Inheritance and Dynamic Binding

Derived Classes (Subclasses)

- **General Form**

```
class className : [public] parentClass {

    <friend classes>

    private:
     <private data members>
     <private constructors>
     <private member functions>
    protected:
     <protected data members>
     <protected constructors>
     <protected member functions>
    public:
     <public data members>
     <public constructors>
     <public member functions>

    <friend functions and friend operators>
 };
```

Derived Classes

```
class bankAccount {
  protected:
      char owner[50];
      float balance;
      int accountNumber;
      int getNewAcctNum();
  public:
      bankAccount();
      bankAccount(char *name, float amt, int aNo);
      bankAccount(bankAccount& acct); //copy constructor
      float getBalance(){return balance; }
      int getAccountNumber(){return accountNumber; }
      //...
};
class cheqAccount : public bankAccount {
};
class savingAccount : public bankAccount {
};
class termDeposit : public bankAccount {
};
```

...Derived Classes

```
main()
{
  bankAccount acct1;
  cheqAccount acct2;
  acct2.deposit(100);
  savingAccount acct3;
  cout << "\nAcct No: " << acct1.getAccountNumber()
            <<"  Balance $" << acct1.getBalance();
  cout << "\nAcct No: " << acct2.getAccountNumber()
            <<"  Balance $" << acct2.getBalance();
  cout << "\nAcct No: " << acct3.getAccountNumber()
            <<"  Balance $" << acct3.getBalance();
  cout << "\n";

  return 0;
}
```

```
Acct No: 1000  Balance $0
Acct No: 1001  Balance $100
Acct No: 1002  Balance $0
```

Signifying Account Type

```
class bankAccount {
  protected:
       char owner[50];
       float balance;
       int accountNumber;
       int getNewAcctNum();
  public:
       bankAccount();
       bankAccount(char *name, float amt, int aNo);
       bankAccount(bankAccount& acct); //copy constructor
       float getBalance(){return balance; }
       int getAccountNumber(){return accountNumber; }
       char acctCode() {return 'A';}
       char *getOwner(){return owner; }
       void deposit(float amount);
       void withdraw(float amount);
};
```

… Signifying Account Type

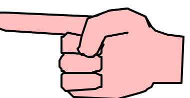
```
main()
{
  bankAccount acct1;
  cheqAccount acct2;
  acct2.deposit(100);
  savingAccount acct3;
  cout << "\nAcct No: " << acct1.acctCode()
             << acct1.getAccountNumber()
             <<"  Balance $" << acct1.getBalance();
  cout << "\nAcct No: " << acct2.acctCode()
             << acct2.getAccountNumber()
             <<"  Balance $" << acct2.getBalance();
  cout << "\nAcct No: " << acct3.acctCode()
             << acct3.getAccountNumber()
             <<"  Balance $" << acct3.getBalance();
  cout << "\n";

  return 0;
}
```

```
Acct No: A1000  Balance $0
Acct No: A1001  Balance $100
Acct No: A1002  Balance $0
```

Overriding Acct Code

```
class cheqAccount : public bankAccount {
  public:
  char acctCode() {return 'C';}
};
class savingAccount : public bankAccount {
  public:
  char acctCode() {return 'S';}
};
class termDeposit : public bankAccount {
  public:
  char acctCode() {return 'T';}
};
```



… Signifying Account Type

```
main()
{
  bankAccount acct1;
  cheqAccount acct2;
  acct2.deposit(100);
  savingAccount acct3;
  cout << "\nAcct No: " << acct1.acctCode()
            << acct1.getAccountNumber()
            <<"  Balance $" << acct1.getBalance();
  cout << "\nAcct No: " << acct2.acctCode()
            << acct2.getAccountNumber()
            <<"  Balance $" << acct2.getBalance();
  cout << "\nAcct No: " << acct3.acctCode()
            << acct3.getAccountNumber()
            <<"  Balance $" << acct3.getBalance();
  cout << "\n";

  return 0;
}
```

```
Acct No: A1000  Balance $0
Acct No: C1001  Balance $100
Acct No: S1002  Balance $0
```

Overriding Member Functions

- **The derived classes can override member functions of the base class**
- **The compiler associates `acct.acctCode()` with the `acctCode()` implementation based on the class of `acct`**

Inherited Member Functions

```
class bankAccount {
  protected:
      char owner[50];
      float balance;
      int accountNumber;
      int getNewAcctNum();
      double interestRatio() {return 0.05;}
  public:
      bankAccount();
      bankAccount(char *name, float amt, int aNo);
      bankAccount(bankAccount& acct); //copy constructor
      float getBalance(){return balance; }
      int getAccountNumber(){return accountNumber; }
      char acctCode() {return 'A';}
      char *getOwner(){return owner; }
      void deposit(float amount);
      void withdraw(float amount);
      void payInterest()
            {deposit(balance * interestRatio());}
};
```

...Inherited Member Functions

```
class cheqAccount : public bankAccount {
  protected:
      double interestRatio() {return 0.00;}
  public:
  char acctCode() {return 'C';}
};

class savingAccount : public bankAccount {
  protected:
      double interestRatio() {return 0.08;}
  public:
  char acctCode() {return 'S';}
};

class termDeposit : public bankAccount {
  protected:
      double interestRatio() {return 0.10;}
  public:
  char acctCode() {return 'T';}
};
```

Paying Interest -What went wrong?

```
main()
{
  bankAccount acct1;
  cheqAccount acct2;
  savingAccount acct3;
  acct1.deposit(100);
  acct2.deposit(100);
  acct3.deposit(100);
  // cout stuff
  cout << "\nPay Interest";
  acct1.payInterest();
  acct2.payInterest();
  acct3.payInterest();
  // cout stuff
  return 0;
}
```

```
Acct No: A1000  Balance $100
Acct No: C1001  Balance $100
Acct No: S1002  Balance $100
Pay Interest
Acct No: A1000  Balance $105
Acct No: C1001  Balance $105
Acct No: S1002  Balance $105
```

What Happend

```
main(){
  // ...
  cheqAccount acct2;
  acct2.deposit(100);
  acct2.payInterest();
  // ...
}
```

When the program ran it executed

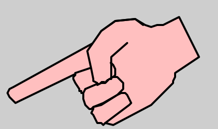
```
bankAccount::payInterest()
      {deposit(balance * interestRatio());}
```

which in turn executed

```
      bankAccount::deposit() and
      bankAccount::interestRatio()
```

What we wanted was to have executed

```
      bankAccount::deposit() and
      cheqAccount::interestRatio()
```



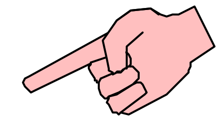
...What Happend

When the program ran it executed

```
bankAccount::payInterest()
     {deposit(balance * interestRatio());}
```

The compiler used the `interestRatio` func. declared in `bankAccount` even though it was executed for a `cheqAccount`

In C++ we need to tell the compiler to use the `interestRatio` func. found in the derived class



Virtual Functions

```
class bankAccount {
  protected:
      char owner[50];
      float balance;
      int accountNumber;
      int getNewAcctNum();
      virtual double interestRatio() {return 0.05;}
  public:
      bankAccount();
      bankAccount(char *name, float amt, int aNo);
      bankAccount(bankAccount& acct); //copy constructor
      float getBalance(){return balance; }
      int getAccountNumber(){return accountNumber; }
      char acctCode() {return 'A';}
      char *getOwner(){return owner; }
      void deposit(float amount);
      void withdraw(float amount);
      void payInterest()
            {deposit(balance * interestRatio());}
};
```

Paying Interest -This time its right

```
main()
{
  bankAccount acct1;
  cheqAccount acct2;
  savingAccount acct3;
  acct1.deposit(100);
  acct2.deposit(100);
  acct3.deposit(100);
  // cout stuff
  cout << "\nPay Interest";
  acct1.payInterest();
  acct2.payInterest();
  acct3.payInterest();
  // cout stuff
  return 0;
}
```

```
Acct No: A1000  Balance $100
Acct No: C1001  Balance $100
Acct No: S1002  Balance $100
Pay Interest
Acct No: A1000  Balance $105
Acct No: C1001  Balance $100
Acct No: S1002  Balance $108
```

Virtual Functions

- Virtual Functions are bound at run-time
- Declared by placing `virtual` before functions return type
- Once a function is declared virtual in a base class, all descendants will treat it is virtual -even if not specified
- “Once virtual always virtual”
- Virtual functions can only be overridden in the derived class and only with the same parameter lists

Static versus Dynamic Binding

- ```
  class BankAccount{ deposit(float); …};
  class SavingAccount : public BankAccount {
  deposit(float); …};
  ```

Type Relaxation

```
BankAccount *p = new SavingAccount;
BankAccount & b = *(new SavingAccount);
```

Static Binding:

```
p->deposit(100);  //means BankAccount::deposit()
b.deposit(100); //means BankAccount::deposit()
```

Dynamic Binding:

```
class BankAccount{ virtual deposit(float); …};

p->deposit(100);  //meansSavingAccount::deposit()
b.deposit(100); //means SavingAccount::deposit()
```
```

Virtual Tables: Implementing Dynamic Binding

```
Class BankAccount {
   virtual void deposit(float);
   virtual void withdraw(float);
   virtual float getBalance(); };

class SavingAccount :
public BankAccount{
   void deposit(float); };

class CheqAccount :
public BankAccount{
   void withdraw(float);
};

class Term :
public SavingAccount{
   void deposit(float);
   void withdraw(float);
};
```

```
&BankAccount::deposit
&BankAccount::withdraw
&BankAccount::payInterest

&SavingAccuont::deposit
&BankAccount::withdraw
&BankAccount::payInterest

&BankAccount::deposit
&CheqAccount::withdraw
&BankAccount::payInterest

&Term::deposit
&Term::withdraw
&BankAccount::payInterest
```

Virtual Tables: Implementing Dynamic Binding

```
BankAccount *p = new CheqAccount;
BankAccount *q = new Term;

p->withdraw(100);  //p->vtblptr[1](100);
P->deposit(200);   //p->vtblptr[0](200);
cout << p->getBalance();

q->deposit(100);
q->withdraw(30);
cout << p->getBalance();
```

```
&BankAccount::deposit
&BankAccount::withdraw
&BankAccount::payInterest

&SavingAccuont::deposit
&BankAccount::withdraw
&BankAccount::payInterest

&BankAccount::deposit
&CheqAccount::withdraw
&BankAccount::payInterest

&Term::deposit
&Term::withdraw
&BankAccount::payInterest
```

<u>Vtblptr</u>
Owner
balance

<u>Vtblptr</u>
Owner
balance

Virtual and Overloaded Functions (Silly Example)

```
class classA {
  public:
       virtual void foo(char c)
         {cout << "virtual classA::foo() rtn: " << c << "\n";}
};
class classB : public classA {
  public:
       void foo(const char* s)
         {cout << "classB::foo() rtn: " << s << "\n";}
       void foo(int i)
         {cout << "classB::foo() rtn: " << i << "\n";}
       virtual void foo(char c)
         {cout << "virtual classB::foo() rtn: " << c << "\n";}
};
class classC : public classB {
  public:
       void foo(const char* s)
         {cout << "classC::foo() rtn: " << s << "\n";}
       void foo(double x)
         {cout << "classC::foo() rtn: " << x << "\n";}
       virtual void foo(char c)
         {cout << "virtual classC::foo() rtn: " << c << "\n";}
};
```

…Virtual and Overloaded Functions

```
main()
{
  int n = 100;
  classA aA;
  classB aB;
  classC aC;

  aA.foo('A');
  aB.foo('B');
  aB.foo(n);
  aB.foo("classB");
  aC.foo('C');
  aC.foo(3.1415);
  aC.foo("classC");

  return 0;
}
```

```
virtual classA::foo() rtn: A
virtual classB::foo() rtn: B
classB::foo() rtn: 100
classB::foo() rtn: classB
virtual classC::foo() rtn: C
classC::foo() rtn: 3.1415
classC::foo() rtn: classC
```

…Non virtual versions of virtuals are not inherited

```
main()
{
  int n = 100;
  classA aA;
  classB aB;
  classC aC;
  aA.foo('A');
  aB.foo('B');
  aB.foo(n);
  aB.foo("classB");
  aC.foo('C');
  aC.foo(n); //THIS GIVES COMPILE ERROR
  aC.foo(3.1415);
  aC.foo("classC");

  return 0;
}
```

Virtual Functions and Overloading

- “Once virtual always virtual”
- Virtual functions can only be overridden in the derived class and only with the same parameter lists
- You can declare non-virtual and overloaded functions with the same name as a virtual functions (not good idea)
- The non-virtual functions must have different parameter lists
- The non-virtual functions are not inherited

- Giver programmers more freedom, but be careful

Invoking Overridden Member Functions (printing accts)

```
class bankAccount {
  protected:
       char owner[50];
       float balance;
       int accountNumber;
       int getNewAcctNum();
       virtual double interestRatio() {return 0.05;}
  public:
       bankAccount();
       bankAccount(char *name, float amt, int aNo);
       bankAccount(bankAccount& acct); //copy constructor
       //...
       virtual void print();
};

void bankAccount::print() {
  cout << "Acc:" << accountNumber << " $" << balance << "\n";
}
```

...Invoking Overridden Member Functions (printing accts)

```
class cheqAccount : public bankAccount {
  protected:
       double interestRatio() {return 0.00;}
  public:
  void print()
       {cout << "Cheq"; bankAccount::print(); }
};
class savingAccount : public bankAccount {
  protected:
       double interestRatio() {return 0.08;}
  public:
  void print()
       {cout << "Sav"; bankAccount::print(); }
};
class termDeposit : public savingAccount {
  protected:
       double interestRatio() {return 0.10;}
  public:
  void print()
       {cout << "Term"; bankAccount::print(); }
};
```

...Invoking Overridden Member Functions (printing accts)

```
main()
{
  bankAccount acct1;
  cheqAccount acct2;
  savingAccount acct3;
  acct1.deposit(100);
  acct2.deposit(100);
  acct3.deposit(100);
  acct1.print();
  acct2.print();
  acct3.print();
  cout << "\nPay Interest\n";
  acct1.payInterest();
  acct2.payInterest();
  acct3.payInterest();
  acct1.print();
  acct2.print();
  acct3.print();
  return 0;
}
```

```
Acc:1000 $100
CheqAcc:1001 $100
SavAcc:1002 $100

Pay Interest
Acc:1000 $105
CheqAcc:1001 $100
SavAcc:1002 $108
```

...Invoking Overridden Member Functions (printing accts)

```
void cheqAccount::print(){
  cout << "Cheq";
  bankAccount::print();
}
```

- **bankAccount `print()` implementation is explicitly refered to since `cheqAccount` class overrides this function**
- **Necessary because C++ does not support the notion of `super` in Smalltalk**
- **In this way we can invoke functions which have been overridden in derived classes (one of the three ways of using inheritance)**
- **Does this work for constructors?**

Initializing Instance Attributes

```
#include <classlib\date.h>
class bankAccount {
        //...
  public:
        //...
        virtual void print();
};
class cheqAccount : public bankAccount {
  public:
  void print();
};
class savingAccount : public bankAccount {
  protected:
  double interestRate; //for now
  public:
  void print();
};
class termDeposit : public savingAccount {
  protected:
        TDate date;
  public:
  void print();
};
```

... Initializing Instance Attributes

```
bankAccount::bankAccount() {
  balance = 0;
  accountNumber = getNewAcctNum();
}
void bankAccount::print() {
  cout << "Acc:" << accountNumber << " $" << balance;
}
void cheqAccount::print(){
  cout << "Cheq";
  bankAccount::print();
}
void savingAccount::print(){
  cout << "Sav";
  bankAccount::print();
  cout << " int rate: " << interestRate;
}

void termDeposit::print(){
  cout << "Term";
  savingAccount::print();
  cout << " " << date;
}
```

Only constructor so far

... Initializing Instance Attributes

```
main()
{
  bankAccount acct;
  cheqAccount cheqacc;
  savingAccount savacc;
  termDeposit termdep;
  acct.print();
  cout << "\n";
  cheqacc.print();
  cout << "\n";
  savacc.print();
  cout << "\n";
  termdep.print();
  return 0;
}
```

In each case the `bankAccount` constructor has initialized the account number and balance

Also the date appears to have been initialized (by its constructor) though we cannot initialize interest Rate

```
Acc:1000 $0
CheqAcc:1001 $0
SavAcc:1002 $0 int rate: 3.74828e-205
TermSavAcc:1003 $0 int rate: 6.77279e-312 February 5, 1997
```

... Initializing Instance Attributes

```
class savingAccount : public bankAccount {
  protected:
  double interestRate = 0.08; //for now
  public:
  void print();
};
```

COMPILE ERROR:
Cannot initialize a class member here

We need to create constructors for the derived classes and also cause the base class constructors to run

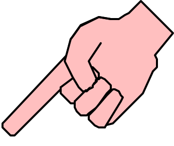
... (Constructors for Derived Classes)

```
bankAccount::bankAccount() {
  balance = 0;
  accountNumber = getNewAcctNum();
}
bankAccount::bankAccount(float amt) {
  balance = amt;
  accountNumber = getNewAcctNum();
}
savingAccount::savingAccount() {
  interestRate = 0.08;
}
savingAccount::savingAccount(double rate) {
  interestRate = rate;
}
termDeposit::termDeposit() {
}
```

... Initializing Instance Attributes

```
main()
{
  bankAccount acct;
  cheqAccount cheqacc;
  savingAccount savacc;
  termDeposit termdep;
  acct.print();
  cout << "\n";
  cheqacc.print();
  cout << "\n";
  savacc.print();
  cout << "\n";
  termdep.print();
  return 0;
}
```

For each derived class the base classes' <u>default</u> constructors has run



```
Acc:1000 $0
CheqAcc:1001 $0
SavAcc:1002 $0 int rate: 0.08
TermSavAcc:1003 $0 int rate: 0.08 February 5, 1997
```

... Invoking a base class constructor (Wrong Way)

```
bankAccount::bankAccount() {
  balance = 0;
  accountNumber = getNewAcctNum();
}
bankAccount::bankAccount(float amt) {
  balance = amt;
  accountNumber = getNewAcctNum();
}
savingAccount::savingAccount() {
  interestRate = 0.08;
}
savingAccount::savingAccount(double rate) {
  interestRate = rate;
}
termDeposit::termDeposit() {
  savingAccount::savingAccount( 0.10);
}
```

Compile Error: Cannot invoke a constructor

... Invoking a base class constructor (Right Way)

```
bankAccount::bankAccount() {
  balance = 0;
  accountNumber = getNewAcctNum();
}
bankAccount::bankAccount(float amt) {
  balance = amt;
  accountNumber = getNewAcctNum();
}
savingAccount::savingAccount() {
  interestRate = 0.08;
}
savingAccount::savingAccount(double rate) {
  interestRate = rate;
}
termDeposit::termDeposit() : savingAccount( 0.10){
}
```

Initializer List

... Initializing Instance Attributes

```
main()
{
  bankAccount acct;
  cheqAccount cheqacc;
  savingAccount savacc;
  termDeposit termdep;
  acct.print();
  cout << "\n";
  cheqacc.print();
  cout << "\n";
  savacc.print();
  cout << "\n";
  termdep.print();
  return 0;
}
```

TermDeposit constructor has invoked a specific SavingAcct constructor to increase interest rate

```
Acc:1000 $0
CheqAcc:1001 $0
SavAcc:1002 $0 int rate: 0.08
TermSavAcc:1003 $0 int rate: 0.1 February 5, 1997
```

... Invoking a base class constructor (Also Wrong)

```
bankAccount::bankAccount() {
  balance = 0;
  accountNumber = getNewAcctNum();
}
bankAccount::bankAccount(float amt) {
  balance = amt;
  accountNumber = getNewAcctNum();
}
savingAccount::savingAccount() {
  interestRate = 0.08;
}
savingAccount::savingAccount(double rate) {
  interestRate = rate;
}
termDeposit::termDeposit()
  :  bankAccount(150.0), savingAccount( 0.10) {
}
```

COMPILE ERROR -Why?