第4章 基本的な設計手順

1 アプリケーションモデルを考える
2 Visual C++ 2010でMVCモデルを活用する
3 小遣い帳アプリケーションのモデル
4 動作のイメージ
5 見取り図の作成
6 HomeBankの見取り図

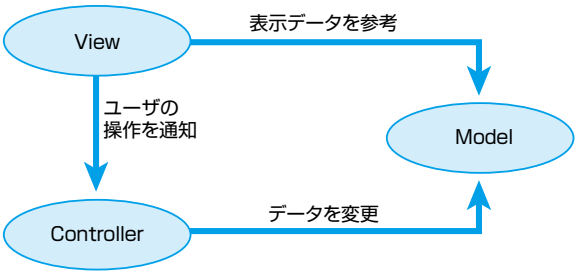
この章からは、具体的に「小遣い帳」のサンプルアプリケーションを作りながら、Visual C++ 2010を 学習していきます。この章ではMVCモデルを使って、サンプルアプリケーションの簡単な設計を行います。

この章で学習する内容と身に付くテクニック

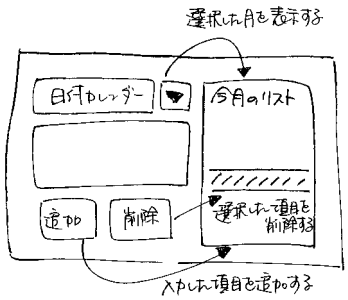
この章では、プログラム作成にあたって、最初に行うべき設計について学習します。主な学習内容は次のとおりです。

- **どのようなアプリケーションが作りたいかをはっきりさせる**
- **必要な機能をはっきりさせる**
- **アプリケーションの動きを推測する**
- **見取り図を描く**

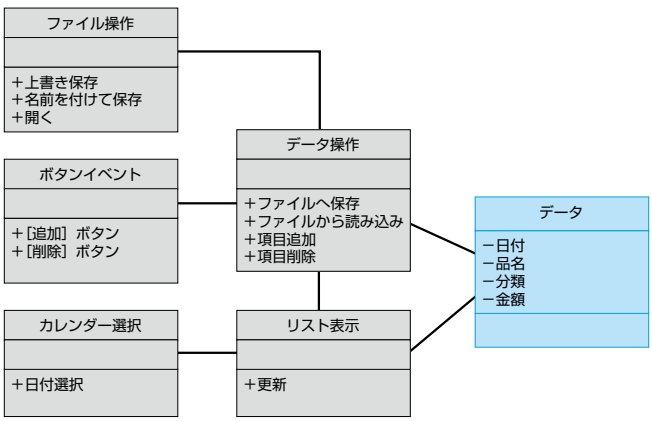
STEP 1 MVC（Model-View-Controller）モデルとは何かを説明します。



STEP 2 アプリケーションができあがったときの動きを想像してみます。[追加] ボタンをクリックしたときの動き、[削除] ボタンをクリックしたときの動き、カレンダーで日付を選択したときの動きなどを確認します。



STEP 3 最後に見取り図や画面イメージを作成して、アプリケーションの機能や動きをまとめておきます。



1 アプリケーションモデルを考える

アプリケーションを作成するときには、最初の「要件」や「構成」が大切になります。要件（Requirement）はアプリケーションを作成するときのユーザー（顧客など）から与えられることがありますが、その要件を満たすようにシステムまたはアプリケーションの構成を考え出さなければいけません。

本書で説明するような小さなアプリケーションの場合、特に構成を決めずに、がしがしとコードを組んでいってもある程度完成してしまいます。しかし、プログラミングする期間が数か月単位と長期にわたる場合や、複数のプログラマで1つのアプリケーションを完成させようとする場合は、アプリケーションの構成を決めて皆で認識をすり合わせしておくことが大切です。そうしないと、それぞれのプログラマが持つ完成イメージがだんだんとずれていってしまい、いざ組み合せようとしたときに最初のイメージと合わないものができてしまうことがあります。

また、大きなシステムの場合には、いきなり複雑なものを作っても到底うまくいきません。少しずつ単純な部品を組み立てて、組み上げていくことが必要です。それぞれの部品が正しく動くように完成されていれば、それらを組み合わせたアプリケーションのテストは簡単になっていきます。部品が未完成だったり、不具合を含んでいたりすると、組み立て方が悪いのか、部品そのものが悪いのかの区別がつかなくなります。

この構成を決める作業が「設計」です。特に最近のアプリケーション開発では、過去の知識の蓄積から適切なモデルを使うことが重要になっています。

さて、本書ではサンプルアプリケーションとして「小遣い帳アプリケーション」を作っていきます。小遣い帳アプリケーションは、名前のとおり小さな家計簿です。しかし、ひとことで「家計簿」と言っても、市販されている家計簿からExcelなどで作成した家計簿まで色々あります。ですので、アプリケーションのイメージを固めるために、筆者からいくつかの要件を出しておきます。

- 今日買った品名や金額などをアプリケーションに入力して保存できる。
- 品名は分類を付けて複数保存できる。
- 後から日付を指定して家計簿を開くことができる。
- 一覧は月単位で表示する。月の合計も表示する。

このアプリケーションを作るときに、いきなり統合外発環境を起動してWindowsフォームにボタンやリストボックスを配置して作ってもよいのですが、ちょっと構成をまとめておきましょう。小遣いアプリケーションにはいくつかの重要な部品が必要です。これを書き出していくと次の2点になります。

- 月単位の品名を、入力したり一覧表を表示したりする画面
- 入力したデータを保存したり読み込んだりする機能

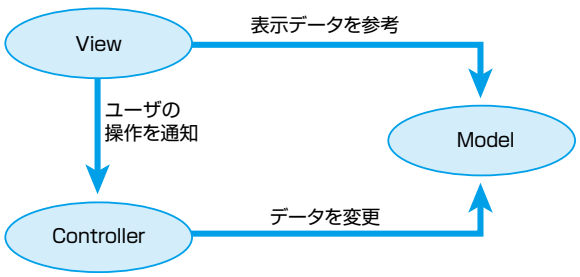
1つはユーザーが入力するための画面です。アプリケーションはユーザーが入力するための画面を持っています。小遣い帳アプリケーションのように、データを入力したり、あらかじめ入力してあるデータを表示したりする場所です。これを、ビュー（View）と呼びます。

もう1つはデータを蓄積するための場所です。小遣い帳の場合はファイルに保存しますが、アプリケーションによってはデータベースを利用することもあるでしょう。あるいはインターネットを使ってデータをサーバーに送信するかもしれません。このデータを使うところをモデル（Model）と呼びます。

実は、ビュー（View）とモデル（Model）を使ったアプリケーションの構成の定番があります。独自に構成を模索してもいいのですが、ここでは先人の知恵は有効に活用することにしましょう。それは「MVCモデル」という設計の方法です。

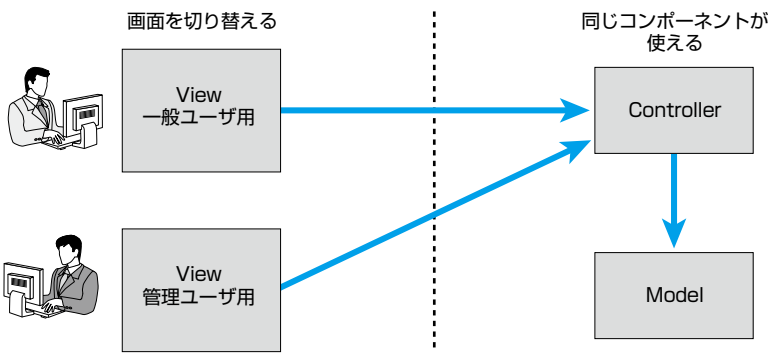
MVCモデルとは、モデル（Model）とビュー（View）、そしてコントローラー（Controller）と呼ばれる3つの部分に分かれたアプリケーションの設計方法です。それぞれの部分は右の図のようになります。

この設計方法の利点は、ユーザーが操作する画面（ビュー）とアプリケーションが持っているデータ（モデル）がうまく分かれていることです。先ほど、アプリケーションを作成するときには部品を組合せたほうがうまくいくという話をしましたが、これが1つの解決策になります。



ビューを分けておくことで、利用者（ユーザー）に対する画面を比較的簡単に切り替えることができます。小遣い帳アプリケーションは月単位の一覧表の画面しか持ちませんが、これを年単位にしたり、すべてを表示したりする画面を作ることもできます。このときアプリケーションが持っているデータは1種類しか必要ありません。右下の図のようにそれぞれの画面とデータを組み合わせることで、部品の数を少なくできます。

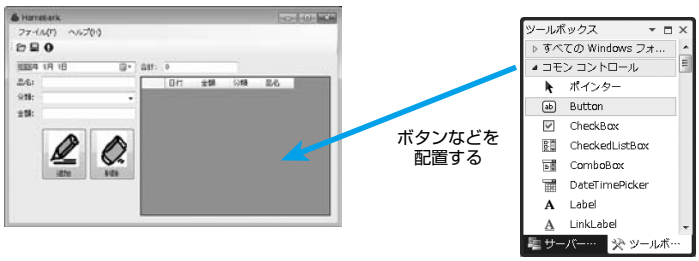
画面とデータを組み合わせるためのコントローラーは、ここでは繋ぎのようなものです。画面から直接データを扱うのではなく、ワンクッションおいてコントローラーを通して保存されたデータを扱います。このようにしておくことで、保存するデータの形式に依存しないアプリケーション開発が可能です。これらは、実際に本書で小遣い帳アプリケーションを作っていく中で実感できるでしょう。



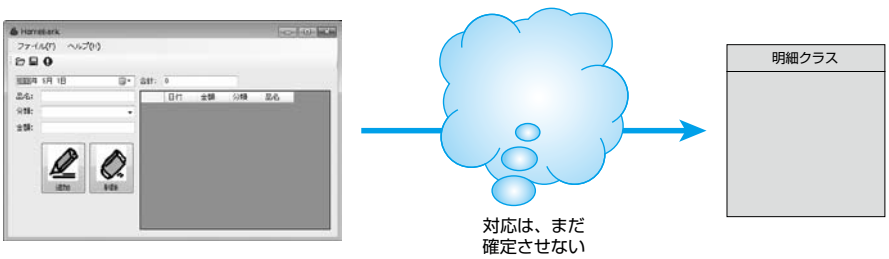
2 Visual C++ 2010でMVCモデルを活用する

具体的に、Visual C++ 2010では、どのようにMVCモデルを使うのか見てみましょう。

まずは、ユーザーが利用するところの画面を作ります。フォームの上に、品名を入力するためのテキストボックスや日付を入力するためのカレンダーコントロールなどを配置します。

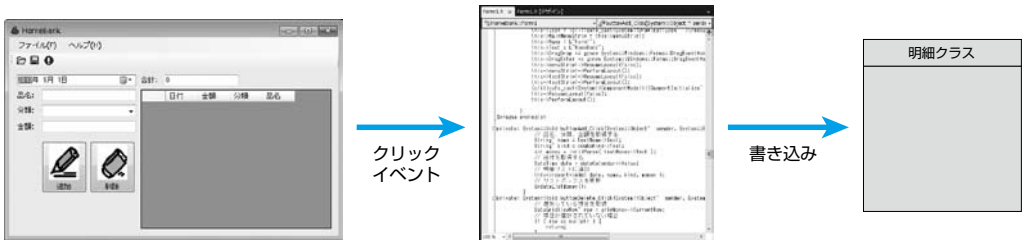


ここで作成したビューは、後から配置を変えることができます。また、品名を入力するためのテキストボックスを改良して、よく入力する項目をリストで表示させることも可能です。コントロールの配置や、どんなコントロールから入力されたかは、モデル（データ）とは関係しないように作ります。



そこで、モデルの部分は、入力した品名などを保存しておくことに集中させます。ここでは「明細クラス」として、1つのクラスを決めておきます。明細クラスの詳しい中身については、もう少し先で考えていきましょう。

さて、画面の部分（ビュー）とデータの部分（モデル）の2つができたので、繋ぎの部分であるコントローラーを見てみましょう。



たとえば、ボタンをクリックしたときには、クリックイベントが発生します。Visual C++ 2010ではデリゲートという機能でこれが実現されています。統合開発環境では、フォームデザイナーでボタンをダブルクリックすると、自動的にイベントを記述するためのメソッドが作成されます。

このボタンイベント内に、データを扱う明細クラスを操作するためのコードを書いていきます。これがコントローラーの役目になります。

コントローラーの部分は、画面とデータのやり取りに使います。画面のデザインが変わってもデータを扱うクラス（明細クラス）が変化しないように、クッションの役割になります。

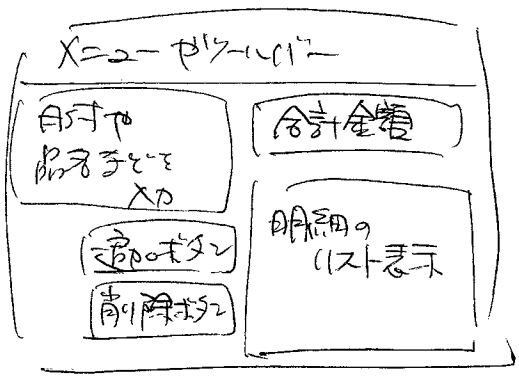
3 小遣い帳アプリケーションのモデル

それでは、本書で作成するサンプルプログラム（小遣い帳アプリケーション）のMVCモデルを詳しく見ていきましょう。

小遣い帳アプリケーションの要件の中から、画面に関するものを書き出していきます。

- 品名などの追加や削除ができる。
- どのファイルから読み込むのか選択できる。
- どのファイルに書き出すのか選択できる。
- 今月の収支をリストで一覧表示する。
- 今月の収支の合計を計算して表示する。

これらの入出力を画面から行おうとすると、次のような画面ができます。



最初はこのような手書きの画面で十分です。Excelなどを使って最初から綺麗な画面を作ってもいいのですが、画面の修正に手間がかかるので、試行錯誤をするときには向きません。大雑把な形で何枚か書いておくといいでしょう。

画面に配置されているコントロールは、ボタンやテキストボックス、リストボックスです。また、ユーザーの使い勝手を考えて、メニューやツールバーを加えておきましょう。これが小遣い帳アプリケーションの最初の画面の設計図になります。

次に、データの部分（モデル）を考えていきましょう。データに保存しなければいけない項目は以下の4つです。

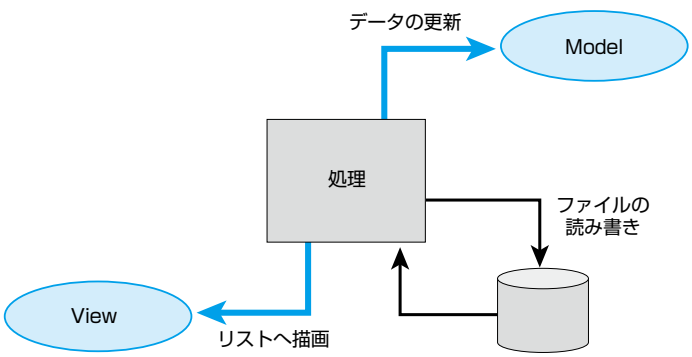
- 購入した日付
- 購入した品名
- 購入したときの金額
- 後で整理するための分類項目

この4つの項目がデータとして明細クラスに保存されます。ファイルに保存したり、保存されているファイルから読み込んだりする場合には、このクラスを使ってアクセスします。

最後に、画面とデータ以外の部分であるコントローラーを考えます。コントローラーには以下の機能を追加します。

- 画面から小遣い帳のデータを更新する機能
- 指定した月のデータを画面に表示する機能
- 作成途中のデータをファイルに保存したり、読み込んだりする機能

これらの機能を図にすると次のようになります。



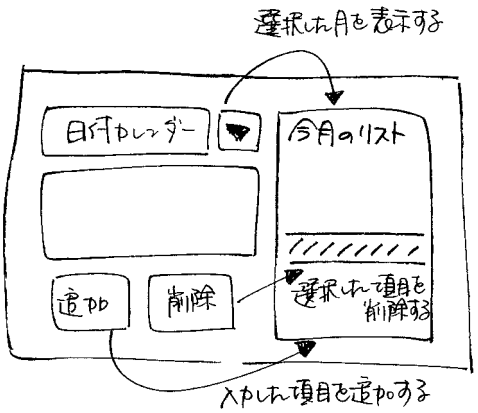
小遣い帳アプリケーションは小さなアプリケーションなので、MVCモデルを利用する利点は明確にならないかもしれません。しかし、この分割の仕方を見てもわかるとおり、それぞれの部品に分けておくことによって、部品ごとに完成させていったり、少しずつ部品を拡張したりすることが可能です。本書では扱いませんが、単体テストのためのツールを使うことで、モデルやコントローラー単体でテストを行って、完成度を高めることができます。

4 動作のイメージ

今度は画面イメージを見ながら、実際の処理の動きを考えます。

動作を想像する

実現したい機能の図や、Windowsアプリケーションの動きを加えた機能のリストと、画面イメージを見比べます。そして、ボタンをクリックしたときの動きはどうなるのか、そのときに今月のリストの状態はどうなるのかを推測してみてください。



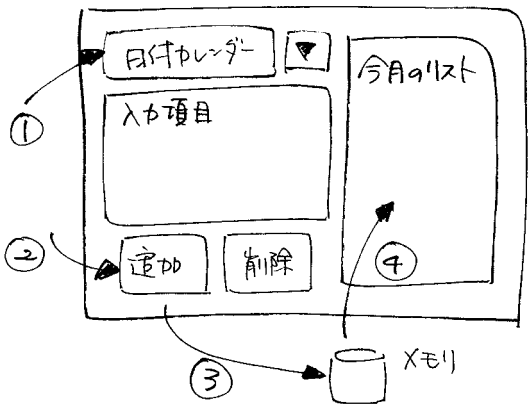
■画面イメージに動作を書き加えてみる

たとえば、［追加］ボタンをクリックしたときの動作を考えてみましょう。

［追加］ボタンをクリックすると、分類や金額が右側にある今月のリストに追加されます。ここから［追加］ボタンと［今月のリスト］が連携して動いていることが想像できます。［追加］ボタンをクリックするというユーザーの動作を受けて、Windowsアプリケーションが［今月のリスト］に項目を追加して、［今月のリスト］を表示する、という流れになると考えることができます。

［追加］ボタンをクリックしたときの動き

具体的な動きを確認しましょう。たとえば、昨日、「Visual C++入門」という本を買って小遣い帳に入力する場合を考えてみます。



■追加ボタンをクリックしたときの流れ

① カレンダーで昨日の日付を選択する。品名は「Visual C++入門」、分類は「本代」、金額は2000円とする。
② ［追加］ボタンをクリックする。
③ プログラムの中にあるリストデータに、先ほど入力した情報が追加される。
④ リストデータに従って、今月のリストを更新する。その結果、今月のリストに、「Visual C++入門 本代 2000円」という1行が追加される。

画面から行う操作に対して、プログラムの中ではどのように処理されて、再び画面がどう変化するのかという流れをつかむことがポイントになります。画面イメージを使って指で追っていくだけでも十分です。どのようにプログラム内部で処理するのかの感触を得ることが大切です。

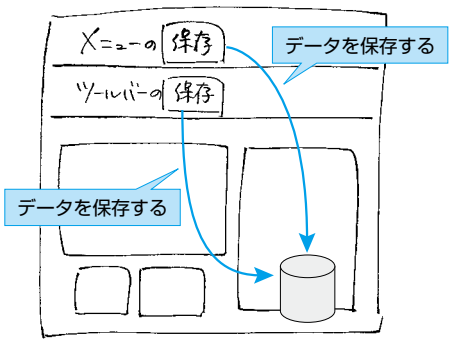
ヒント

ペーパープロトタイプによる動作確認

実装する機能をピックアップしたら、実際にどのような操作をするのかを確認してみましょう。どんなにすばらしい機能でも、ユーザーの操作手順が複雑すぎる場合は、まったく使用されない機能になってしまいます。また、操作手順を簡単にしすぎた結果、複数の機能が混乱するような状態になっても困ります。
本書では、動きを確認する際に手書きの画面イメージ（ペーパープロトタイプ）と矢印や番号を使っています。リストボックスのいくつかの表示パターンを用意しておき、紙芝居のようにボタンを押したときに切り替わるようにすると、より具体的な操作が確認できます。

メニューの［保存］をクリックしたときの動き

メニューの［保存］をクリックしたときのプログラムの動きを確認しましょう。



■データの保存

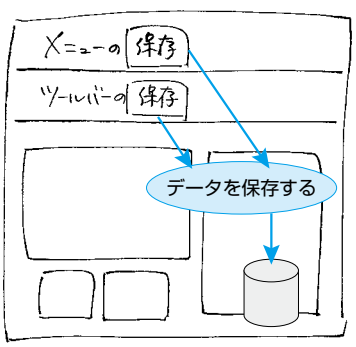
ファイルに「保存」する動きを示すと、左の図のようになります。小遣い帳では、［ファイル］メニューの［保存］をクリックした場合と、ツールバーの［保存］ボタンをクリックした場合の2とおりの方法を用意しましょう。この結果、図のようにメニューやツールバーのボタンからファイルへ2本の矢印が引けます。

ここで、改めて考えてみましょう。

- メニューの［ファイル］－［保存］をクリックし、ファイルへ保存する。
- ツールバーの［保存］ボタンをクリックし、ファイルへ保存する。

この2つの文章を比べると、共通な部分が見えてきます。前半の「メニューの…」と「ツールバーの…」は異なっていますが、後半の「ファイルへ保存する」の部分は同じです。つまり、どちらの動作も、「○○した後に、ファイルへ保存する」という形になっています。

共通の部分を明らかにすると、次の図のようになります。



■共通の処理

ファイルへ保存する機能を「データを保存する」という形でまとめます。これによって、「データを保存する」共通部分が、実際にファイルへデータを保存する作業を行うのです。

先の2つの文章を、共通部分が明確になるように書き直すと、次のようになります。

- メニューの［ファイル］－［保存］をクリックし、データを保存する機能を呼び出す。
- ツールバーの［保存］ボタンをクリックし、データを保存する機能を呼び出す。

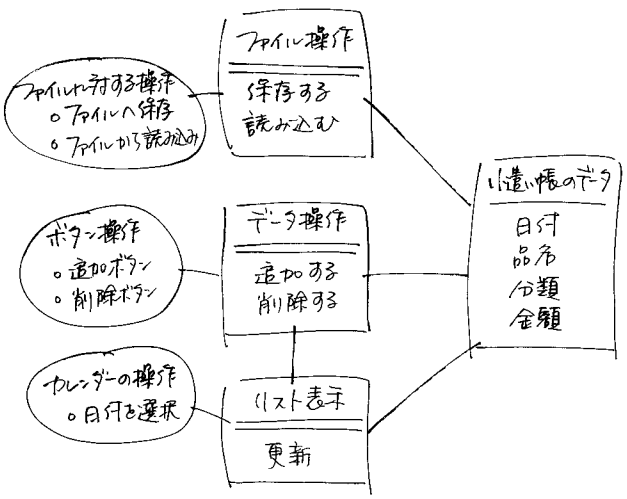
いかがでしょうか。画面イメージを使ってプログラムの動きをたどってみましたが、うまく想像できたでしょうか。最初にプログラムがどのように動くのかをイメージしておくことが、次のステップに確実に進む道になります。

5 見取り図の作成

小遣い帳の機能を取捨選択し、画面イメージを使用して動きを想像したところで、いよいよ小遣い帳の見取り図を描いていきます。

簡単な見取り図

本来は、機能と画面イメージ、そして動作を相互に確認しながら、少しずつ見取り図を描いていくべきなのですが、紙面の都合上長く説明できません。ここでは、簡単な見取り図を用意しました。ここからスタートしましょう。



■小遣い帳の簡単な見取り図

この図を簡単に説明します。

●左側の楕円

ユーザーが行うことができる「動作」を記述しています。メニューの［上書き保存］や［名前を付けて保存］をクリックしたときは「ファイルへ保存」という動作、［開く］をクリックしたときは「ファイルから読み込み」という動作になります。同様に、ボタンをクリックする動作は、［追加］ボタンをクリックする動作と、［削除］ボタンをクリックするときの動作になります。また、カレンダーで日付を選択するという動作も加えてあります。

●右側の四角

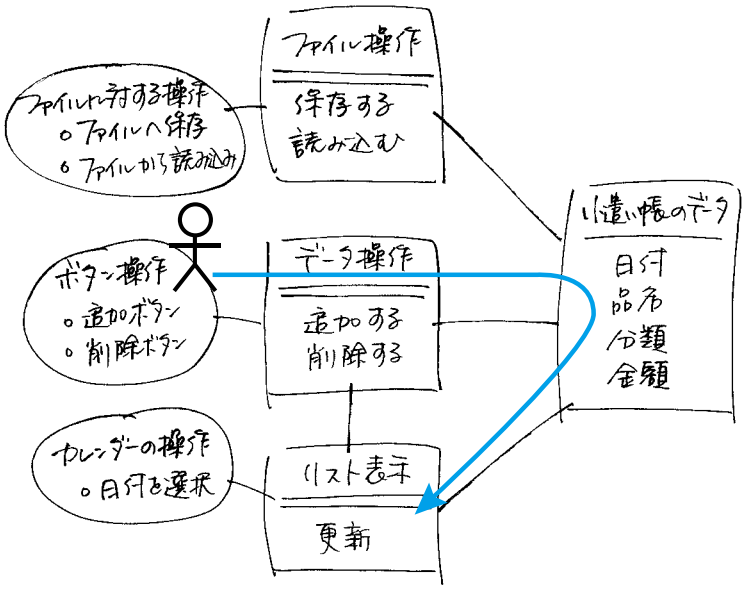
小遣い帳が今月のリストを表示するためのデータです。今月のリストには、日付、品名、分類、そして金額を表示する予定です。

●中央の3つの四角

動作とデータの部分をつなぐ「機能」の役目を果たします。

［追加］ボタンの処理の流れ

たとえば、［追加］ボタンをクリックして新しい項目を今月のリストに加える処理の流れを、青い線で次に示します。



■追加ボタンをクリックしたときの流れ

① 金額を入力して、［追加］ボタンをクリックする。
② データへの更新をするために、追加処理を呼び出す。
③ 追加処理によって、小遣い帳のデータに、日付、品名、分類、金額が新しく加わる。
④ 今月のリストの表示を更新して、新しい項目が追加された状態にする。

ここでの流れは、前の図と同じ流れになります。2つの図をよく見比べて、どのようなプログラムの動きになるのかひとつひとつ確認してください。

この段階でうまくいかない部分があれば、どんどん直していきます。また、小遣い帳に新しい機能を加えるときは、画面イメージや見取り図と照らし合わせ、矛盾がないように仕上げます。

参照

クラス図について
第7章のコラム

ヒント

スケッチUML

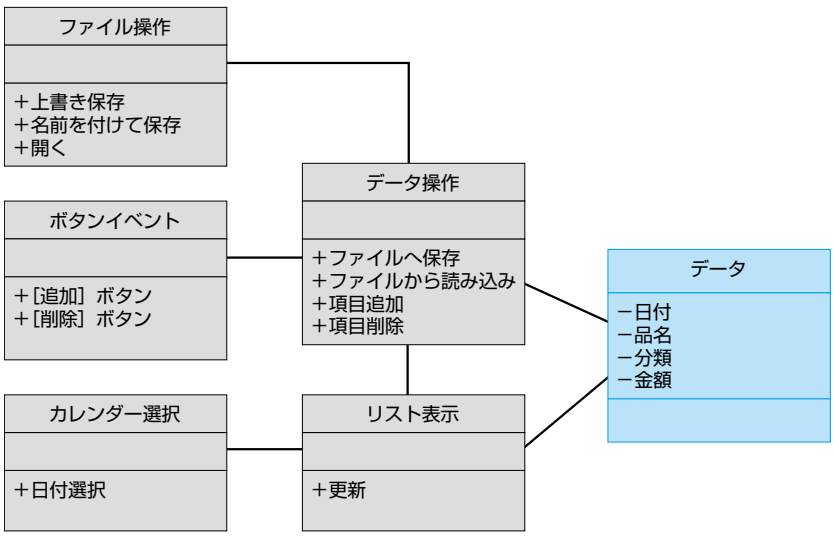
上の図のように四角い枠で操作をまとめた図を「クラス図」といいます。クラス図はUMLの一種で、クラス設計を行うときに使われる図です。
最近はクラス図やその他のUMLを記述するツールもたくさんありますが、UMLの規則に束縛されて少々書きにくいこともあります。また、試行錯誤しながら複数人で議論を行う場合などは、ホワイトボードや紙に手書きでUMLを書いたほうが手早くできます。
このように、あまりUMLの規則を重視せず、主要な部分だけを簡単に書いていくUMLを「スケッチUML」といいます。スケッチUMLで記述するクラス図では、クラス間の複雑な関係（多重度など）は書かず、風景をスケッチするようにクラス図を組み立てていきます。

6 HomeBankの見取り図

最後に小遣い帳の見取り図の完成版を示します。プログラミングを行う場合、できあがるアプリケーションの全体像を想像することがポイントとなります。全体像を図に表してみましょう。本書で作成するHomeBankアプリケーションは、この章で作成した見取り図を使用してプログラミングを行います。この先の章でプログラムの内容がわからなくなったら、再びこの章に戻って見取り図を確認してください。

見取り図

いつまでも「小遣い帳」という呼び名ではわかりにくいので、正式な名前を決めましょう。ここでは家計簿の意味もあるので、「HomeBank」という名前にします。



■HomeBankの見取り図

これが、HomeBankアプリケーションの見取り図です。このような図は、正式には「クラス図」という名称で呼ばれます。

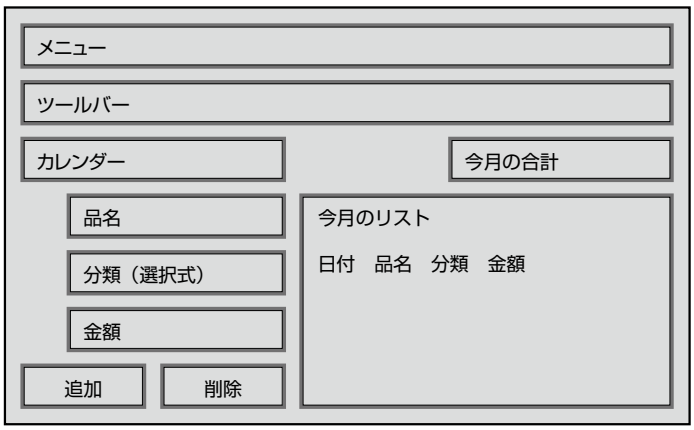
クラスについては、後の章で詳しく説明しますが、ここでは図に描かれているひとつひとつの四角が「クラス」と呼ばれる機能のまとまりである、ということを覚えておいてください。それぞれの四角（クラス）を結ぶ直線は、クラスの関連を表しています。先に描いた図のように、それぞれの動作を指で追ってみることをお勧めします。

ヒント

クラス図

クラス図は、UMLというオブジェクト指向のモデル表記法で定義されています。UMLについては、第7章のコラムを参照してください。

画面イメージ



■画面イメージ

画面イメージは、上の図のようになります。漠然とした画面イメージをわかりやすくするために、コメントを書いてもよいでしょう。詳細な画面は次の章で統合開発環境を使用しながら作成します。そのための下書きにする気持ちで、画面イメージを整理しておきます。

今回のHomeBankアプリケーションでは、メニューやツールバーを使用します。しかし、画面イメージに書き込むと煩雑になってしまうため、次のように別表にしました。メニューの名前と、その機能を簡単に書いておきます。

また、今月のリストに表示する項目も表にしてあります。このように、図示しにくいものは画面イメージを補足する形でまとめるとよいでしょう。

■メニューの名前とその機能

[ファイル] メニュー	機能
[開く]	ファイルからデータを読み込む
[上書き保存]	ファイルにデータを保存する
[名前を付けて保存]	ファイル名を付けてデータを保存する
[閉じる]	アプリケーションを終了する

■リストの表示項目

項目名	内容
日付	使用した年月日
品名	購入した対象
分類	分類の名称
金額	使用した金額

コラム

なぜ設計が必要なのか

設計がないと作業が分担できない

まず設計書がない状態を考えてみましょう。

グループで作業をしようとするときに、設計書がないとアプリケーションのイメージは設計者の頭の中にしかありません。そうなると、何をするにもいちいち設計者に問い合わせないと、詳しい機能を知ることができません。また、割り振られた作業量を知りたくても、その基準がわかりません。

このため、設計者が頭の中に持っているものを書き出して、皆がわかるようにする必要があります。これが設計書となります。

設計書があると早く終わる

設計書を作るのに時間がかかると言う人が多いようです。それは、いきなりWordやExcelなどを使い、初めからデジタル形式で書こうとするから時間がかかるのです。最初は無理をせず、イメージが固まるまでは紙やホワイトボードを使うのはどうでしょうか。

設計書の目的は、皆でイメージを共有することです。紙の上に鉛筆やボールペンで書いてもよいと思いませんか。またホワイトボードを活用すると、よりイメージの共有が活発になります。どちらの場合も、コピーを残すことを忘れないようにしましょう。

このように、手書きを活用したほうが、結果的にアプリケーションの設計や構築は早く終わることが多いと思います。デジタル形式のデータをあれこれといじって体裁を整えようと悩む時間を減らし、もっと有効活用する方法を考えましょう。

変更があったとき無駄な作業を減らせる

設計書がない、もしくは変更点をきちんと管理していない場合は、仕様の変更があったときに、その変更点がどんな作業を生むかわかりません。それがコードの追加なのか、削除なのか、修正なのかが判断できず、結局どの場合も追加作業になってしまうようなこともあります。

変更点がきちんと管理されている設計書があれば、このようなことは防げます。

完成したアプリケーションの判断基準になる

設計書がない場合、アプリケーションができあがっても、それが設計者の定義したとおりにできているかどうかを判断することができません。顧客から「要求したものと違う」と言われても、根拠がないので反論することもできません。

設計書があれば、設計者が定義したとおりに作成されたかどうか判断できます。また、顧客から要求したものと違うと言われても、それが要求の解釈の違いによるものか、あるいは追加作業によるものか、またはそれ以外のものかが判断できます。

第6章 イベントへの対応

1 ボタンのクリックイベント
2 コントロールからのテキストの取得
3 リストボックスへの項目の追加
4 リストボックスからの項目の削除
5 メッセージボックスによる問い合わせ
6 カレンダーの日付変更イベント

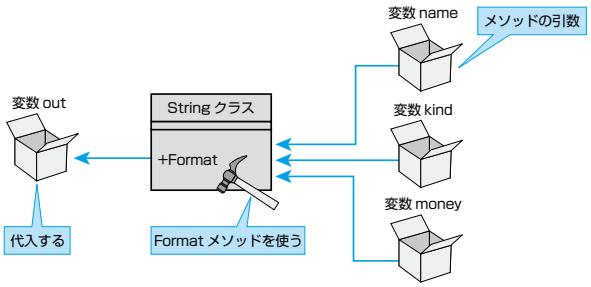
この章では、HomeBankプロジェクトにボタンをクリックしたときのプログラムの処理を実装します。プログラミングは、「習うより慣れろ」と言われます。どんどんプログラミングの世界へ踏み込んでいきましょう。

この章で学習する内容と身に付くテクニック

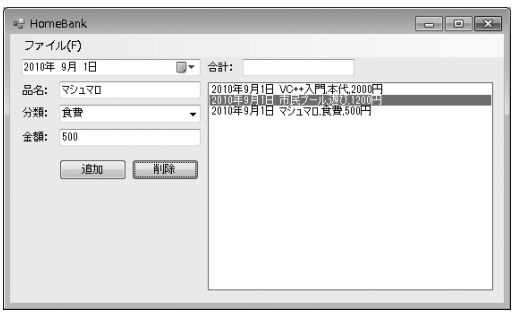
この章では、[追加] ボタンをクリックしたときなどのような、コントロールのイベントを中心に学習します。主な学習内容は次のとおりです。

- **ボタンのクリックイベント**
- **文字列の扱い方**
- **リストボックスの操作**
- **確認メッセージによる処理の問い合わせ**
- **カレンダーの日付変更イベント**

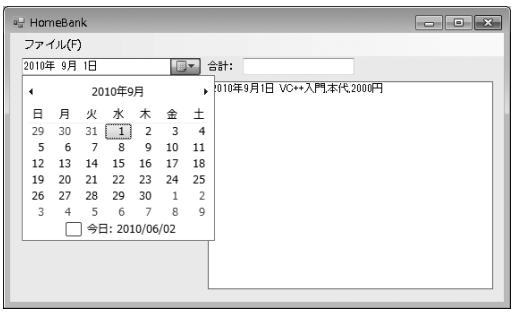
STEP 1 [追加] ボタンがクリックされたときに、リストボックスへ項目を追加します。この文字列を編集するためにStringクラスを使用します。



STEP 2 [削除] ボタンがクリックされたときに、リストボックスで選択されている項目を削除します。このとき、リストボックスが空の場合や、項目が選択されていない場合も考慮します。



STEP 3 HomeBankアプリケーションで、日付が変更されたときに今月のリストが切り替わるようにします。その準備として、カレンダーの日付変更イベントを使用し、今月のリストをクリアする処理を加えます。



1 ボタンのクリックイベント

［追加］ボタンや［削除］ボタンをクリックしたときの処理を実装しましょう。第3章ではボタンをクリックしたときの動作を簡単に説明しましたが、ここではもう少し詳しく確認します。

クリックイベントとは

ボタンをクリックしたときには、イベントが発生します。これを「クリックイベント」といいます。HomeBankアプリケーションでは、［追加］ボタンがクリックされたときに、今月のリストに日付や品名などを含む1行を追加します。この動きは［追加］ボタンのクリックイベント内で処理します。

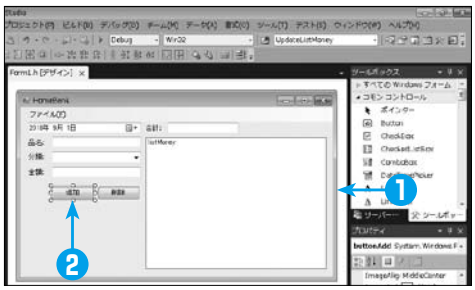
クリックイベントの作成

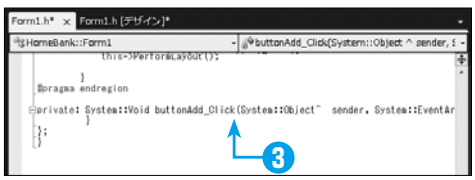
［追加］ボタンのクリックイベントを作成しましょう。前の章で作成したプロジェクトを引き続き使用するか、またはダウンロードしたサンプルプロジェクトを使用します。

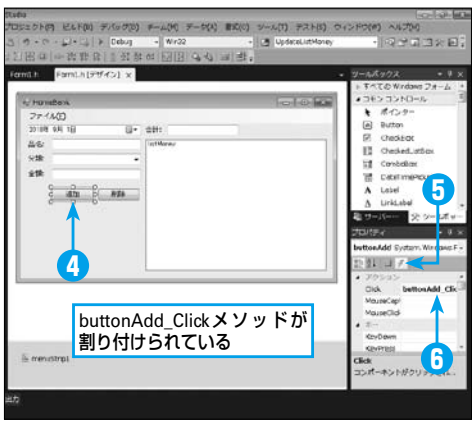
参考ファイル

￥第06章￥06-00￥HomeBank￥HomeBank.sln

1. フォームデザイナーにForm1フォームを表示する。
2. ［追加］ボタン（buttonAdd）をダブルクリックする。
   - コードエディターにForm1.hファイルが表示される。
3. buttonAdd_Clickメソッドが作成されたことを確認する。
4. フォームデザイナーに切り替え、［追加］ボタンが選択されていることを確認する。
5. プロパティウィンドウで、［イベント］ボタンをクリックする。
   - イベント一覧に切り替わる。
6. Clickイベントに「buttonAdd_Click」と表示されていることを確認する。







クリックイベントの記述

［追加］ボタンをクリックしたときのイベントは、buttonAdd_Clickメソッドに割り付けられます。このメソッドに、HomeBankアプリケーションでの［追加］ボタンの処理（日付や品名などをリストに追加）を記述します。ここではまず、［追加］ボタンをクリックすると、buttonAdd_Clickメソッドが呼び出されることを確認しましょう。Debug::WriteLineメソッドを使用して、出力ウィンドウにデバッグメッセージを表示します。

❶ フォームデザイナーで、［追加］ボタン（buttonAdd）をダブルクリックする。

▶ コードエディターに切り替わり、buttonAdd_Clickメソッドが表示される。

❷ buttonAdd_Clickメソッドに次のコードを記述する（色文字部分）。

```
private: System::Void buttonAdd_Click(System::Object^  sender, ➔
    System::EventArgs^  e) {
            System::Diagnostics::Debug::WriteLine("追加ボタンをクリックしました");
        }
```

❸ ［ビルド］メニューの［HomeBankのビルド］をクリックする。

▶ 問題なくビルドされることを確認する。

動作の確認

プログラムを実行し、［追加］ボタンをクリックするとデバッグメッセージが表示されることを確認しましょう。

❶ ［標準］ツールバーの［デバッグ開始］ボタンをクリックし、プログラムを実行する。

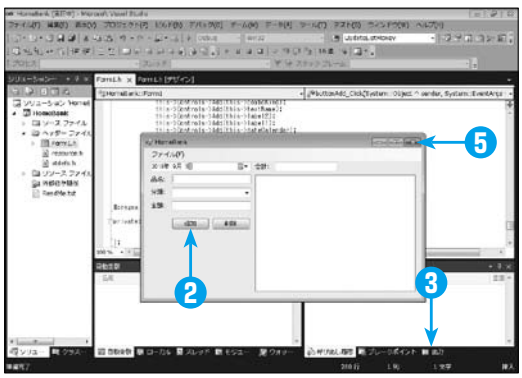
❷ HomeBankアプリケーションの［追加］ボタンをクリックする。

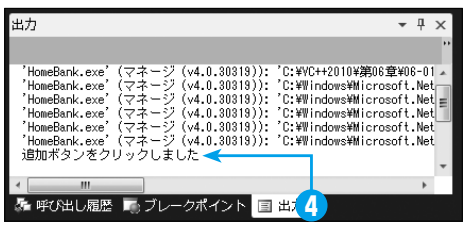
❸ 統合開発環境の［出力］タブをクリックする。

❹ 出力ウィンドウに「追加ボタンをクリックしました」というデバッグメッセージが表示されていることを確認する。

❺ HomeBankアプリケーションの閉じるボタンをクリックする。

▶ プログラムが終了し、統合開発環境に戻る。





名前空間とusing namespace

Debug::WriteLineメソッドを使用して、出力ウィンドウにメッセージを表示することができました。このメソッドを使用すると、［追加］ボタンや［削除］ボタンをクリックしたときの動きや、HomeBankアプリケーションの内部の動きを調べることができます。

今回の「System::Diagnostics::Debug::WriteLine(...)」のような長い記述方法を「完全修飾名」といいます。この方法で何度もコードを記述するのは面倒な作業です。この問題は、using名前空間ディレクティブを使用することで解決できます。それには、「using namespace」を記述します。

1
```
System::Diagnostics::Debug::WriteLine("追加ボタンをクリックしました");
```

2
```
using namespace System::Diagnostics;
Debug::WriteLine("追加ボタンをクリックしました");
```

この2つのコードは、同じ動きになります。2では、「using namespace」の後に「System::Diagnostics」という名前空間を記述して、Debug::WriteLineメソッドの修飾名を省略できるようにしています。Visual C++で作成したWindowsアプリケーションでは、多くのusing namespaceが使われています。

using namespaceには、完全修飾名を毎回記述する手間を省くという利点以外にも、明示的にusing namespaceを記述することによって、どの名前空間を使い、どのクラスライブラリを利用しているのかがわかるという副次的な利点もあります。

ヒント

クリックイベントのメソッド名

クリックイベントのメソッド名は、ボタンをダブルクリックしたときに自動的に作成されます。Visual C++では、既定で＜コントロール名＞＋アンダースコア（_）＋「Click」となります。
この他に、イベントのメソッド名を自由に設定することもできます。また、プロパティウィンドウでClickイベントの▼をクリックし、一覧から既存のメソッドを選択することもできます。

メソッドの呼び出しの確認方法

第3章では、ボタンをクリックしたときにMessageBox::Showメソッドを使用して、メッセージボックスを表示しました。ここでは、Debug::WriteLineメソッドを使用して［追加］ボタンをクリックしたときのメソッドの呼び出しを確認しています。どちらもボタンをクリックしたときの動作をメッセージとして表示する方法です。

名前空間

「System::Diagnostics」の部分を「名前空間」といいます。Debug::WriteLineメソッドは、「System::Diagnostics名前空間にあるDebugクラスのメソッド」ということです。名前空間によって.NET Frameworkの豊富なクラスライブラリを区別しています。

using namespaceの記述場所

using namespaceの記述場所は、省略形を使用する前の行であればどこに記述してもかまいません。正確には、ブロック（{ }で囲まれた範囲）により有効範囲が制限されます。
通常は、ファイルの先頭部分にusing namespaceを記述しておきます。この場所に記述すると、ファイル内のすべてのクラスやメソッドで使うことができます。

using namespaceの使用

実際にusing namespaceを使用してbuttonAdd_Clickメソッドを書き換えましょう。

❶ コードエディターで、Form1.hファイルの先頭部分に次のコードを記述する（色文字部分）。

```
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
    using namespace System::Diagnostics;
```

❷ buttonAdd_ClickメソッドのDebug::WriteLineメソッドの呼び出し部分から、先ほど記述した「System::Diagnostics::」の部分を削除し、次のコードに書き換える（色文字部分）。

```
private: System::Void buttonAdd_Click(System::Object^  sender, ➔
    System::EventArgs^  e) {
             Debug::WriteLine("追加ボタンをクリックしました");
         }
```

❸ ［ビルド］メニューの［HomeBankのビルド］をクリックする。

➡ 問題なくビルドされることを確認する。

動作の確認

using namespaceを使用する前と動作が変わらないことを確認しましょう。

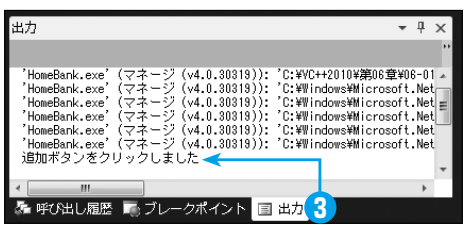
❶ ［標準］ツールバーの［デバッグ開始］ボタンをクリックし、プログラムを実行する。

❷ HomeBankアプリケーションの［追加］ボタンをクリックする。

❸ 統合開発環境の出力ウィンドウにデバッグメッセージが表示されることを確認する。

❹ 閉じるボタンをクリックする。

➡ プログラムが終了し、統合開発環境に戻る。



2 コントロールからのテキストの取得

テキストボックスやコンボボックスの値を取り出して、プログラム内で利用する方法を確認します。

品名、分類、金額の取得

フォームには、2つのテキストボックス（品名と金額）と1つのコンボボックス（分類）があります。このコントロールに表示されている値（文字列）を取得し、出力ウィンドウに表示しましょう。

1 コードエディターにForm1.hファイルを表示する。

2 buttonAdd_Clickメソッドに次のコードを記述する（色文字部分）。

```
private: System::Void buttonAdd_Click(System::Object^  sender, ➔
    System::EventArgs^  e) {
            Debug::WriteLine("追加ボタンをクリックしました");
            // 品名、分類、金額を取得する
            Debug::WriteLine( textName->Text );   ←1
            Debug::WriteLine( comboKind->Text );
            Debug::WriteLine( textMoney->Text );
        }
```

3 [ビルド] メニューの [HomeBankのビルド] をクリックする。

➧ 問題なくビルドされることを確認する。

コードの解説

1
```
            Debug::WriteLine( textName->Text );
```

コントロールから文字列を取り出すにはTextプロパティを使用します。たとえば、品名を表すテキストボックスのオブジェクト名は、「textName」です。このオブジェクトのTextプロパティの値を参照するには、「textName->Text」と記述します。

ヒント

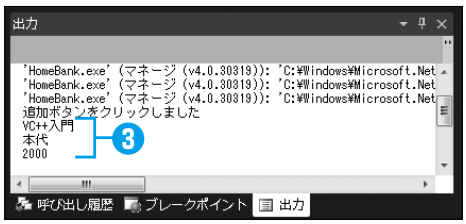
文字列とは

複数の文字が集まったものを文字列といいます。文字列からは、1文字ずつ部分的に取り出すことができます。また、複数の文字列を1つに結合することもできます。

動作の確認

コントロールに入力した文字列が出力ウィンドウに表示されることを確認しましょう。

1. [標準] ツールバーの [デバッグ開始] ボタンをクリックし、プログラムを実行する。
2. [品名] ボックス、[分類] ボックス、[金額] ボックスに右の図のように入力し、[追加] ボタンをクリックする。
3. 統合開発環境の出力ウィンドウに各コントロールの文字列が出力されることを確認する。
4. 閉じるボタンをクリックする。
   - プログラムが終了し、統合開発環境に戻る。



変数への値の代入

先ほどはTextプロパティを直接扱いましたが、今度は変数を使用して、品名、分類、金額を出力ウィンドウに表示します。変数を使用することによって、値に意味を持たせ、コードをわかりやすくします。

1. コードエディターにForm1.hファイルが表示されていることを確認する。
2. buttonAdd_Clickメソッドを次のように書き換える（色文字部分）。

```
private: System::Void buttonAdd_Click(System::Object^  sender, ➔
    System::EventArgs^  e) {
            Debug::WriteLine("追加ボタンをクリックしました");
            // 品名、分類、金額を取得する
            String^ name = textName->Text;
            String^ kind = comboKind->Text;
            String^ money = textMoney->Text;
            Debug::WriteLine( name );
            Debug::WriteLine( kind );
            Debug::WriteLine( money );
        }
```

1
2

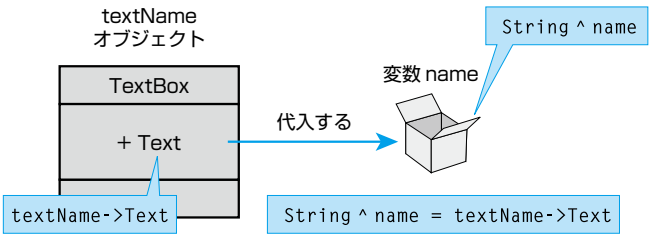
3. [ビルド] メニューの [HomeBankのビルド] をクリックする。
   - 問題なくビルドされることを確認する。

コードの解説

1

```
String^ name = textName->Text;
```

textNameオブジェクトのTextプロパティから取り出した値をString型の変数nameに代入します。変数nameの前にあるハット（カレット、キャレットとも呼ぶ。以降「ハット」と表記）(^) はハンドルを示す記号です。ハンドルについては、この後で説明します。



変数とは、値を入れる箱のようなものです。Stringは文字列を扱うためのクラスです。nameは変数の名前で、**1**の式は、変数nameという箱の中に文字列を代入する、という意味になります。

2

```
Debug::WriteLine( name );
```

コントロールの値が代入された変数nameを使用し、Debug::WriteLineメソッドで出力ウィンドウに書き出します。

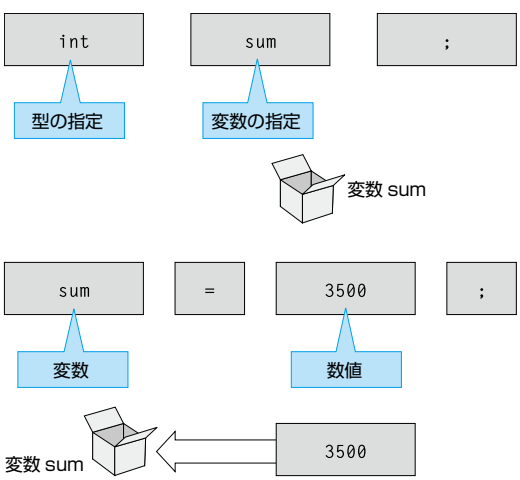
変数とハンドル

変数は、アプリケーションの中で値を記憶しておく領域です。変数の値は、直接示す方法と、ハンドルを使用して間接的に示す方法とがあります。変数には自由に名前を付けることができます。また、変数は「データ型」を持ち、どのような形式でデータ（値）を記憶しているのかを表します。

```
int sum;
```

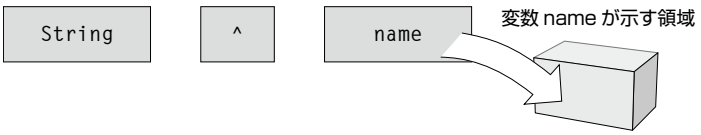
この例では、「sum」という変数をint型として定義しています。int型は数値を保存するための型です。変数sumには、数値を代入することができます。たとえば、変数sumに3500という値を代入するには、代入するための記号「=」を使用して、「sum = 3500;」と記述します。

変数sumが持つ領域に、3500という数値を保存するイメージです。



```
String^ name;
```

このコードは、ハンドルを使用した例です。ハンドルとは、「あらかじめ記憶してある領域を指し示すもの」です。この例では、String型のハンドルを「name」という名前で宣言しています。ハンドルの記号はハット（^）を使用します。



参照

ハンドルについて

第9章のコラム

構文 変数の定義

```
<型> <変数名> ;                          // 変数の定義
<型>^ <変数名> ;                         // ハンドルの定義
<型> <変数名> = <値> ;                   // 変数の定義と値の代入を同時に行う
<型>^ <変数名> = <オブジェクト> ;        // ハンドルの定義とオブジェクトの代入を同時に行う
```

データ型

Visual C++のデータ型には、数値を保存するためのint型や文字列のハンドルを示すString型以外にもさまざまな型があります。ここでは主な基本型を示します。

■基本のデータ型

型	説明
bool	ブール値（true/false）
char	8ビット符号付き整数
short	16ビット符号付き整数
int	32ビット符号付き整数
long	32ビット符号付き整数
float	単精度浮動小数点数
double	倍精度浮動小数点数
wchar_t	Unicode（16ビット）文字
Object^	オブジェクト型のハンドル
String^	文字列型のハンドル

ヒント

ローカル変数の利用

変数nameのようにメソッド内で定義される変数を「ローカル変数」と呼びます。ローカル変数の有効範囲（スコープ）は「{ }」で囲まれたブロックの範囲になります。ローカル変数にわかりやすい名前を付けることは、過剰なコメントを付けることよりも保守性を良くします。

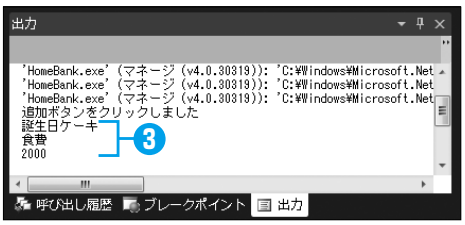
ローカルウィンドウ

ローカル変数は、デバッグ実行時にローカルウィンドウに表示されます。ローカルウィンドウでは、ブレークポイントでプログラムを一時的に停止しているときに、変数の値の確認や変更ができます。

動作の確認

コントロールに入力した文字列が、変数に代入され、出力ウィンドウに表示されることを確認しましょう。

1. ［標準］ツールバーの［デバッグ開始］ボタンをクリックし、プログラムを実行する。
2. ［品名］ボックス、［分類］ボックス、［金額］ボックスに右の図のように入力し、［追加］ボタンをクリックする。
3. 統合開発環境の出力ウィンドウに、各コントロールの文字列が出力されることを確認する。
4. 閉じるボタンをクリックする。
   - プログラムが終了し、統合開発環境に戻る。



1行にまとめた品名、分類、金額の表示

今度は出力ウィンドウに、品名、分類、金額を1行にまとめて表示します。リストボックスに1行追加するための準備です。文字列をフォーマット（整形）するには、String::Formatメソッドを使用します。

1. コードエディターにForm1.hファイルが表示されていることを確認する。
2. buttonAdd_Clickメソッドを次のコードに書き換える（色文字部分）。

```
private: System::Void buttonAdd_Click(System::Object^  sender, ➔
    System::EventArgs^  e) {
             Debug::WriteLine("追加ボタンをクリックしました");
             // 品名、分類、金額を取得する
             String^ name = textName->Text;
             String^ kind = comboKind->Text;
             String^ money = textMoney->Text;
             String^ out;
             out = String::Format("品名:{0} 分類:{1} 金額:{2}", name, kind, ➔
                 money );
             Debug::WriteLine( out );
         }
```

1

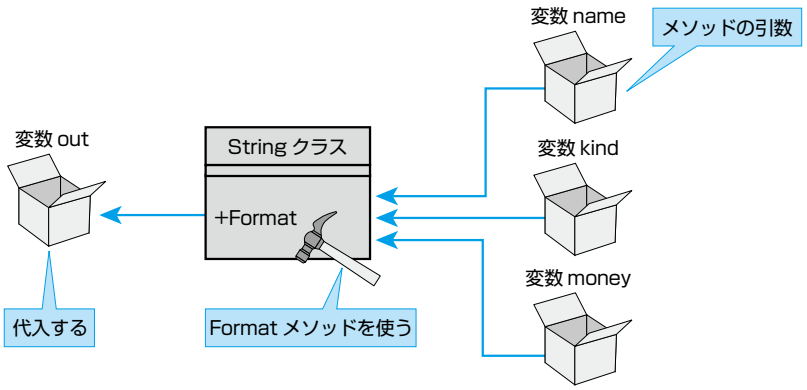
3. ［ビルド］メニューの［HomeBankのビルド］を選択する。
   - 問題なくビルドされることを確認する。

コードの解説

1

```
String^ out;
out = String::Format("品名:{0} 分類:{1} 金額:{2}", name, kind, money );
```

String::Formatメソッドは、文字列をフォーマットするためのメソッドです。String::Formatメソッドには、フォーマット用の文字列と、それぞれの引数を指定します。ここでは、フォーマット用として「"品名:{0} 分類:{1} 金額:{2}"」と、3つの変数（name、kind、money）を指定しています。それぞれの変数の値が、中かっこ（{}）で指定された番号の部分と置き換わります。



■**String::Formatメソッド**

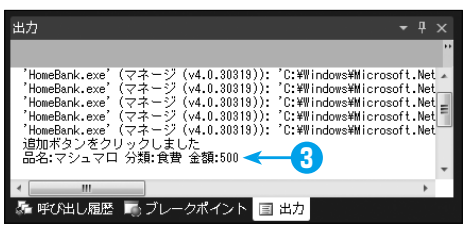
構文 String::Format

```
String^ String::Format( ＜フォーマット文字列＞, ＜変数＞, ... );
```

動作の確認

コントロールに入力した文字列が、1行にまとめて出力ウィンドウに表示されることを確認しましょう。

1. [標準]ツールバーの[デバッグ開始]ボタンをクリックし、プログラムを実行する。
2. [品名]ボックス、[分類]ボックス、[金額]ボックスに右の図のように入力し、[追加]ボタンをクリックする。
3. 統合開発環境の出力ウィンドウに、フォーマットされた文字列が表示されることを確認する。
4. 閉じるボタンをクリックする。
   - プログラムが終了し、統合開発環境に戻る。



3 リストボックスへの項目の追加

［追加］ボタンをクリックしたときに、リストボックスに1行追加されるようにプログラムを変更しましょう。

項目の追加

これまでは出力ウィンドウに文字列を出力しましたが、今度はリストボックスに表示しましょう。品名、分類、金額に加え、日付も入れます。この日付はカレンダー（dateCalendar）から取得します。また、複数の文字列を1つに結合するために結合演算子（+）を使用します。

❶ コードエディターにForm1.hファイルを表示する。

❷ buttonAdd_Clickメソッドの「Debug::WriteLine();」行を削除し、次のコードを記述する（色文字部分）。

```
private: System::Void buttonAdd_Click(System::Object^  sender, ➔
    System::EventArgs^  e) {
            // 品名、分類、金額を取得する
            String^ name = textName->Text;
            String^ kind = comboKind->Text;
            String^ money = textMoney->Text;
            // 日付を取得する
            String^ date = dateCalendar->Text;    ←1
            // フォーマット
            String^ out = String::Format("{0},{1},{2}円", name, kind, money );
            out = date + " " + out;    ←2
            // リストに追加する
            listMoney->Items->Add( out );    ←3
        }
```

❸ ［ビルド］メニューの［HomeBankのビルド］をクリックする。

➧ 問題なくビルドされることを確認する。

コードの解説

1

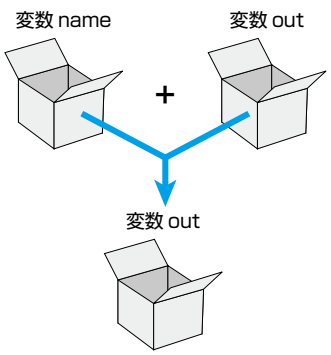
```
String^ date = dateCalendar->Text;
```

日付を取得するためには、カレンダーコントロールであるdateCalendarオブジェクトのTextプロパティを使用します。他のコントロールと同様に、Textプロパティから値を取得するには「dateCalendar->Text」と記述します。これを変数dateに代入します。

2

```
out = date + " " + out;
```

結合演算子（+）は、文字列を結合するための演算子です。プラス記号（+）を使い文字列を連結します。ここでは変数dateと半角のスペース（" "で表す）と変数outを連結し、その結果を変数outに代入しています。この結果、日付の文字列「2010年9月1日」に「VC++入門～」をつなげて「2010年9月1日VC++入門～」という1つの文字列が作成されます。



■結合演算子（+）

構文 結合演算子（+）

```
＜連結された文字列＞ = ＜文字列＞ + ＜文字列＞
```

3

```
listMoney->Items->Add( out );
```

リストボックスであるlistMoneyオブジェクトに1行追加しています。リストボックスに項目を追加するには、Itemsプロパティの Addメソッドを使用します。Itemsプロパティはリストボックスが持っているそれぞれの項目の集まりを表しています。この集まりを「コレクション」といいます。このコードは、リストボックスのコレクションに変数outを追加しています。

構文 リストボックスへの項目の追加

```
＜リストボックス名＞->Items->Add( ＜追加する文字列＞ );
```

参照

コレクションについて
第8章の1

動作の確認

各コントロールに入力した文字列が、リストボックスに追加されることを確認しましょう。

1. ［標準］ツールバーの［デバッグ開始］ボタンをクリックし、プログラムを実行する。
2. ［品名］ボックス、［分類］ボックス、［金額］ボックスに右の図のように入力し、［追加］ボタンをクリックする。
3. リストボックスに1行追加されたことを確認する。
4. 閉じるボタンをクリックする。
   - プログラムが終了し、統合開発環境に戻る。

ヒント

文字列操作のメソッド

文字列を扱うには、Stringクラスを使用します。文字列操作のためのメソッドをいくつか紹介しましょう。

■前後の空白を削除する
String::Trim()

■部分文字列を返す
String::Substring(＜開始位置＞)
String::Substring(＜開始位置＞, ＜文字数＞)

■文字列を置き換える
String::Replace(＜検索文字列＞,
＜置換文字列＞)

■大文字または小文字に変換する
String::ToUpper()
String::ToLower()