Chapter 1 C++による大規模開発

1.1 デバッガとは

デバッガは、プログラムの動作を解析するためのツールで、プログラムの実行中に問題を特定し、修正することができます。デバッガを使用すると、プログラムを一時停止させることができます。その際、特定の行に到達したときや、特定の条件が満たされたときにプログラムを一時停止するように指示することができます。
例えば、プログラムがエラーで停止している場合、デバッガを使用してエラーが発生した場所を特定することができます。また、プログラムの変数やオブジェクトの値を調べたり、関数がどのように呼び出されたかを追跡したりすることができます。
デバッガは、コードを理解するためのステップバイステップの手順を提供します。プログラムを実行して、それぞれのステップで何が起こっているかを確認することができます。これにより、プログラムの振る舞いをより深く理解することができます。
デバッガは、プログラムの開発に不可欠なツールです。エラーを特定し、修正するためには、プログラムの動作を理解することが必要です。デバッガを使用することで、プログラムの動作をより詳細に理解することができ、開発プロセスを効率的に進めることができます。

1.2 大規模開発とデバッガ

大規模開発において、デバッガは非常に重要なツールとなります。大規模なプログラムを開発する場合、特に複数の人間が開発に関わっているとき、バグが発生する可能性が高くなります。このような大規模な開発において、バグのの発生場所を特定することは困難であり、バグの修正に多くの時間がかかることがあります。
このような場合、デバッガを使用することで、バグが発生した場所を特定することができます。
もし、このような大規模開発でデバッガが使えないとバグの特定と修正にかかる時間が大幅に増加し、プログラム開発の効率が低下することが予想されます。もしデバッガが使えない場合には、代替手段として、ログ出力や単体テストなどを利用することで、バグの特定と修正に時間をかけることなくプログラム開発を進めることができます。ただし、これらの手法はデバッガに比べると効率が低く、開発にかかる時間が増加する可能性があるため、デバッガを使用できる環境を整えることが望ましいといえます。

1.3 Visual Studio(IDE)とデバッガ

Visual Studioは、Microsoftが提供する統合開発環境(IDE)です。統合開発環境（IDE：Integrated Development Environment）は、プログラム開発を行う際に必要とされる様々なツールや機能を一つのソフトウェアパッケージにまとめたものです。主にソフトウェア開発者が使用し、プログラムの開発、編集、コンパイル、実行、テスト、デバッグなどの作業を効率的に行うことができます。

一般的なIDEには、以下のような機能があります。

1.3.1 ソースコードエディター

プログラムを作成するためのテキストエディターで、主要なプログラミング言語に対応しています。また、スペルチェック、コードの自動整形、コードの補完、ファイルの差分の表示などの機能があります。

1.3.2 コンパイラ、インタプリター、デバッガ

プログラムのコンパイル、実行、デバッグを行うためのツールです。これらのツールをIDEで統合することで、より簡単に開発を行うことができます。

1.3.3 ビルドツール、デプロイツール

プログラムのビルドやデプロイメントに必要なツールが統合されています。ビルドツールは、コードのコンパイル、リンク、パッケージ化を自動化するための機能があります。デプロイツールは、アプリケーションを開発環境から実行環境にデプロイするための機能があります。

1.3.4 プロジェクト管理機能

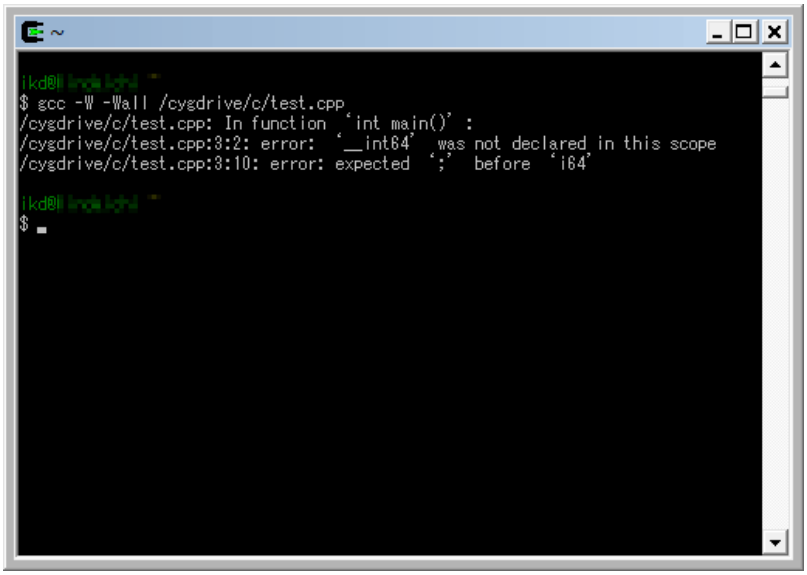
複数のファイルから構成されるプログラムを管理するための機能があります。プロジェクトファイルを作成し、ソースコードファイルやリソースファイルを追加したり、ファイルの結合、参照の解決、依存関係の解決などを自動化することができます。

1.3.5 バージョン管理ツールの統合

ソースコードのバージョン管理を行うためのツールがIDEに統合されている場合があります。これにより、複数人での開発時におけるソースコードの競合や衝突の解決、コードの変更履歴の確認、ロールバックなどが容易になります。

統合環境を利用することで、プログラムの開発プログラムの開発プロセスを効率化することができます。例えば、エディターにコードを書いてからコマンドラインでコンパイルする必要があります。(図1.1)

図1.1



また、コンパイラとエディタなどのツールが別々になっているため、別のウィンドウで作業する必要があったりするため、作業の流れが中断される可能性があります。一方、IDEを使うと、必要な作業がすべて統合されているため、作業の中断を最小限に抑えることができます。

> ただし、ツールを別々で使う場合に比べて、必ずしもIDEの方が優れているというわけではないので、そこな勘違いしないようにしてください。

また、IDEは多くの場合、自動化された作業フローを提供しています。例えば、プログラムのビルドとデプロイの自動化や、テストやデバッグのためのコードスニペットの提供などです。これらの自動化された作業フローにより、開発者はより多くの時間をコードの開発に費やすことができます。

IDEは、開発者が使用する言語やフレームワークに合わせて特化しています。Visual Studioは、.NET FrameworkやC#、Visual Basic、C++に最適化されています。Eclipseは、JavaやAndroidアプリ開発に最適化されています。そのため、開発者は、自分が使用する言語やフレームワークに最適化されたIDEを選ぶことができます。

統合環境は、開発者がプログラム開発に集中できるように設計されています。エディター、コンパイラ、デバッガ、ビルドツール、デプロイツール、プロジェクト管理機能、バージョン管理ツールなど、必要なツールを一つのソフトウェアパッケージにまとめて提供することで、開発者の作業効率を向上させます。また、特定の言語やフレームワークに特化しており、開発者が使用する言語やフレームワークに最適化された機能を提供します。 Visual Studioには、C++、C#、Visual Basicなどのプログラミング言語をサポートする開発ツールが含まれており、プログラムの作成、編集、ビルド、実行、デバッグなどの作業を統合的に行うことができます。

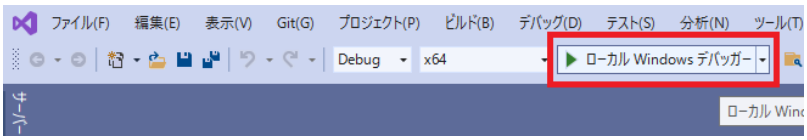
Visual Studioの中でも、デバッグ機能は非常に強力で、多くのデバッグ機能を提供しています。Visual Studioのデバッグ機能を使用することで、ブレークポイントの設定、ステップ実行、ウォッチウィンドウの使用、コールスタックの参照、変数の値の表示など、プログラムのデバッグに必要な機能を簡単に使用することができます。

1.4 Visual Studioでのデバッガの利用(基本)

1.4.1 デバッガありで実行(F5)

デバッガを使うためには、プログラムをデバッガありで実行する必要があります。
「デバッガありでの実行」は図1.2のボタンをクリックするか、ショートカットキーのF5を入力することで行えます。

図1.2



1.4.2 デバッガの停止(shift+F5)

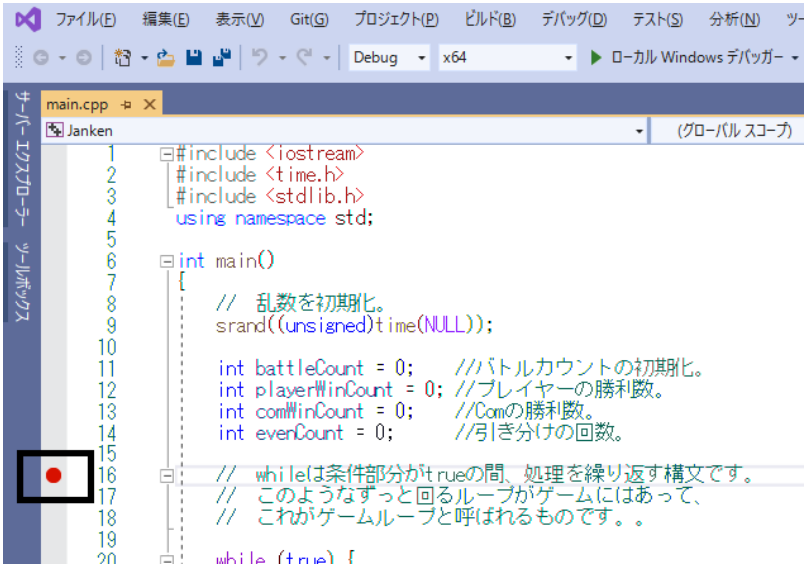
デバッガありで実行開始したプログラムはデバッガの停止ボタンを押すことで停止することができます(図1.3)。
デバッガの停止はショートカットキーのshift+F5でも停止することができます。

図1.3

1.4.3 ブレイクポイント(F9)

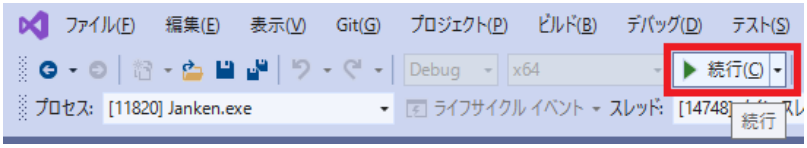
ブレイクポイントを設置することで実行中のプログラムを停止することができます。
プログラムの処理の流れの確認や、後述するウォッチ機能を利用しての変数の値などを行うときに頻繁に利用する機能です。
ブレイクポイントは図1.4のようにプログラムを停止させたい行をマウスでクリックすることで設置することができます。
また、ショートカットキーのF9を入力することでも設置することができます。

図1.4



ブレイクポイントの解除は、解除したいブレイクポイントをマウスでクリックするかF9を入力することで解除することができます。
また、ブレイクポイントで停止したプログラムを再開するには、図1.5の「続行」をクリックするか、ショートカットキーのF5を入力してください。

図1.5



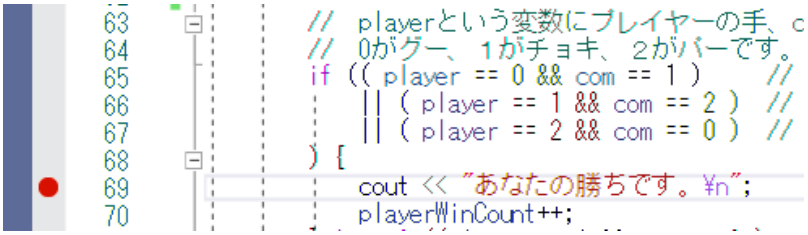
1.4.4【ハンズオン】じゃんけんに勝利したときif文の中にブレイクポイントを設置して見よう

じゃんけんゲームのプログラムでデバッガの機能を利用して、勝利したときのif文の処理が正しく実行されているか確認してみましょう。
Sample_01/Janken.slnを立ち上げてください。

step-1 ブレイクポイントの設置

main.cppの69行目にブレイクポイントを設置してください(図1.6)。

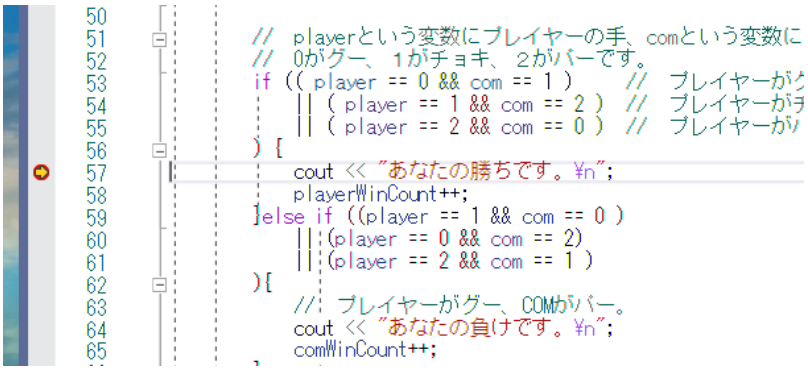
図1.6



step-2 デバッガありで実行

プログラムをデバッガありで実行して、じゃんけんゲームをプレイしてみてください。
ブレイクポイントがうまく設置できていると、じゃんけんに勝利したときに図1.7のようにプログラムが停止するようになります。

図1.7



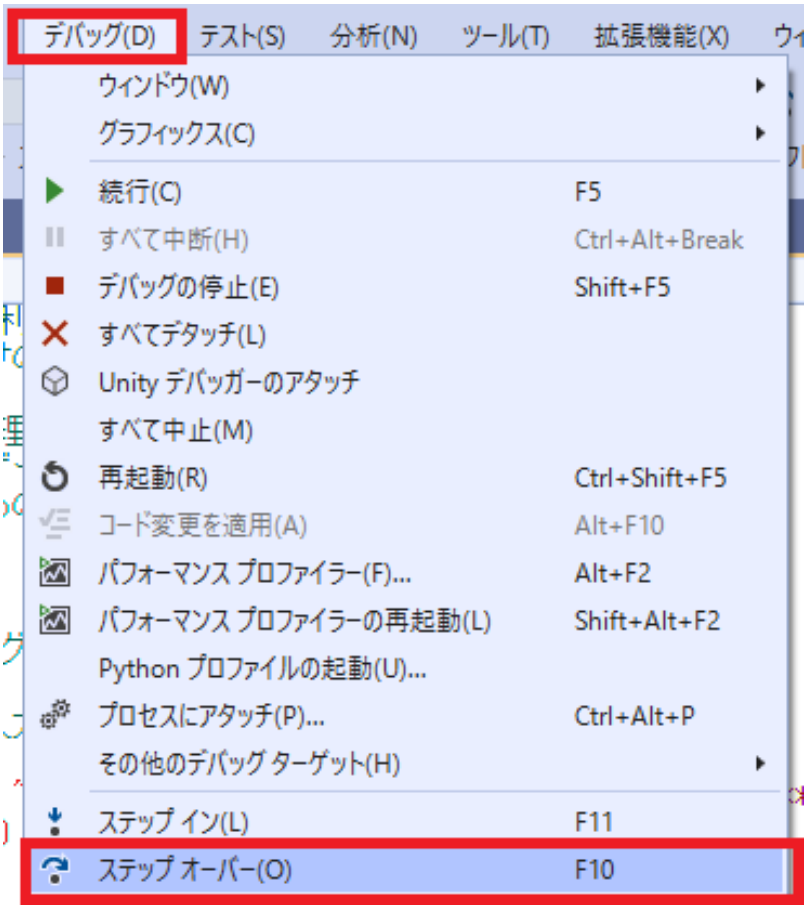
step-3 続行

続行ボタンを押してプログラムを続行してください。

1.4.5 ステップオーバー(F10)

ここからは、プログラムのステップを進めるための機能を見ていきます。
まずはステップオーバーです。
ステップオーバーはブレイクポイントで停止させたプログラムを１行進めることができます。
ステップオーバーはメニューの「デバッグ/ステップオーバー」で実行できます(図1.8)。ショートカットキーのF10でも実行できます。

図1.8



ステップオーバーは関数の中には入らずにステップを進めるため、ステップオーバーと呼ばれます。

1.4.6 ステップイン(F11)

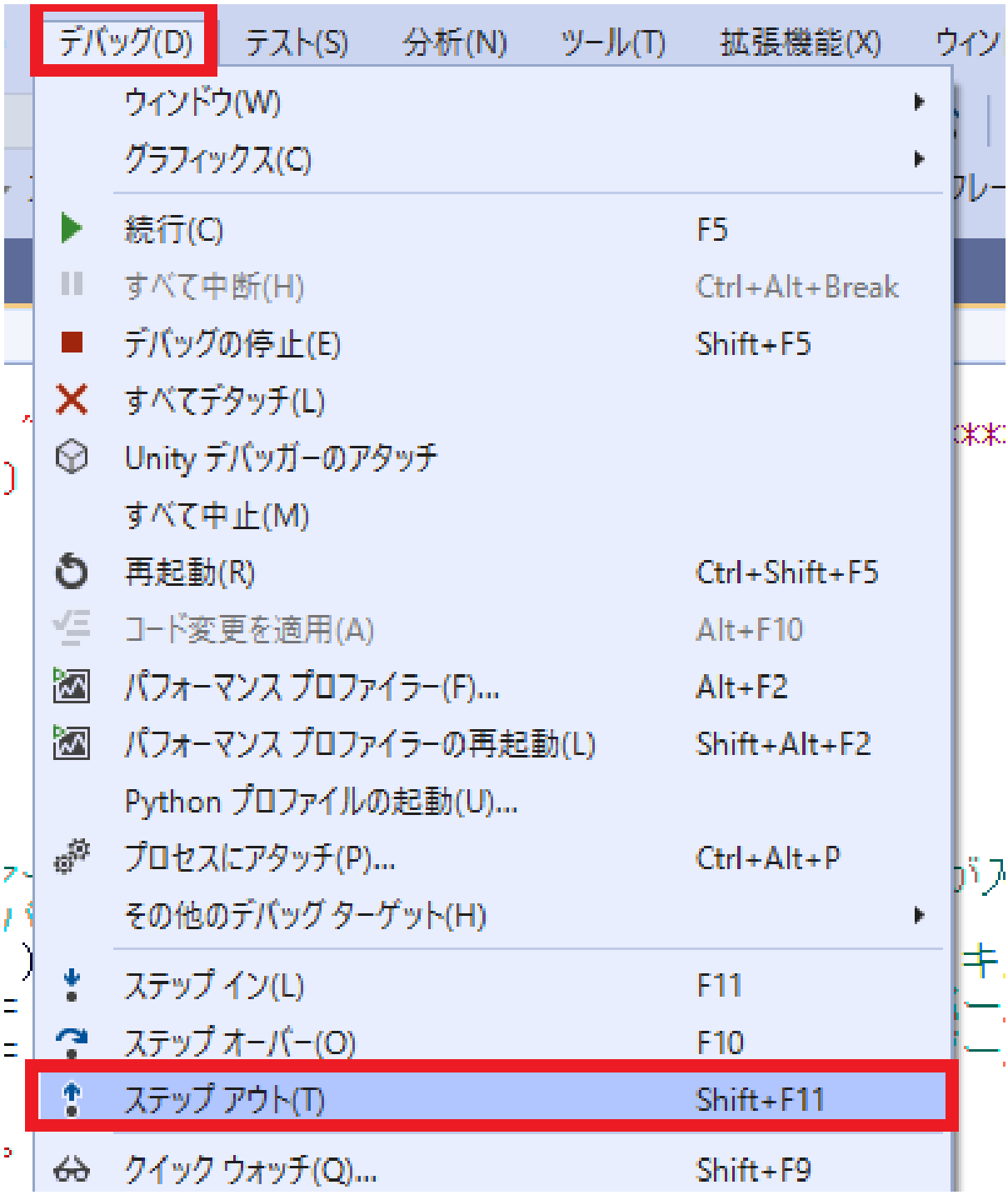
ステップインもプログラムのステップを１行進めるのですが、こちらはステップオーバーとは違い、関数の中に入ることができます。
ステップインはメニューの「デバッグ/ステップイン」で実行できます(図1.9)。
ショートカットキーのF11でも実行できます。

図1.9

1.4.7 ステップアアウト(Shift + F11)

ステップアウトは関数を抜けることができます。
ステップアウトはメニューの「デバッグ/ステップアウト」で実行できます(図1.10)。
ショートカットキーのshift + F11でも実行できます。

図1.10



1.4.8【ハンズオン】ステップオーバー/ステップイン/ステップアウトを試す

では、ハンズオンで各種ステップ実行を試してみましょう。
先ほどのハンズオンでじゃんけんに勝利した際にブレイクポイントを設置していると思いますので、
一旦そこのブレイクポイントでプログラムを停止させてからステップ実行をしていきましょう。
では、まずはプログラムをデバッガありで実行して、じゃんけんゲームをプレイして、69行目のブレイクポイントでプログラムを停止させてください。

step-1 ステップオーバーで61行目までプログラムを進める。

step-2 ステップインでDispComNoTe()の中に入る

step-3 ステップオーバーでDispComNoTe()の処理を23行目まで進める

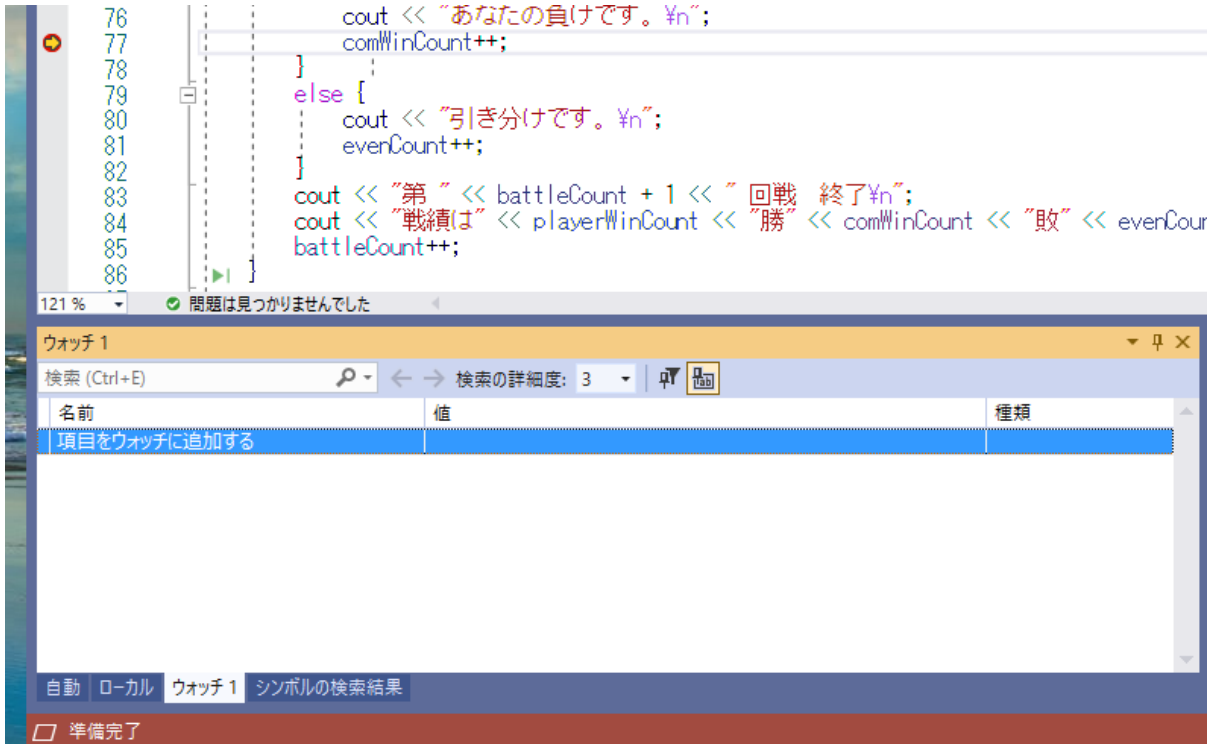
step-4 ステップアウトでDispComNoTe()から抜ける

1.4.9 ウォッチ

プログラム実行中に、プレイヤーの体力など、変数の値がどうなっているのか確認したい場合があります。
このようなときに使える機能がウォッチです。
ウォッチを使うためには、ブレイクポイントを設置してプログラムを停止させる必要があります。
プログラムが停止すると、その時点での変数の値を確認することができます。
ウォッチはメニューの「デバッグ/ウィンドウ/ウォッチ/ウォッチ1」から開くことができます(図1.11)。
ウォッチを開けると図1.12のように画面下部にウォッチ画面が表示されているはずです。

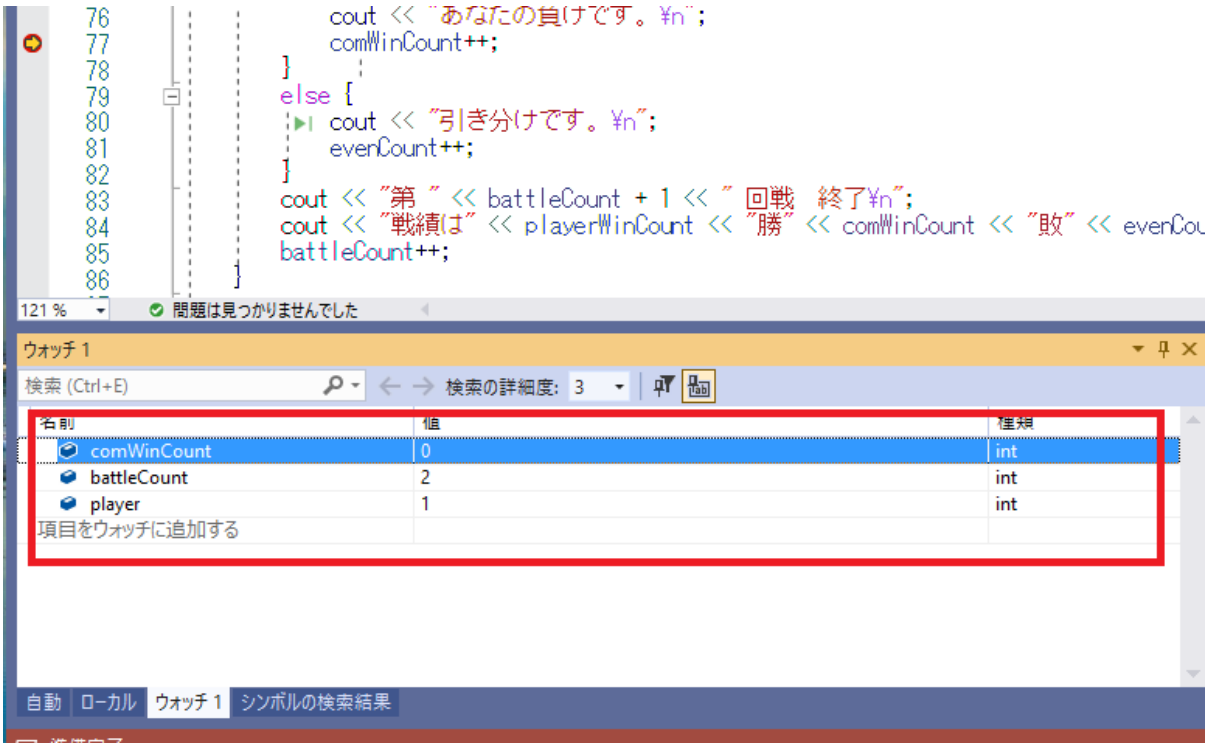
図1.11

図1.12



ウォッチウィンドウ値を確認したい変数を入力 or ドラッグアンドロップすると、変数の値を確認することができます(図1.13)。

図1.13



(注意：ウォッチウィンドウはデバッグありで実行中しか表示されないので注意してください。メニューにも出てきません。)

1.4.10【ハンズオン】ウォッチを使ってみる

では、ウォッチを使ってみましょう。
また、じゃんけんゲームをデバッガありで起動してプレイしてください。
するとじゃんけんで勝利したときにプログラムがブレイクポイントで停止すると思いますので、停止したら変数player、comをウォッチに追加して値を確認してください。

1.1.11 評価テスト-1

次の評価テストを行いなさい。

評価テストへジャンプ

1.1.12 実習課題-1

Question_01のじゃんけんゲームはいくつかプログラムの間違い、バグがあります。
デバッガの機能を活用して、バグの原因を見つけ出してバグを修正しなさい。

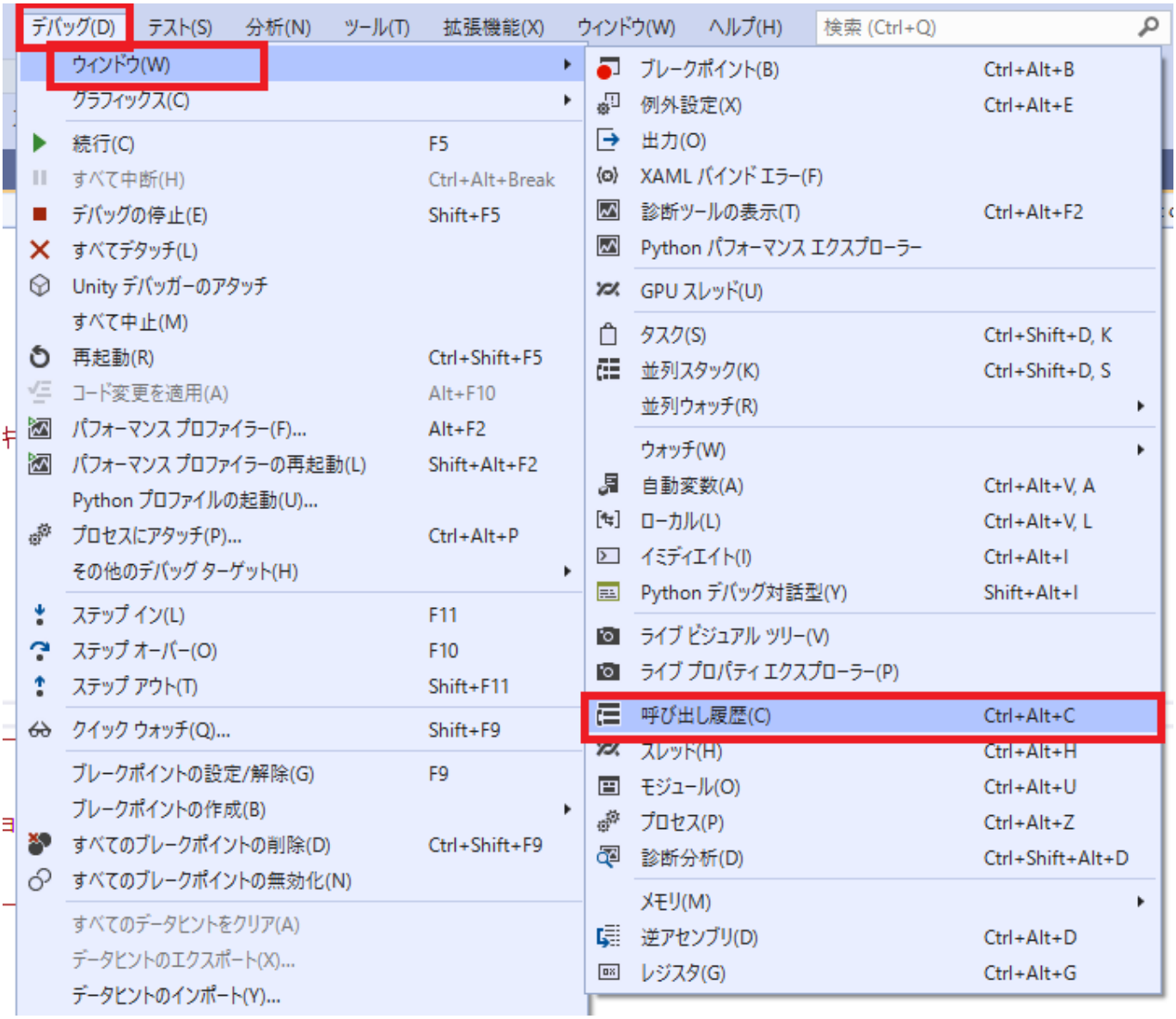
1.2 デバッガの利用(応用)

このチャプターでは、デバッガの利用をさらに見ていきましょう。

1.2.1 呼び出し履歴

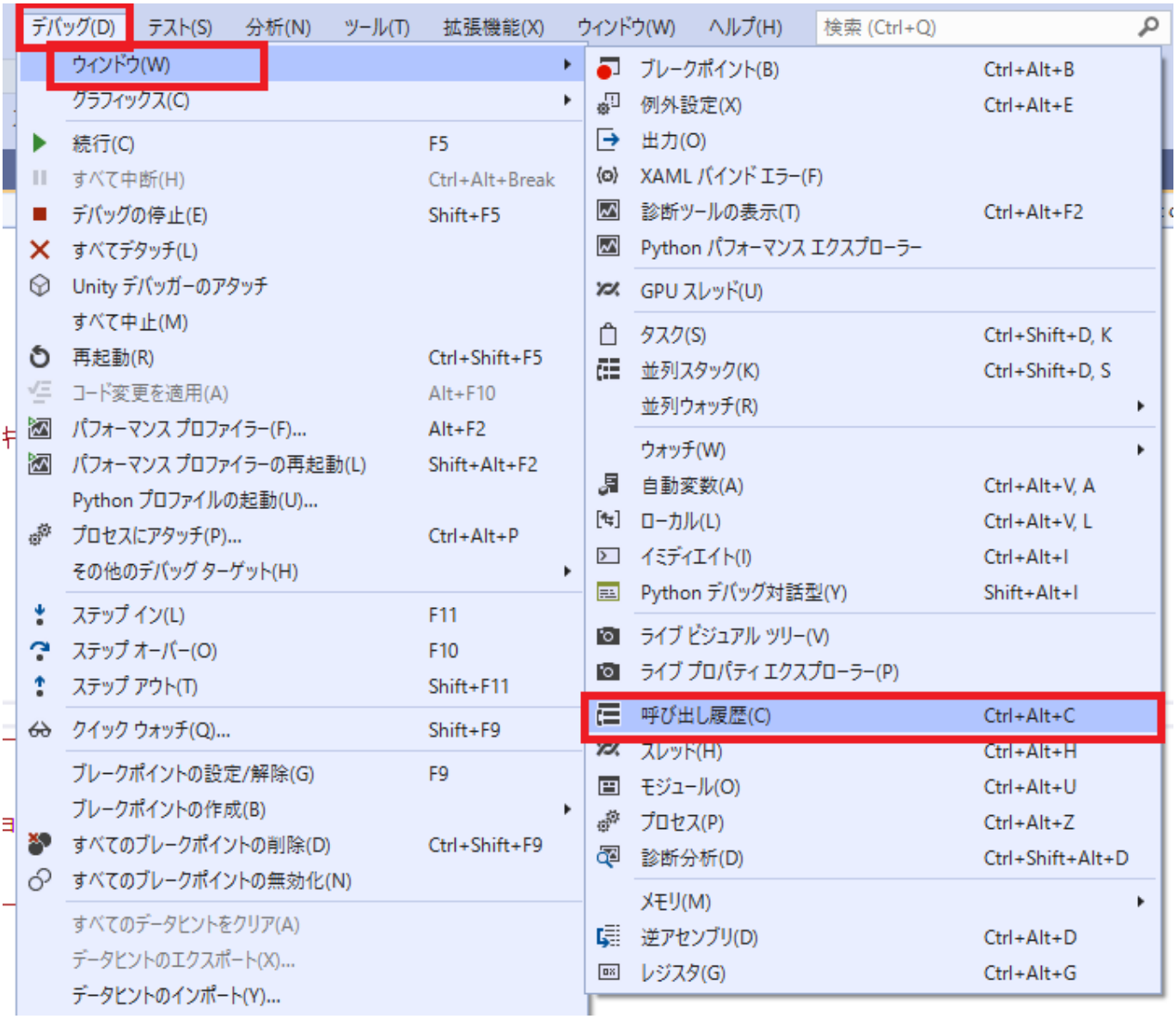
関数の呼び出し履歴を確認することができます。呼び出し履歴は、メニューの「デバッグ/ウィンドウ/呼び出し履歴」を選択すると開くことができます(図1.13)。

図1.13



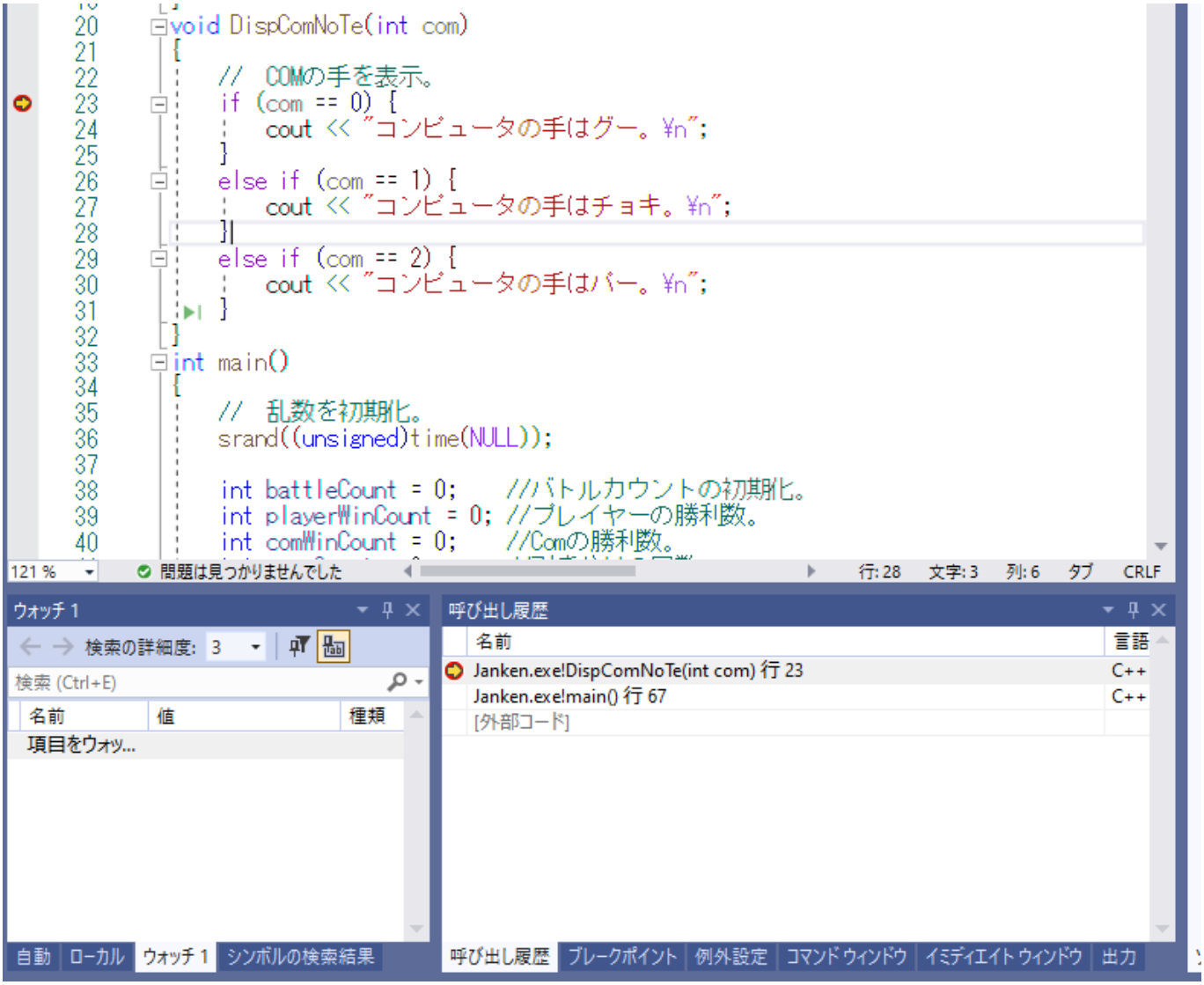
呼び出し履歴を開くと画面の下の方に、呼び出し履歴のウィンドウが追加されます(図1.14)。

図1.13



呼び出し履歴はブレイクポイントでプログラムを停止しているときに利用することができます。呼び出し履歴は、関数がどこから呼ばれているのかをコールされてきたのかを確認することができます。図1.15はDispComNoTe()のブレイクポイントでプログラムが停止しています。

図1.15



このとき、呼び出し履歴を確認すると、DispComNoTe()はmain()から呼ばれていることが分かります。呼び出し元のプログラムをは、呼び出し元の関数名をダブルクリックすることで確認することができます。
(注意：呼び出し元に戻ったときに、１行ずれているので注意が必要です。)

1.2.2 【ハンズオン】呼び出し履歴を使ってみる。

Sample_01を利用して呼び出し履歴を使ってみましょう。

step-1 DispPlayerNoTe()にブレイクポイントを設置する。

DispPlayerNoTe()の10行目にブレイクポイントを設置してください。

step-2 ブレイクポイントでプログラムを停止させる。

では、じゃんけんゲームをデバッガありで実行して、プレイして先ほどのブレイクポイントでプログラムを停止させてください。

step-3 呼び出し履歴を確認する

プログラムを停止させることができたら、呼び出し履歴をつかって、DispPlayerNoTe()が何行目から呼ばれているか確認してください。

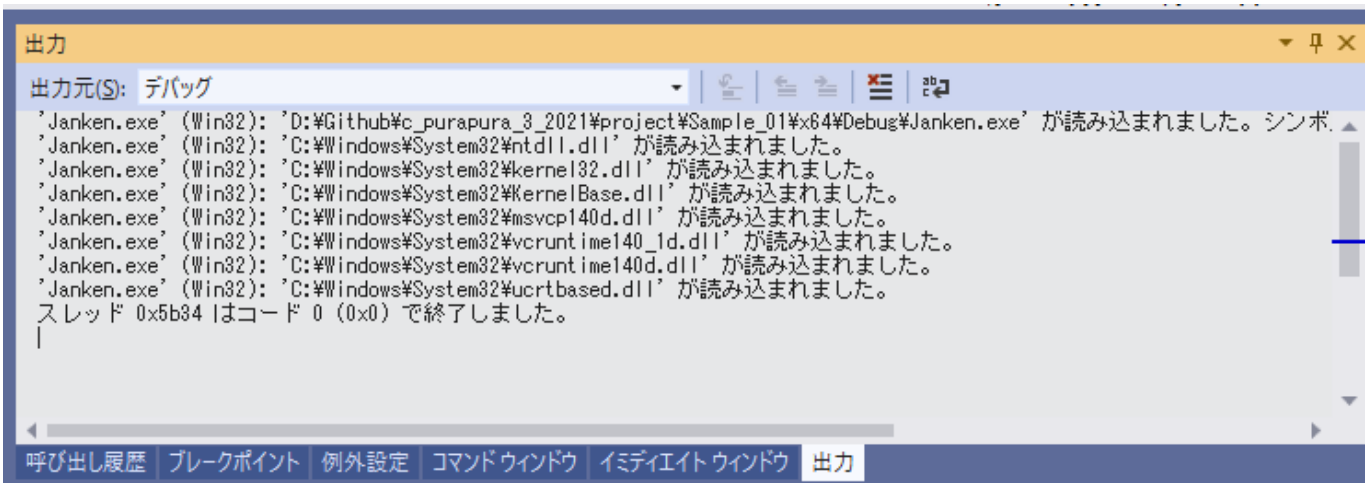
1.2.3 出力

プログラムからのコンソール出力を確認することができます。
出力は、メニューの「デバッグ/ウィンドウ/出力」を選択することで開くことができます(図1.16)。

図1.16

開くと画面の下の方に、出力ウィンドウが追加されます(図1.17)。

図1.17



出力ウィンドウには、プログラムから重要なランタイムのエラーメッセージが出力されている場合があります。
特にDirectXなどでのプログラミングにおいて、動作がおかしいなどの原因不明のエラーが起きた時は、真っ先に出力ウィンドウを調べましょう。
DirectXからエラーメッセージが出力されている場合があります。
君たちのプログラムで出力ウィンドウに値を出力することもできます。
出力ウィンドウへの出力はOutputDebugStringA()を利用します。

```
OutputDebugStringA("Hello world\n");
```

このような機能は、ゲーム実行中にプレイヤーの体力などを確認したいときに利用します。

1.2.3 【ハンズオン】出力ウィンドウを使ってみる。

Sample__01を利用して、出力ウィンドウに値を出力するコードを追加しましょう。
main.cppの87行目にリスト1.1のプログラムを入力してください。

```
//【ハンズオン】出力ウィンドウを使ってみる。
char text[256];
sprintf(text, "player = %d, com = %d\n", player, com);
OutputDebugStringA(text);
```

入力出来たら実行してください。
出力ウィンドウに変数playerとcomの数値が出力されるようになります。

1.2.3 データブレイクポイント

最後に超強力なデバッガの機能、「データブレイクポイント」を紹介します。この機能を使うと、変数の値が変更されたときにプログラムを停止できます。大規模な開発になってくると、様々な要因が重なって変数の値が不正な値に書き換えらえて意図しない不具合が起きることがあります。このような不具合の原因を調べることは、大規模なプロジェクトになってくると、非常に難しいことです。データブレイクポイントは、このような不正な値の書き換えのデバッグ作業を強力にサポートしてくれます。データブレイクポイントを設置するためには、監視したいデータのアドレスが必要になってきます。そのため、ウォッチ機能を使って、対象の変数のアドレスを調べる必要があります。この設置の仕方を文章で伝えるのは、少々手間ですので、下記の動画を参照してみてください。

データブレイクポイントの使い方

1.2.3 【ハンズオン】データブレイクポイントを使ってみる。

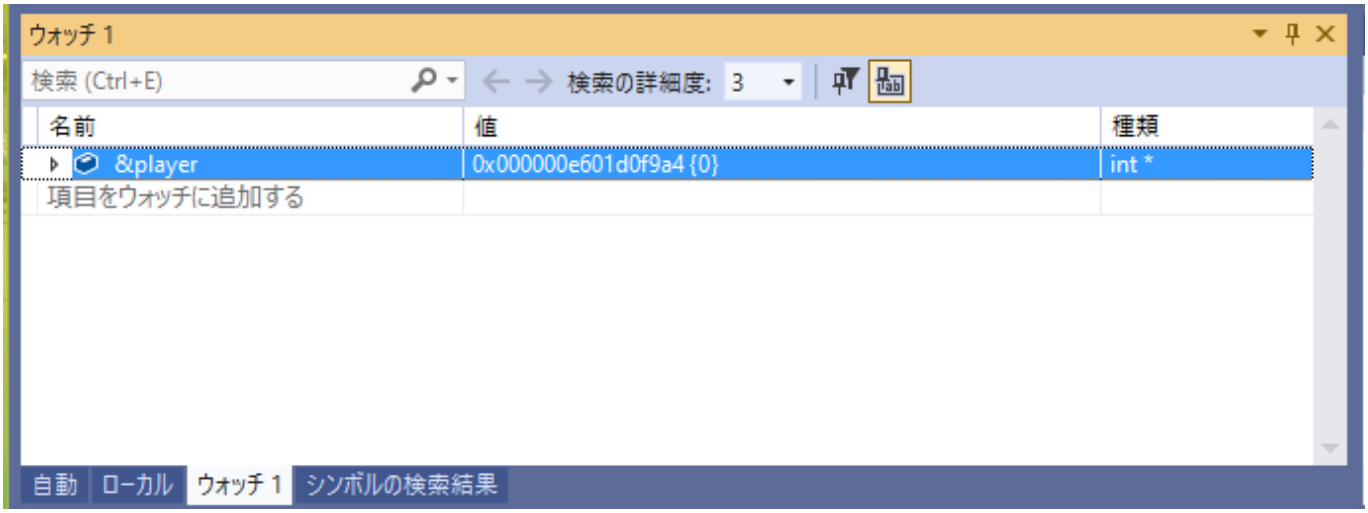
step-1 main.cppの58行目にブレイクポイントを設置。

step-2 デバッガありで実行して、プログラムを58行目で止める。

step-3 変数playerのアドレスをコピーする

ウォッチウィンドウに「&player」と入力して、アドレスをコピーしてください(図1.18)。

図1.18



step-4 データブレイクポイントを設置する。

メニューの「デバッグ/ブレイクポイントの作成/データブレイクポイント」を選択して、データブレイクポイントの作成画面を開く。開けたら、図1.19のように変数playerのアドレスをコピーして、データブレイクポイントを作成する。

step-5 プログラムを再開する。

プログラムを再開して、変数playerの値が変わったタイミングでプログラムが停止することを確認してください。

1.2.4 評価テスト-2

次の評価テストを行いなさい。
評価テストへジャンプ

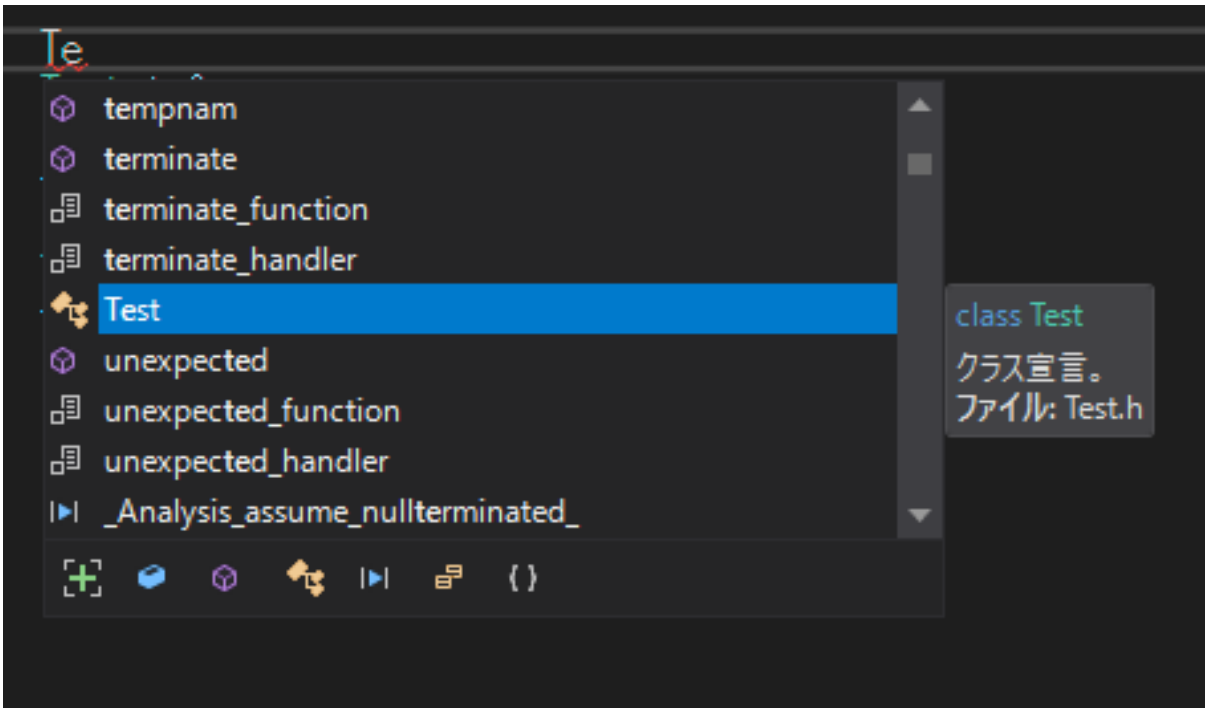
1.3 エディタの利用(基本)

大規模なC++の開発において、高性能なエディタの利用は欠かせません。このチャプターではMicrosoft社が提供している統合環境、VisualStudioのエディタの機能について学んでいきます。

1.3.1 インテリセンス(Ctrl + スペース)

VisualStudioには、強力にコードの入力をサポートしてくれるインテリセンスという機能があります。すでに皆さん使っているのではないかと思いますが、コードを入力すると、残りのコードの候補が出てきます。

あれがインテリセンスの機能です(図1.20)。 **図1.20**

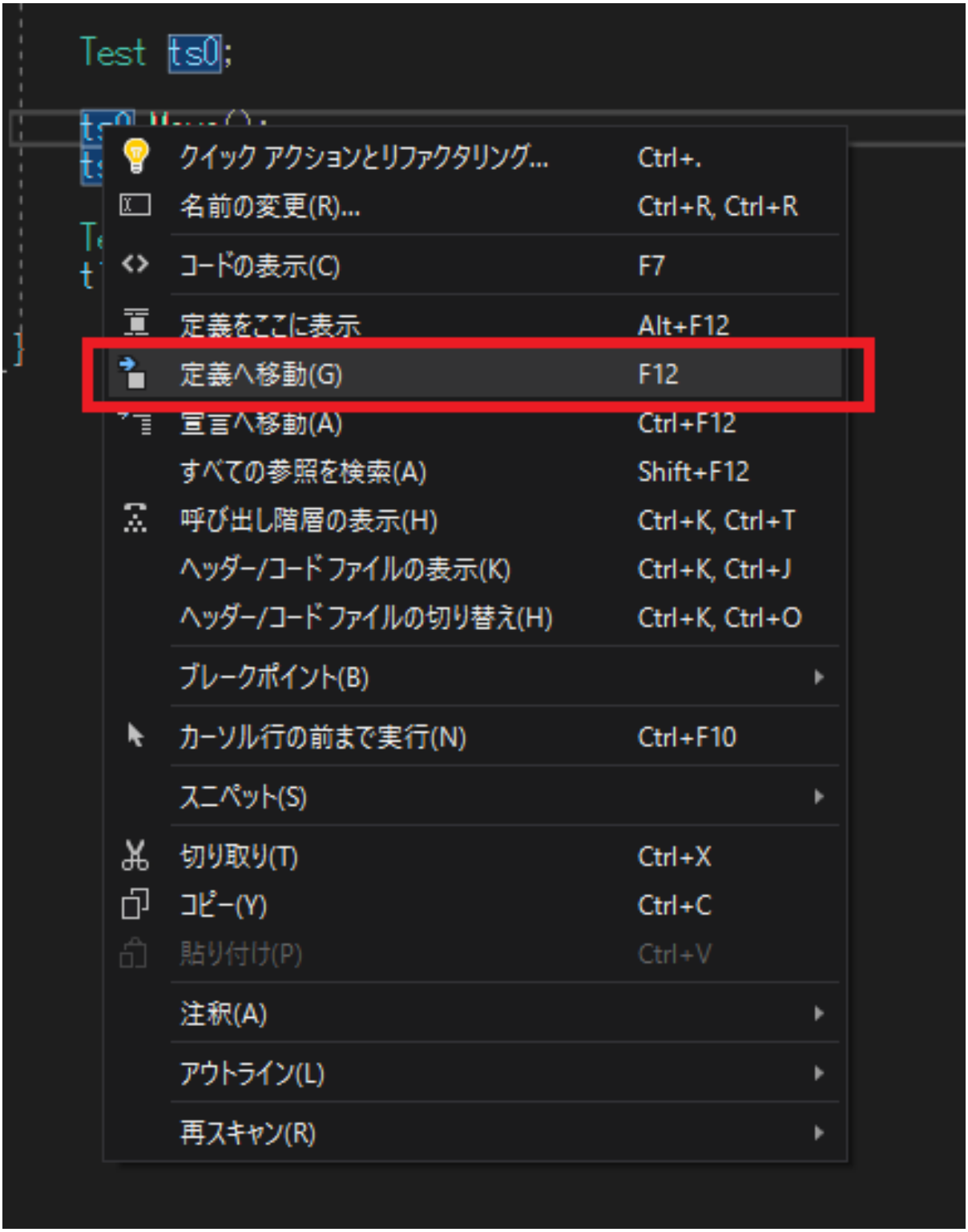


インテリセンスは消えてしまってもショートカットキーのctrl + スペースを入力すると再度出すことができます。インテリセンスは非常に強力な機能なので、ドンドン活用しましょう。

1.3.2 定義へ移動(F12)

定義へ移動を使うと、クラス定義、変数定義、関数定義、enum定義、構造体定義など、様々な定義にジャンプすることができます。これも非常に便利なので、ドンドン活用しましょう。この機能は、定義へ移動したいシンボルにカーソルを合わせて、右クリックで出てくるポップアップメニューから選べば使えます(図1.21)。

図1.21

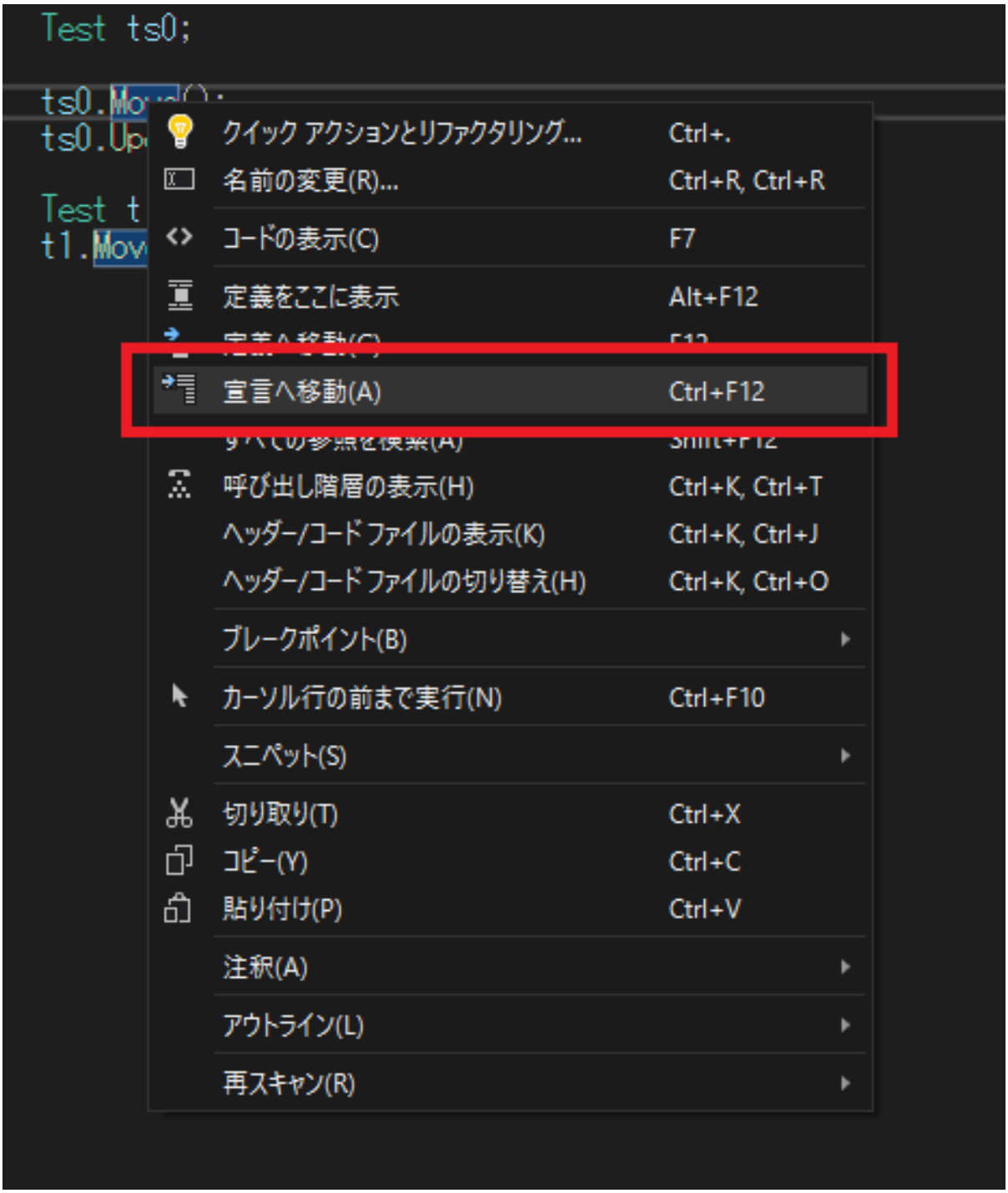


また、ショートカットキーのF12でも定義へ移動することができます。

1.3.3 宣言へ移動(Ctrl + F12)

関数宣言、クラス宣言にジャンプすることができます。この機能は、宣言に移動したいシンボルにカーソルを合わせて、右クリックで出てくるポップアップメニューから選べば使えます(図1.22)。

図1.22



また、ショートカットキーのCtrl + F12でも宣言に移動することができます。

1.3.4 名前の変更

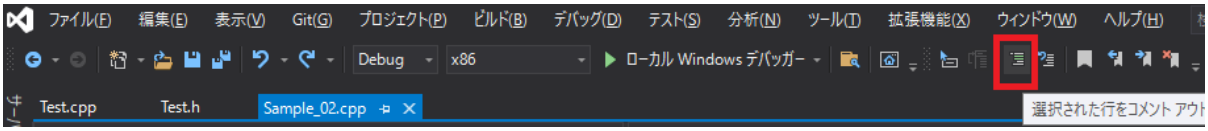
変数、関数、クラスなどの名前の変更を一括でやってくれます。この機能は、変更したいシンボルにカーソルを合わせて、右クリックで出てくるポップアップメニューから選べば使えます(図1.23)。

図1.23

1.3.5 一括コメントアウト

選択した行をコメントアウトすることができます。この機能はメニュー上部の選択された行をコメントアウトを選択すると実行することができます(図1.24)。

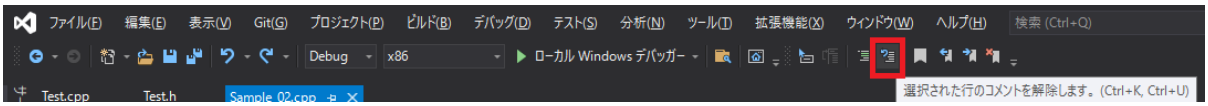
図1.24



1.3.6 一括コメント削除

選択した行のコメントを削除することができます。この機能はメニュー上部の選択された行をコメントアウトを選択すると実行することができます(図1.25)。

図1.25



1.3.7【ハンズオン】ここまでの機能を使ってみる

では、Sample_02/Sample_02.slnを使ってここまでの機能を使ってみましょう。

step-1 インテリセンスを使ってプログラムを入力する。

main.cppに次のプログラムを入力してください。(インテリセンスを積極的に使いましょう)

```
// step-1 インテリセンスを使ってプログラムを入力する。
Test ts0;
ts0.Mobe();
ts0.Update(0);
```

step-2 Ctrl + スペースでインテリセンスを再開してみる。

では、続いて一旦消えてしまった。インテリセンスを再度表示してみましょう。main.cppに次のプログラムを入力してください。

```
// step-1 インテリセンスを使ってプログラムを入力する。
Test ts0;
ts0.M
```

step-3 定義へ移動

Testクラス定義とTest::Update()関数の定義にジャンプしなさい。

step-4 宣言へ移動

Test::Update()関数の宣言にジャンプしなさい。

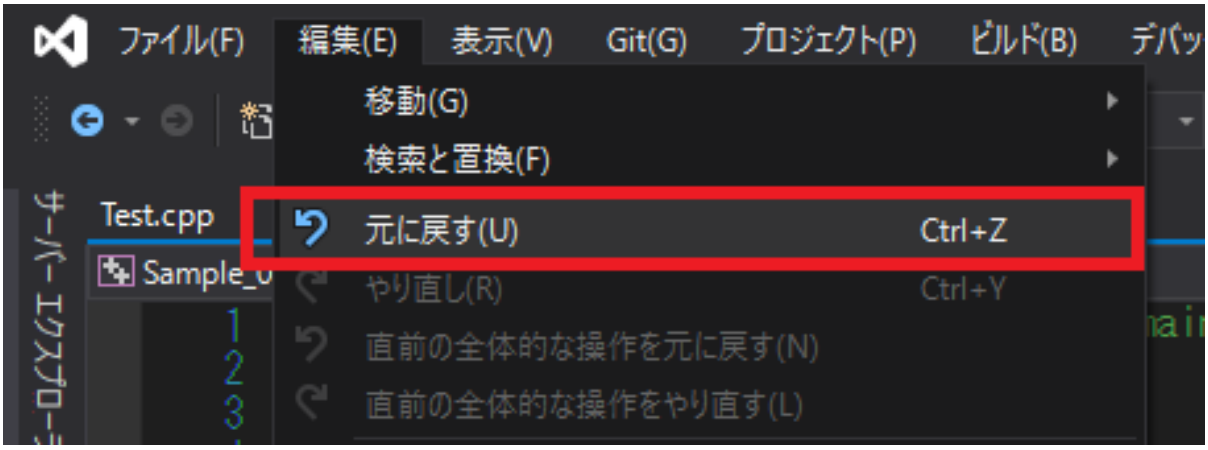
step-5 関数名の変更

Test::Mobe()関数の名前をMoveに変更しなさい。

1.3.8 アンドゥ、元に戻す(ctrl + z)

変更した内容を元に戻すことができます。この操作は非常に頻繁に行う操作です。必ず覚えましょう。アンドゥはメニューの「編集/元に戻す」から行うことができます(図1.26)。

図1.26

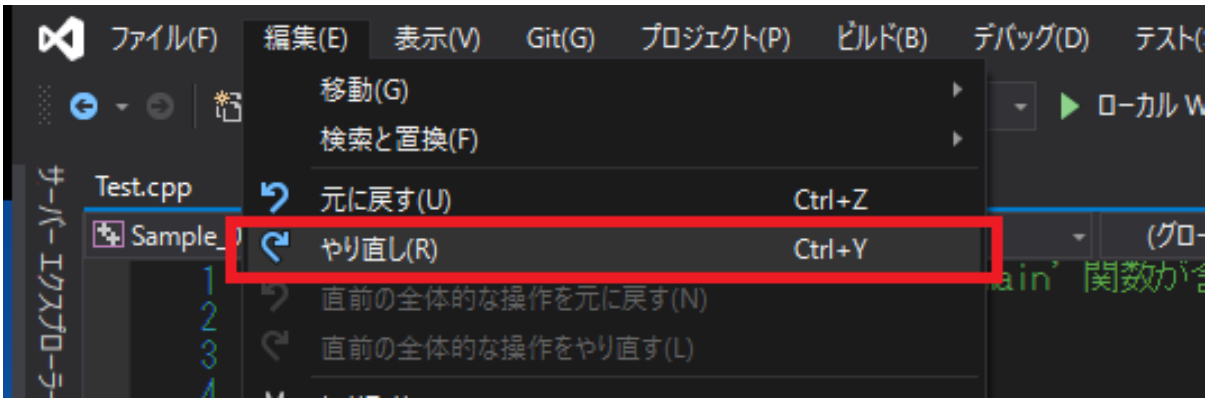


アンドゥはショートカットキーのctrl + Zでも行うことができます。

1.3.9 リドゥ、アンドゥを取り消す(ctrl + y)

アンドゥを取り消すことができます。この操作は非常に頻繁に行う操作です。必ず覚えましょう。リドゥはメニューの「編集/やり直し」から行うことができます(図1.27)。

図1.27



リドゥはショートカットキーのctrl + Yでも行うことができます。

1.3.10 検索(ctrl + F)

テキストを検索することができます。この操作はメニューの「編集/検索と置き換え/クイック検索」から行うことができます。この操作は頻繁に行う操作です。必ず覚えましょう。

図1.27

検索はショートカットキーのctrl + Fでも行うことができます。

1.3.11【ハンズオン】ここまでの機能を使ってみる

では、Sample_02/Sample_02.slnを使ってここまでの機能を使ってみましょう。

step-1 アンドゥを使ってみる。

step-2 リドゥを使ってみる。

step-3 検索を使ってみる。

1.3.12 評価テスト-3

次の評価テストを行いなさい。
評価テストへジャンプ

Chapter 2 C++による大規模開発 ～C++標準テンプレートライブラリの利用(基本)～

2.1 C++標準テンプレートライブラリとは

C++標準テンプレートライブラリとは、C++のプログラミングを楽にするための強力なライブラリです。その中でもこのチャプターでは以下の3つについて取り上げます。

1. vector(可変長配列)
2. list(双方向リスト)
3. map(連想配列)

これらのクラスは、STLと呼ばれる標準テンプレートライブラリの一部であり、ゲーム開発などで非常に有用です。このクラスは基本的に同様の関数を備えているため、1つのクラスを理解すれば、他のクラスの使用方法も簡単に習得できます。

それでは、これらのクラスについて学んでいきましょう。

Chapter 2.2 可変長配列「vector」

2.2 可変長配列「vector」

vectorは要素数を動的に変化させることができる配列です。「動的」というのはプログラム実行中ということです。
これまで、皆さんが勉強した配列は、固定長配列です。固定長配列はプログラムの実行中にサイズを変えることはできません。
次のプログラムを読んでみてください。

```
int hoge[10];
 ・
 省略
 ・
for( int i = 0; i < 10; i++){
    std::cout << hoge[i];
}
```

このプログラムには配列hogeが登場していますが、この配列のサイズは10で固定です。プログラム実行中に、動的にサイズを変えることはできません。
可変長配列はこのサイズをプログラム実行中に変更することができるのです。
要素を追加するにはemplace_back()関数を利用します。

2.2.1 【ハンズオン】vectorを使ってみる。

では、習うより慣れろです。さっそくvectorを使ってみましょう。Sample_02_01を立ち上げてください。 立ち上がったら、main.cppを開いてください。

step-1 vectorをインクルード

vectorを利用するためには、まずインクルードを行う必要があります。main.cppにリスト2.1のプログラムを入力してください。
[リスト2.1]

```
//step-1 vectorをインクルード
#include <vector>
```

step-2 int型の要素を記録できる可変長配列を定義する。

続いて、int型の要素を記録できる可変長配列を定義します。main.cppにリスト2.2のプログラムを入力してください。
[リスト2.2]

```
//step-2 int型の要素を記録できる可変長配列を定義する。
std::vector<int> hoge;
```

step-3 hogeに要素を追加する

続いて、hogeに要素を追加します。要素の追加はemplace_back()関数を利用します。main.cppにリスト2.3のプログラムを入力してください。
[リスト2.3]

```
//step-3 hogeに要素を追加する
hoge.emplace_back(10); //配列の末尾に10を追加する。
hoge.emplace_back(20); //配列の末尾に20を追加する。
hoge.emplace_back(30); //配列の末尾に30を追加する。
```

step-4 hogeのサイズを表示する。

さて、step-3でhogeにint型のデータを3つ追加しました。ですので、可変長配列hogeのサイズは3になっているはずです。可変長配列のサイズはsize()関数で取得することができます。
では、本当にhogeのサイズが3になっているか確認してみましょう。main.cppにリスト2.4のプログラムを入力して下さい。 [リスト2.4]

```
//step-4 hogeのサイズを表示する。
std::cout << "hogeのサイズは" << hoge.size() << "です。\n";
```

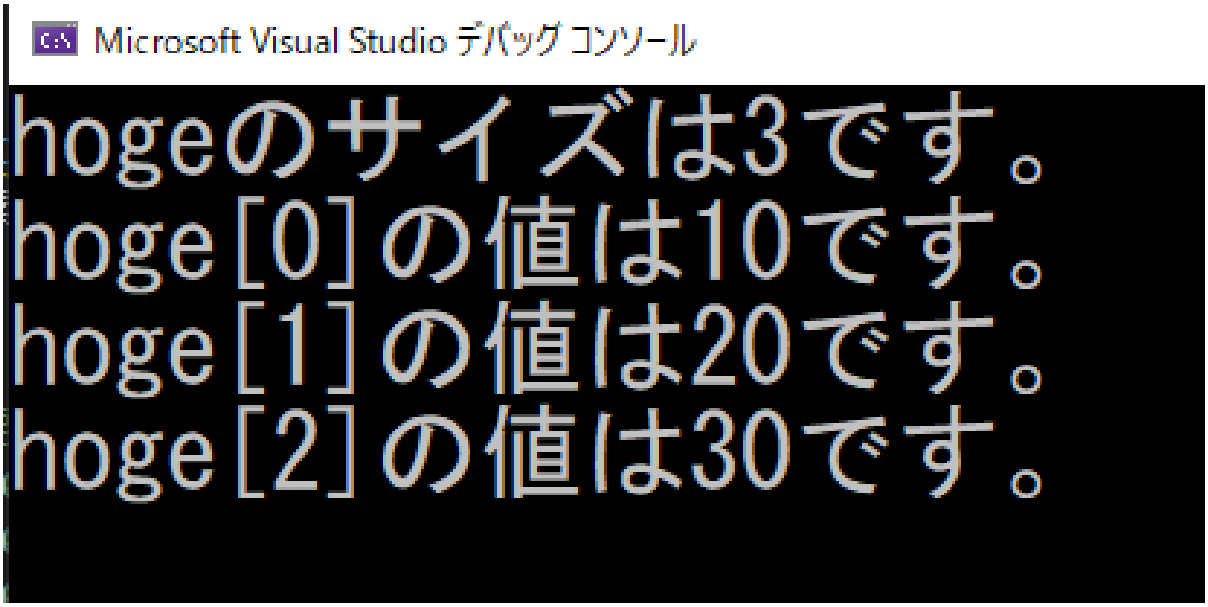
step-5 for文を使って、hogeの要素の値を表示する。

可変長配列は、固定長配列と同じように添え字演算子、[]を利用してアクセスすることができます。では、for文で回して、hogeの各要素を表示してみましょう。

main.cppにリスト2.5のプログラムを入力してください。
[リスト2.5]

```
//step-5 for文を使って、hogeの要素の値を表示する。
for (int i = 0; i < hoge.size(); i++) {
    std::cout
        << "hoge["
        << i
        << "]の値は"
        << hoge[i]
        << "です。\n";
}
```

では、ここまでで一度実行してみて下さい。図2.2のように表示されていたら実装できています。
[図2.2]



step-6 さらにhogeに要素を追加する。

step-6では更にhogeに要素を追加してみましょう。main.cppにリスト2.6のプログラムを入力して下さい。
[リスト2.6]

```
//step-6 さらにhogeに要素を追加する。
hoge.emplace_back(40);
hoge.emplace_back(50);
```

step-7 再度hogeのサイズを表示する。

step-6でhogeに要素を２つ追加したので、hogeの要素数は5になっているはずです。では、再度hogeのサイズを表示するプログラムを追加しましょう。
main.cppにリスト2.7のプログラムを入力してください。
[リスト2.7]

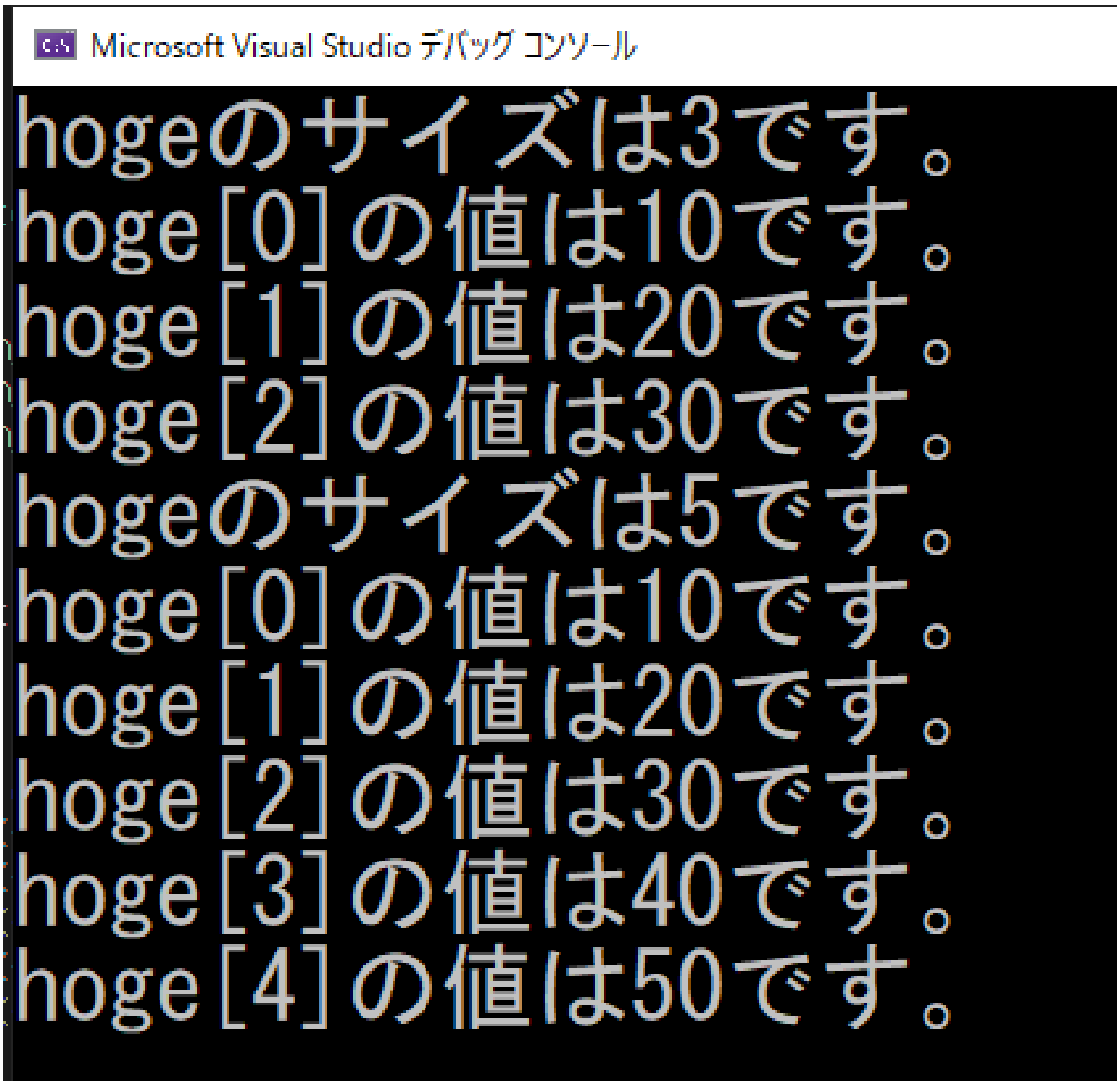
```
//step-7 再度hogeのサイズを表示する。
std::cout << "hogeのサイズは" << hoge.size() << "です。\n";
```

step-8 もう一度for文を使って、hogeの要素の値を表示する。

では、最後にもう一度for文を使ってhogeの要素の値を表示してみましょう。main.cppにリスト2.8のプログラムを入力してください。
[リスト2.8]

```
//step-8 もう一度for文を使って、hogeの要素の値を表示する。
for (int i = 0; i < hoge.size(); i++) {
    std::cout
        << "hoge["
        << i
        << "]の値は"
        << hoge[i]
        << "です。\n";
}
```

入力出来たら実行してみてください。図2.3のように表示されていたら実装できています。
[図2.3]



2.2.1 int型以外も記録できるの？

vectorクラスはテンプレートクラスとなっているため、int型以外の値も記録できます。float型を記録できる可変長配列を定義して、利用するコードを下記に示します。

```
std::vector<float> hoge;
hoge.emplace_back(10.0f);
hoge.emplace_back(20.0f);
hoge.emplace_back(30.0f);
```

vectorクラスはintやfloatのような組み込み型だけではなく、ユーザー定義のクラスでも利用できます。下記のコードはMatrixクラスのオブジェクトの値を記録しているコードです。

```
Matrix m0, m1, m2;
   ・
  省略
   ・
std::vector<Matrix> hoge;
```

```
  hoge.emplace_back( m0 );
  hoge.emplace_back( m1 );
  hoge.emplace_back( m2 );
```

2.2.2 評価テスト-3

次の評価テストを行いなさい。
評価テストへジャンプ

2.3 双方向リスト「list」

listは要素数を動的に変化させることができる双方向リストです。listは可変長配列のvectorと同様に、emplace_back()関数を利用することで、要素数を増やすことができます。次のプログラムを見てみてください。

```
std::list< int > hoge;
hoge.emplace_back( 10 );
hoge.emplace_back( 20 );
hoge.emplace_back( 30 );
```

vectorと同様にsize()関数を利用することで、要素数を取得することができます。

```
std::list< int > hoge;
hoge.emplace_back( 10 );
hoge.emplace_back( 20 );
hoge.emplace_back( 30 );

std::cout << "hogeのサイズは" << hoge.size() << "です。\n";
```

2.3.1 listとvectorの違い

さて、要素数を動的に変化させるという点では、listとvectorは全く同じもののように思えます。この節では、listとvectorの違いについていくつか見ていきましょう。

listは添え字演算子が使えない。

listはvectorと違って添え字演算子が使えません。つまり、次のようなプログラムがかけないということです。

```
std::list< int > hoge;
hoge.emplace_back( 10 );
hoge.emplace_back( 20 );
hoge.emplace_back( 30 );

// できないのでコンパイルエラーが起きる。
std::cout << hoge[2];
```

では、listを使っているときに要素のアクセスをどうするのか？というとイテレーター(反復子)を利用して、要素の先頭から順繰りアクセスしていきます。

```
std::list< int > hoge;
hoge.emplace_back( 10 );
hoge.emplace_back( 20 );
hoge.emplace_back( 30 );
```

```
//begin()関数を使って、先頭イテレータを取得する。
std::list< int >::iterator it = hoge.begin();
// 10と表示される。
std::cout << *it << "\n"
// 次の要素に進む。
it++;
// 20と表示される。
std::cout << *it << "\n"
```

イテレーターはlistが格納している要素を指し示す、ポインタのようなものです。ポインタと同じように*演算子を利用して、イテレーターが指し示している要素にアクセスすることができます。
また、イテレーターはlistだけではなくvectorや、この後勉強するmapなど、C++の標準ライブラリのコンテナクラスの全て利用できます。

> コンテナクラスとは、listやvectorのように、要素を記憶することができるC++標準ライブラリのクラスです。
> listやvector以外にも、array、deque、fowrard_list、map、multimap、unordered_map、set、multisetなどいくつか存在しています。
> どのコンテナクラスにもメリット、デメリットが存在しており、必要に応じて適切なコンテナクラスを選択することが重要になってきます。

イテレーターの典型的な利用方法として、for文での利用があげられます。

```
std::list< int > hoge;
hoge.emplace_back( 10 );
hoge.emplace_back( 20 );
hoge.emplace_back( 30 );

// for文を使って、リストの全要素にアクセスする。
for(
    std::list< int >::iterator it = hoge.begin();
    it != hoge.end(); // イテレーターが終端に到達するまで
    it++              // 次のイテレーターへ
){
    std::cout << *it << "\n";
}
```

listを使うべき場面は？

基本的にゲームにおいては、可変長配列vectorの方が多くの場面でlistより向いています。
しかし、頻繁に要素の追加と削除が発生する場合に、vectorクラスの代わりに利用することを検討すべきです。
この理由については、C++による大規模開発 ～C++標準テンプレートライブラリの利用(応用)～で解説します。

2.3.1 【ハンズオン】listを使ってみる。

では、習うより慣れろです。さっそくvectorを使ってみましょう。Sample_02_02を立ち上げてください。 立ち上がったら、main.cppを開いてください。

step-1 listをインクルード

listを利用するためには、まずインクルードを行う必要があります。main.cppにリスト2.1のプログラムを入力してください。
[リスト1]

```
//step-1 listをインクルード
#include <list>
```

step-2 int型の要素を記録できる双方向リストを定義する。

続いて、int型の要素を記録できる可変長配列を定義します。main.cppにリスト2のプログラムを入力してください。
[リスト2]

```
//step-2 int型の要素を記録できる双方向リストを定義する。
std::list<int> hoge;
```

step-3 hogeに要素を追加する

続いて、hogeに要素を追加します。要素の追加はemplace_back()関数を利用します。main.cppにリスト3のプログラムを入力してください。
[リスト3]

```
//step-3 hogeに要素を追加する
hoge.emplace_back(10); //配列の末尾に10を追加する。
hoge.emplace_back(20); //配列の末尾に20を追加する。
hoge.emplace_back(30); //配列の末尾に30を追加する。
```

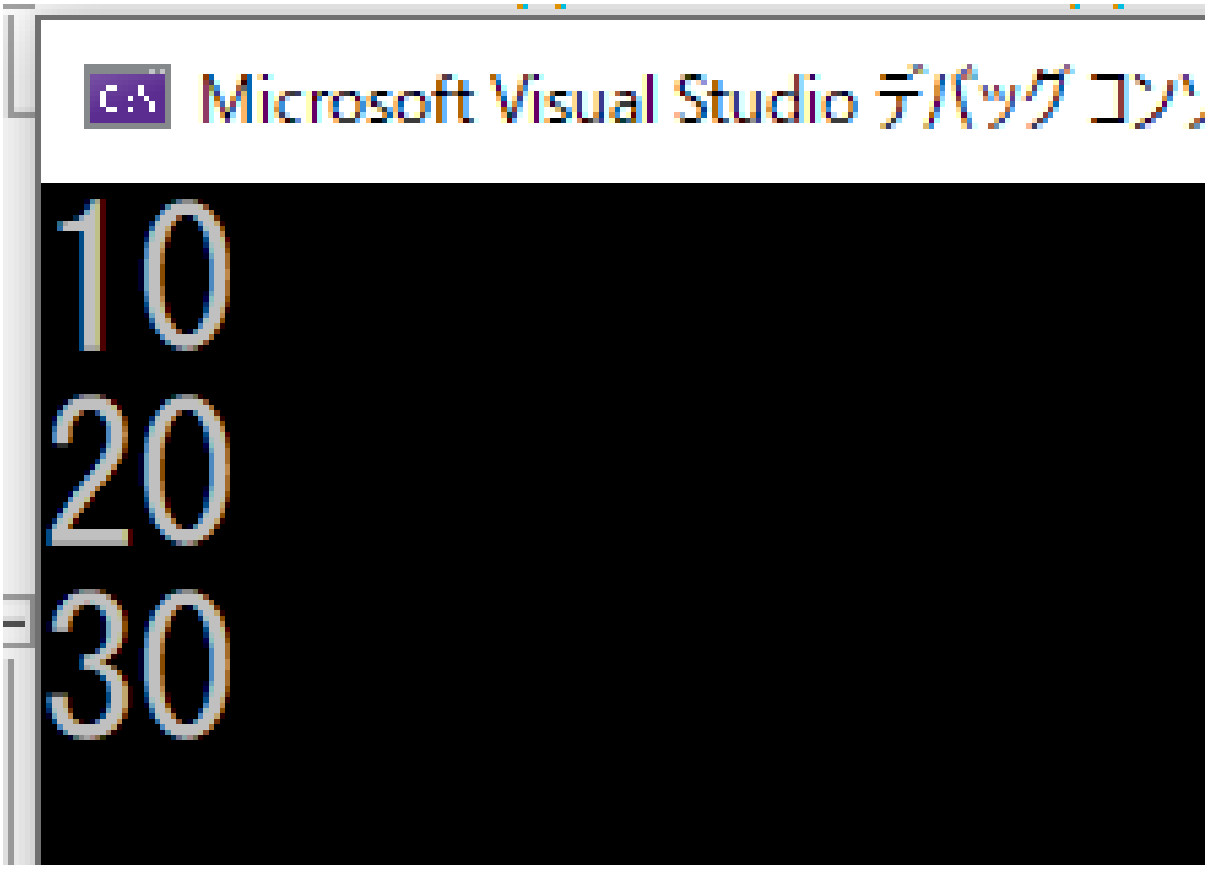
step-5 for文を使って、hogeの要素の値を表示する。

続いて、for文で回して、hogeの各要素を表示してみましょう。可変長配列と違い、添え字演算子が使えないため、イテレータを利用します。
main.cppにリスト4のプログラムを入力してください。
[リスト4]

```
// step-4 for文を使って、hogeの要素の値を表示する。
for (
    std::list<int>::iterator it = hoge.begin();
    it != hoge.end();
    it++
```

```
) {
    std::cout << *it << "\n";
}
```

では、ここまで実行してみて下さい。図2.4のように表示されていたら実装できています。
[図2.4]



2.3.2 評価テスト-4

次の評価テストを行いなさい。
評価テストへジャンプ

2.4 連想配列「map」

この節では次のプログラムを使用します。次のURLからプログラムをダウンロードしておいてください。
Sample_02_03.zip

mapはキーと値付きで要素を記憶することができるコンテナクラスです。mapに格納されている要素はstd::pair型のオブジェクトです。
mapにはstd::pair型のオブジェクトをinsert()関数を利用することで追加することができます。次のコードを見てください。

```
// 名前と年齢を記憶できる連想配列を定義する。
std::map< std::string, int > map;

// 豊臣秀吉をキー、39歳を値に持つデータを作成する。
std::pair< std::string, int > data;
data.first = "豊臣秀吉"; // firstがキー。
data.second = 39;       // secondが値。

// 作成したデータを連想配列に追加する。
map.insert( data );

// 続いて、織田信長、50歳を値に持つデータを作成する。
data.first = "織田信長";  // firstがキー。
data.second = 50;        // secondが値。

// 作成したデータを連想配列に追加する。
map.insert(data);
```

mapへの要素の追加は次のように記述することもできます。

```
// 名前と年齢を記憶できる連想配列を定義する。
std::map< std::string, int > map;
// 豊臣秀吉をキー、39を値にもつデータを挿入。
map.insert( {"豊臣秀吉", 39} );
// 織田信長、50歳を値に持つデータを挿入。
map.insert( {"織田信長", 50} );
```

連想配列に記憶されている要素は、キーを使うことでアクセスすることができます。

```
std::cout << map["豊臣秀吉"] << "\n"; // 39と表示される。
std::cout << map["織田信長"] << "\n"; // 50と表示される。
```

mapもイテレータを利用して、for文を回すことができます。

```
for(
    std::map< std::string, int >::iterator it = map.begin();
    it != map.end();
    it++
){
    // イテレーターはmapに格納されている要素を指しているの、std::pair型のオブジェクトを指しています。
    // なので、各要素のキーと値にアクセスすることができます。
    std::cout << "キーは" << it->first << "\n";
    std::cout << "値は" << it->second << "\n";
}
```

mapを使うべき場面は？

データとデータを関連付けて記憶しておきたい場合。ゲームであれば、例えば読み込み済みのリソースの記録などで利用されます。
値にリソースのファイルパス、値に読み込み済みのリソースなどを記録しておき、使いまわす等の実装が行われることがあります。

mapへのデータの挿入のされ方は？

mapへのデータの挿入の関数がemplace_back()関数ではないことに注意してください。mapにはemplace_back()関数はありません。mapはinsert()関数が用意されており、insert()関数が要素を挿入する位置はC++の使用では定義されていません。
つまりどこに挿入されるか不明になっています。
例えば次のコードの場合、織田信長と豊臣秀吉のデータのどちらが先に表示されるか分からないということです。

```
std::map< std::string, int > map;
// 豊臣秀吉をキー、39を値にもつデータを挿入。
map.insert( {"豊臣秀吉", 39} );
// 織田信長、50歳を値に持つデータを挿入。
map.insert( {"織田信長", 50} );

//  豊臣秀吉が先に表示されるわけではない！！！
for(
    std::map< std::string, int >::iterator it = map.begin();
    it != map.end();
    it++
){
    // イテレーターはmapに格納されている要素を指しているの、std::pair型のオブジェクトを指しています。
    // なので、各要素のキーと値にアクセスすることができます。
    std::cout << "キーは" << it->first << "\n";
    std::cout << "値は" << it->second << "\n";
}
```

mapの実装のされ方は、コンパイラベンダーによって異なりますが、多くのベンダーではキーを参照しての2分木データ構造になっています。

2分木構造になっている理由は、キーを使った要素へのアクセス時間を対数時間にすることができ、高速にすることができるためです。
この詳細については、C++による大規模開発 ～C++標準テンプレートライブラリの利用(応用)～で解説します。

2.4.1 【ハンズオン】mapを使ってみる。

では、さっそくmapを使ってみましょう。Sample_02_03を立ち上げてください。 立ち上がったら、main.cppを開いてください。

step-1 mapをインクルード

mapを利用するためには、まずインクルードを行う必要があります。main.cppにリスト1のプログラムを入力してください。
[リスト1]

```
//step-1 mapをインクルード
#include <map>
```

step-2 名前と年齢を記憶できる連想配列を定義する。

続いて、int型の要素を記録できる可変長配列を定義します。main.cppにリスト2のプログラムを入力してください。
[リスト2]

```
// 名前と年齢を記憶できる連想配列を定義する。
std::map< std::string, int > map;
// 豊臣秀吉をキー、39を値にもつデータを挿入。
map.insert( {"豊臣秀吉", 39} );
// 織田信長、50歳を値に持つデータを挿入。
map.insert( {"織田信長", 50} );
```

step-3 キーを使って要素にアクセスする。

続いて、キーを使って要素にアクセスしてみましょう。リスト3のプログラムを入力してください。
[リスト3]

```
// step-3 キーを使って要素にアクセスする。
std::cout << map["豊臣秀吉"] << "\n"; // 39と表示される。
std::cout << map["織田信長"] << "\n"; // 50と表示される。
```

step-4 for文を使って、mapに格納されているキーと値を表示する。

続いて、for文で回して、mapの各要素を表示してみましょう。main.cppにリスト4のプログラムを入力してください。

[リスト4]

```
//step-4 for文を使って、mapに格納されているキーと値を表示する。
for (
    std::map< std::string, int >::iterator it = map.begin();
    it != map.end();
    it++
    ) {
    // イテレーターはmapに格納されている要素を指しているの、std::pair型のオブジェクトを指しています。
    // なので、各要素のキーと値にアクセスすることができます。
    std::cout << "キーは" << it->first << "\n";
    std::cout << "値は" << it->second << "\n";
}
```

では、ここまで実行してみて下さい。図2.4のように表示されていたら実装できています。
[図2.5]

2.4.2 評価テスト-5

次の評価テストを行いなさい。
評価テストへジャンプ

2.4.3 実習課題

次のプログラムをダウンロードして、プログラム中のコメントと完成動画を参考にして実習を行いなさい。
Question_02_01.zip

Chapter 3 C++による大規模開発 ～C++標準テンプレートライブラリの利用(中級)～

この節では次のプログラムを使用します。次のURLからプログラムをダウンロードしておいてください。
Question_03_01.zip
Question_03_02.zip

3.1 検索

std::find()関数を利用することで、可変長配列に記録されている要素を検索することができます。次のコードはfind()関数の利用例です。

```
std::vector< int > hoge;
hoge.emplace_back(30);
hoge.emplace_back(40);
hoge.emplace_back(50);

std::vector<int>::iterator it;
// find関数は第一引数と第二引数にイテレーターを指定して、
// 検索範囲を指定する。第三引数は検索する値。
// 今回だと、hogeの先頭から終端までの範囲で、40という値を検索している。
it = std::find(
    hoge.begin(),
    hoge.end(),
    40
);

if ( it == hoge.end() ) {
    // 検索対象の値が見つからなかった場合は終端のイテレーターを返してくる。
    std::cout << "見つからなかった" << "\n";
}else{
    std::cout << "見つかった" << *it << "\n";
}
```

3.2 要素の削除

vectorはerase()関数を利用することで、要素を削除することができます。erase()関数に削除したいイテレーターを渡すことで削除することができます。次のコードはerase()関数の利用例です。

```
std::vector< int > hoge;
hoge.emplace_back(30);
hoge.emplace_back(40);
hoge.emplace_back(50);

std::vector<int>::iterator it;
// hogeから40が記録されているイテレータを検索する。
it = std::find(
    hoge.begin(),
    hoge.end(),
    40
);
```

```
if ( it != hoge.end() ) {
    // 見つかったので削除
    hoge.erase( it );
}
```

3.2 実習課題

Question_03_01のコメントを読んで課題に取り組みなさい。

3.3 実習課題(上級)

Question_03_02のコメントを読んで課題に取り組みなさい。

3.1 検索

双方向リストは、可変長配列と同様に、std::find()関数を利用することで、リストに記録されている要素を検索することができます。次のコードはfind()関数の利用例です。

```
std::list< int > hoge;
hoge.emplace_back(30);
hoge.emplace_back(40);
hoge.emplace_back(50);

std::list<int>::iterator it;
// find関数は第一引数と第二引数にイテレーターを指定して、
// 検索範囲を指定する。第三引数は検索する値。
// 今回だと、hogeの先頭から終端までの範囲で、40という値を検索している。
it = std::find(
    hoge.begin(),
    hoge.end(),
    40
);

if ( it == hoge.end() ) {
    // 検索対象の値が見つからなかった場合は終端のイテレーターを返してくる。
    std::cout << "見つからなかった" << "\n";
}else{
    std::cout << "見つかった" << *it << "\n";
}
```

3.2 要素の削除

listは可変長配列と同様にerase()関数を利用することで、要素を削除することができます。erase()関数に削除したいイテレーターを渡すことで削除することができます。次のコードはerase()関数の利用例です。

```
std::list< int > hoge;
hoge.emplace_back(30);
```

```
hoge.emplace_back(40);
hoge.emplace_back(50);

std::list<int>::iterator it;
// hogeから40が記録されているイテレータを検索する。
it = std::find(
    hoge.begin(),
    hoge.end(),
    40
);

if ( it != hoge.end() ) {
    // 見つかったので削除
    hoge.erase( it );
}
```