (*tshirtarrayptr).design);
```

```
DCComicsTShirts[3].design       AQUAMAN
(*tshirtarrayptr).design        AQUAMAN
```

Pointers to Structures

The () are necessary because the dot selector has precedence over the dereferencing operator *

```
printf("tshirtptr design\t%s\n\n", (*tshirtptr).design);
printf("tshirtarrayptr design\t%s\n", (*tshirtarrayptr).design);
```

Without the (), the compiler complains

```
printf("tshirtptr design\t%s\n\n", *tshirtptr.design);
```

```
error: request for member ‘design’ in something not a structure or union
```

Pointers to Structures

The concept of a pointer to structure is used so often in C that a special syntax was developed to reference the members of the target structure.

`(*struct_pointer).member` can be written as `struct_pointer->member`

```
printf("tshirtptr design\t%s\n\n", (*tshirtptr).design);
printf("tshirtptr design\t\t%s\n", tshirtptr->design);

printf("tshirtarrayptr design\t%s\n", (*tshirtarrayptr).design);
printf("tshirtarrayptr design\t%s\n", tshirtarrayptr->design);
```

`this`

One of the questions about classes often asked is,

“When a member function is called, how does C++ keep track of which object it was called on?”.

The answer is that C++ utilizes a hidden pointer named “this”!

`this`

There's only one copy of each class's functionality, but there can be many objects of a class, so how do member functions know which object's data members to manipulate?

Every object has access to its own address through a pointer called `this` (a C++ keyword).

The `this` pointer is not part of the object itself. The memory occupied by the `this` pointer is not reflected in the result of a `sizeof` operation on the object. Rather, the `this` pointer is passed (by the compiler) as an implicit argument to each of the object's non-static member functions.

this

```
class Simple
{
   public:
      Simple(int id)
      {
         setID(id);
      }
      void setID(int id)
      {
         m_id = id;
      }
      int getID()
      {
         return m_id;
      }
    private:
      int m_id;
};
```

```
int main()
{
    Simple simple(1);
    simple.setID(2);
    std::cout << simple.getID();

    return 0;
}
```

What would this print?

2

this

When we call

```
simple.setID(2);
```

C++ knows that function `setID()` should operate on object `simple` and that `m_id` actually refers to `simple.m_id`.

How?

this

Let's look at this line of code

```
simple.setID(2);
```

Although the call to function `setID()` looks like it only has one argument, it actually has two!

When compiled, the compiler converts `simple.setID(2);` into the following

```
setID(&simple, 2);
```

Note that `simple` has been changed from an object prefix to a function argument!

this

```
setID(&simple, 2);
```

`setID()` is now just a standard function call, and the object `simple` (which was formerly an object prefix) is now passed by address as an argument to the function.

Since the function call now has an added argument, the member function definition needs to be modified to accept (and use) this argument as a parameter.

this

Since the function call now has an added argument, the member function definition needs to be modified to accept (and use) this argument as a parameter.

```
void setID(int id)
{
   m_id = id;
}
```

is converted by the compiler into

```
void setID(Simple* const this, int id)
{
   this->m_id = id;
}
```

this

```
void setID(Simple* const this, int id)
{
    this->m_id = id;
}
```

When the compiler compiles a normal member function, it **implicitly** adds a new parameter to the function named `this`.

The `this` pointer is a hidden `const` pointer that holds the address of the object the member function was called on.

this

```
void setID(Simple* const this, int id)
{
    this->m_id = id;
}
```

Inside the member function, any class members (functions and variables) also need to be updated so they refer to the object the member function was called on.

This is done by adding a `this->` prefix to each of them.

In the body of function `setID()`, `m_id` (which is a class member variable) has been converted to `this->m_id`.

When `this` points to the address of `simple`, `this->m_id` will resolve to `simple.m_id`.

this

When we call

```
simple.setID(2);
```

the compiler actually calls

```
setID(&simple, 2);
```

Inside `setID()`, the `this` pointer holds the address of object `simple`.

Any member variables inside `setID()` are prefixed with `this->`.

So when we say `m_id = id`, the compiler is actually executing

```
this->m_id = id
```

which in this case updates `simple.m_id to id`.

Using the `this` Pointer to Avoid Naming Collisions

Member functions use the `this` pointer

implicitly (as we've done so far)

or

explicitly

to reference an object's data members and other member functions. A common explicit use of the `this` pointer is to avoid naming conflicts between a class's data members and member-function parameters (or other local variables)

`this`

All of this happens automatically.

Just remember is that all normal member functions have a `this` pointer that refers to the object the function was called on

`this` always points to the object being operated on.

So how many “this” pointers exist?

Each member function has a `this` pointer parameter that is set to the address of the object being operated on.

this

```
int main()
{
    Simple A(1); // this = &A inside the Simple constructor
    Simple B(2); // this = &B inside the Simple constructor
    A.setID(3);  // this = &A inside member function setID
    B.setID(4);  // this = &B inside member function setID

    return 0;
}
```

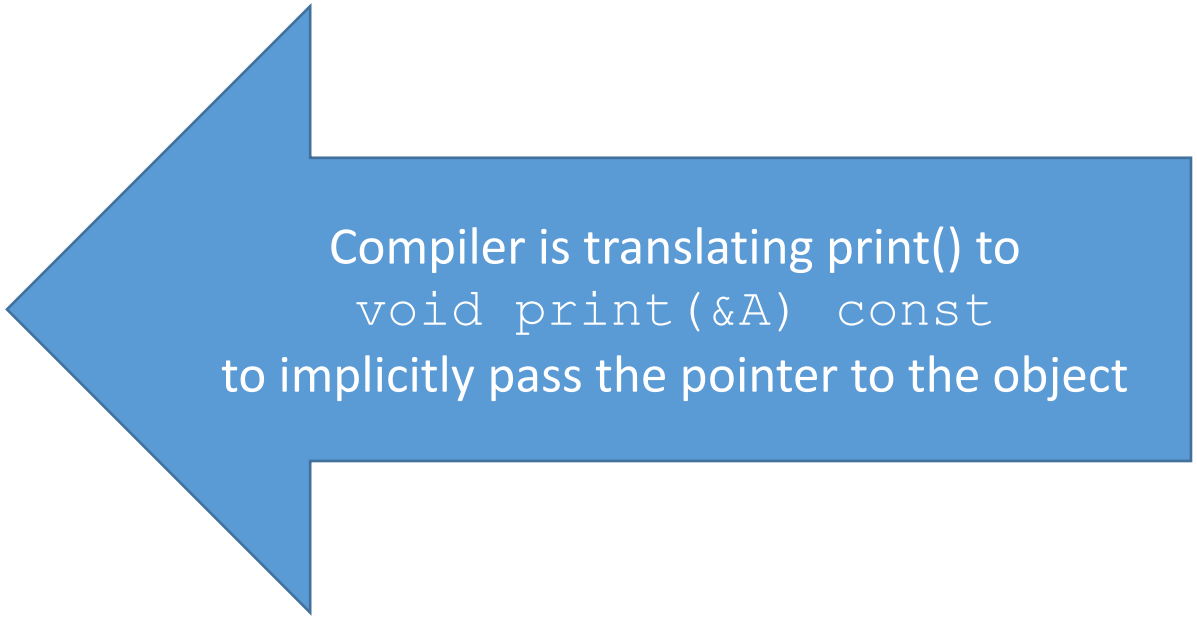
The `this` pointer alternately holds the address of object A or B depending on whether we've called a member function on object A or B.

`this` is just a function parameter - it doesn't add any memory usage to your class

Using the `this` Pointer to Avoid Naming Collisions

We are familiar with the implicit call

```
class Test
{
   public :
      void print(void) const
      {
        cout << "x = " << x;
      }
   private :
      int x{0};
};
```



Using the `this` Pointer to Avoid Naming Collisions

```
void setHour(int hour)
{
    if (hour >= 0 && hour < 24)
    {
            hour = hour;
    }
}
```

`hour` is passed into the function `setHour` and then validation is performed on it.

If it passes validation, then we want to update the object's data member `hour`.

We do this by using `this`.

Using the `this` Pointer to Avoid Naming Collisions

So why not just use different names?

```
void setHour(int hourA)
{
    if (hourA >= 0 && hourA < 24)
    {
        this->hour = hourA;
    }
}
```

A widely accepted practice to minimize the proliferation of identifier names is to use the same name for a set function's parameter and the data member it sets, and to reference the data member in the set function's body via this->.

Using the `this` Pointer to Avoid Naming Collisions

We are familiar with the implicit usage so what does the explicit usage look like?

```
class Test
{
   public :
      void print(void) const
      {
         cout << "x = " << (*this).x;
         cout << "x = " << this->x;
      }
   private :
      int x{0};
};
```

`this`

Recommendation

Do not add `this->` to all uses of your class members.

Only do so when you have a specific reason to.

We will see more examples of when using `this` is necessary.

Type of `this` pointer

The type of the `this` pointer depends on the type of the object and whether the member function in which this is used is declared `const`

In a non-`const` member function of class `Employee`, the `this` pointer has the type

```
Employee* const
```

a constant pointer to a nonconstant `Employee`.

In a `const` member function, this has the type

```
const Employee* const
```

a constant pointer to a constant `Employee`.

`this` is a `const` pointer -- you can change the value of the underlying object it points to, but you can not make it point to something else.