

# App 1 : オンラインコンパイラー

**オンラインコンパイラー** (online compiler) とは、Web ブラウザー上でソースコードの編集、コンパイル、そして実行までが可能な Web サービスです。オンラインコンパイラーのほとんどは無償で利用でき、さらに C++ だけでなくさまざまな言語を実行できます。今後 C++ 以外のプログラミング言語を勉強するときにも利用できるので、どれか一つでも使い方を覚えておくとよいでしょう。

あまりプログラムの開発に慣れていない初学者の方は、プログラミングの勉強以前に開発環境をセットアップするところで挫折してしまうことが多くありますが、オンラインコンパイラーを利用することでそういった難しい部分をスキップしてプログラミングの勉強に専念することができます。オンラインコンパイラーは便利な反面、あまり巨大であったり複雑だったりするプログラムを実行することはできません。本書で紹介しているプログラム例ぐらいであれば問題なく実行できますが、プログラミングを本格的に行っていくには、App 2 で解説するような開発環境のセットアップが必要となります。

## App 1.1 Wandbox

**Wandbox** (ワンドボックス) は、@melpon 氏と@kikairoya 氏により開発、運用されているオンラインコンパイラーです。Wandbox の大きな特徴は、対応しているコンパイラーのバージョンや種類が豊富であることと、複数のファイルを扱うことができることです。

- Wandbox :

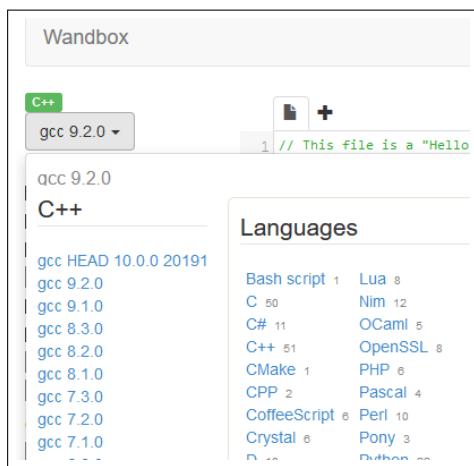
URL <https://wandbox.org/>

他のサービスのほとんどは、1 つの言語につきコンパイラーが 1 種類、多くても数種類しかない場合がほとんどですが、Wandbox では 1~20 種類程度、特に C++ や C 言語にいたっては 50 種類近くも利用できます。さらにコンパイル時のオプションも設定できるので 1 つのソースコードをさまざまな設定で試すことができます。

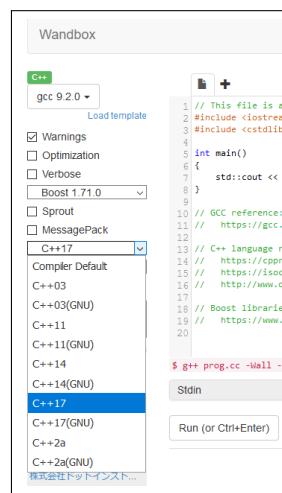
C++ のライブラリの開発では、幅広いバージョンをサポートしなければならないことが多く、Wandbox を利用することでたくさんのコンパイラをセットアップする手間が大幅に省けるため大変重宝します。

### App 1.1.1 単一ファイルのコンパイル

言語選択のコンボボックスで [C++] を選択し、利用したいコンパイラを選択します（図 App1.1）。このとき [Warning] のチェックボックスにチェックが付いていること、言語仕様の選択が [C++17] になっていることを確認してください（図 App1.2）。



❖ 図 App1.1 コンパイラの選択



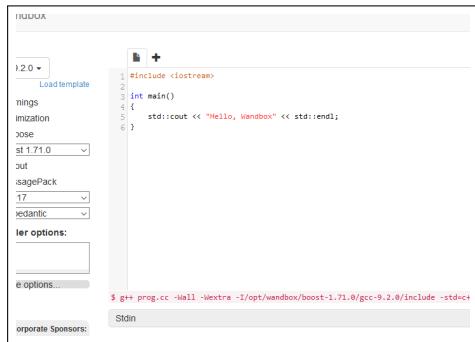
❖ 図 App1.2 コンパイラーオプションの選択

ちなみに「C++2a」というのは次期標準仕様に向けた実装途中のバージョンを指します。もし最新仕様のテストをしたい場合には、ここで [C++2a] を選択すると新しい機能を試すことができます。

利用可能なコンパイラの種類がとても多いので最初のうちは迷ってしまうかと思いますが、主に次の点に注意して選んでもらえれば問題ありません。

- 利用可能なコンパイラは主に 2 つ、GCC と Clang(クラン)ですが、どちらも「HEAD」と書かれたものの 1 つ下のバージョンを使用してください
- ただし GCC では、真ん中のバージョン番号が「x.1.0」となっている場合、さらにもう 1 つ下のバージョンを使うことをおすすめします（5 ページ Column 参照）

中央のテキストエリアにソースコードを記述します（図 App1.3）。



The screenshot shows the Wandbox web interface. On the left, there are various configuration options: 'Compiler' set to 'g++ 9.2.0', 'Platform' set to 'Ubuntu 18.04', 'Linker' set to 'ld.gold', and 'Memory limit' set to '1GB'. Below these are sections for 'I/O options' and 'Environment variables'. The main area is a code editor with the following C++ code:

```
#include <iostream>
int main()
{
    std::cout << "Hello, Wandbox" << std::endl;
}
```

At the bottom of the code editor, there is a command line interface showing the compilation command: '\$ g++ prog.cc -Wall -Wextra -I/opt/wandbox/boost-1.71.0/gcc-9.2.0/include -std=c++17'. Below the command line is a 'Stdin' input field.

❖図 App1.3 ソースコードの記述

[Run] をクリックするか、[Ctrl] + [Enter] キーでを押すことでコンパイル・実行され、コンパイル結果と実行結果が下に表示されます（図 App1.4）。

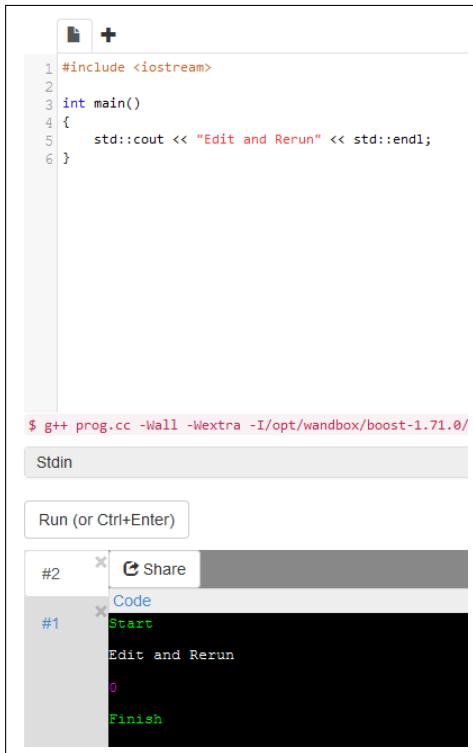


The screenshot shows the Wandbox interface after compilation and execution. The top part of the interface remains the same as in Figure App1.3. Below the code editor, there is a 'Run (or Ctrl+Enter)' button. A terminal window titled '#1' displays the output of the program:

```
$ g++ prog.cc -Wall -Wextra -I/opt/wandbox/boost-1.71.0/gcc-9.2.0/include -std=c++17
Stdin
#1 Share
Code
Start
Hello, Wandbox
0
Finish
```

❖図 App1.4 実行と実行結果

プログラムを修正して再度 [Run] をクリックすると新たな実行結果が表示されますが、過去の実行結果は別のタブで残っているので比較することができます（図 App1.5）。



The screenshot shows a C++ development environment. In the top-left, there's a code editor window with the following code:

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Edit and Rerun" << std::endl;
6 }
```

Below the code editor is a terminal window showing the command:

```
$ g++ prog.cc -Wall -Wextra -I/opt/wandbox/boost-1.71.0/
```

Underneath the terminal is a "Stdin" input field. To the right of these windows is a "Run (or Ctrl+Enter)" button. Below the run button is a tab bar with two tabs labeled "#2" and "#1". The "#2" tab is active and contains the text "Share". The "#1" tab is inactive and contains the text "Code". Under the "Code" tab, there are four entries: "Start", "Edit and Rerun", "0", and "Finish".

❖ 図 App1.5 編集と再実行

#### ◆ Column：コンパイラーのバージョン

GCC のバージョン番号は少々独特になっています。GCC は 5.0.0 以降、「x.1.0」が**×**  
**ジャーリリース**という「大きく機能追加や変更などを行った最初のリリース版」になっ  
ています。

その後バグ修正が行われていくと、「x.2.0」「x.3.0」と真ん中の番号（マイナーバージョ  
ン）が進んでいきます。一応「x.1.0」は安定版としてリリースされるのですが、それ  
でもやはり大きなバグが残っていることがあります。巨大なプロジェクトで使われ始める  
とバグが一気に洗い出されます。

そのため、「x.1.0」はコーナーケースでバグを起こしてしまうかもしれませんので、「x.2.0」  
以降のマイナーリリースを使うとより安心して使うことができます。

ごくまれに「x.y.1」というパッチリリースが出ることがあります。これはセキュリ  
ティー上の問題など、次のマイナーリリースまで待つのが難しい場合にリリースされ  
るバージョンです。

「x.0.0」(HEAD) は「x.1.0」に向けての開発が行われるバージョンで、活発に機能追  
加や変更が行われる反面とても多くのバグがあるバージョンです。

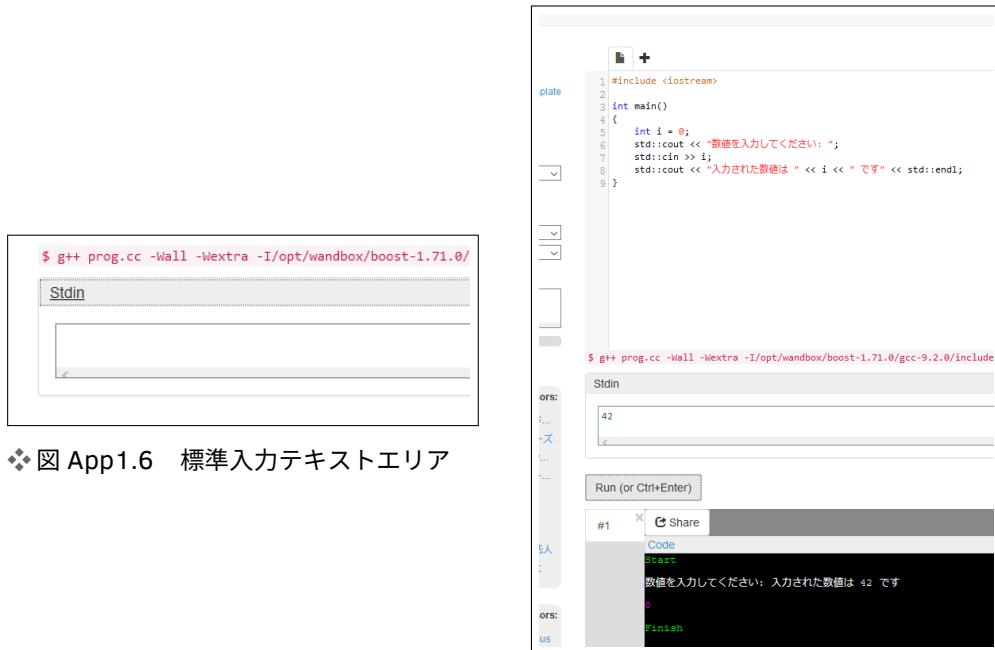
次期標準仕様に向けて機能が追加されていくので、最新の言語仕様を調べたり、できる  
だけ機能が安定した状態でリリースされるように開発に貢献するなどする場合には使  
いますが、みなさんが通常使用するにはまったく向きません。

Clang は GCC と違ってマイナーリリースやパッチリリースをほとんど出さないので、  
最新の開発版 (HEAD) さえ避けねばある程度安定した状態で動作します。

## App 1.1.2 標準入力

Wandbox では実行して結果を表示するだけでなく、入力したい文字列をあらかじめ設定することができます。この文字列は実行前に入力しておく必要があり、プログラムを実行しながら対話的に文字列を与えることはできません。

【Stdin】をクリックすると標準入力に入力する文字列を記述するテキストエリアが出現します（図 App1.6、図 App1.7）。



❖図 App1.6 標準入力テキストエリア

❖図 App1.7 標準入力からの入力

対話的には扱えないので、出力結果のところではプロンプトの表示がおかしくなってしまいます。残念ながらこれをうまく回避することはできないので、Wandbox 上でプログラムを書くときは余計に改行を入れるなど、工夫が必要です。

### App 1.1.3 ファイル I/O

Wandbox はソースファイルそれ自体だけでなく、別のファイルも作ることができます。ファイルの入出力についても試すことができます。

プログラムで一からファイルを作る場合には `std::fstream` や `std::ofstream` を直接使って作成すればよいのですが、あらかじめファイルを用意してそれに対して入出力したい場合には次の手順でファイルを作成できます。

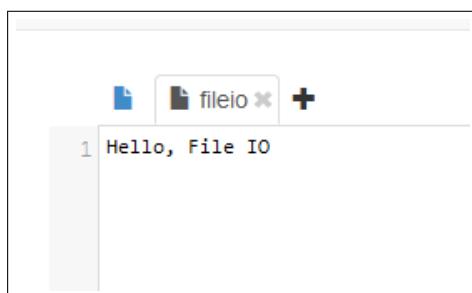
#### 手順

- ① ファイルタブの隣にある [+] をクリックして、新しいファイルを作成します（図 App1.8）。



❖図 App1.8 新しいファイルを作成

- ② 新しくできたファイルタブを選択し、さらにそのタブをもう一度クリックするとファイル名を変更できるので、目的のファイル名に変更してファイルの中身を記述します（図 App1.9）。



❖図 App1.9 ファイルの準備

③作成したファイルはそのファイル名でファイルストリームを作ってプログラムの中から読み書きすることができます（図 App1.10）。

The screenshot shows a code editor window with a tab labeled "fileio". The code in the editor is:

```
1 #include <string>
2 #include <iostream>
3 #include <fstream>
4
5 int main()
6 {
7     std::ifstream ifs{"fileio"};
8
9     std::string line;
10    std::getline(ifs, line);
11
12    std::cout << line << std::endl;
13 }
```

Below the editor, a terminal window shows the command:

```
$ g++ prog.cc -Wall -Wextra -I/opt/wandbox/boost-1.71.0/gcc-9.2.0/include
```

The terminal output is:

```
Stdin
```

Run (or Ctrl+Enter)

```
#1 Share
Code
Start
Hello, File IO
0
Finish
```

❖図 App1.10 ファイルからの入力

## App 1.1.4 分割コンパイル

分割コンパイルする場合もファイル I/O と同じ要領で追加のソースファイルを作成します（図 App1.11、図 App1.12）。このとき、追加したヘッダーファイルをインクルードするには、`#include "filename"` と記述する必要があります。



```
feature.h
1 void show_message();
```

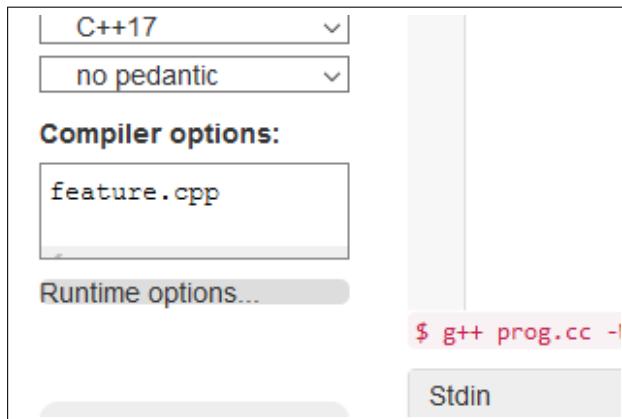
❖ 図 App1.11 feature.h



```
feature.cpp
1 #include "feature.h"
2 #include <iostream>
3
4 void show_message()
5 {
6     std::cout << "Hello, ファイル分割" << std::endl;
7 }
```

❖ 図 App1.12 feature.cpp

ただし、そのままではコンパイルの対象とはならないので、コンパイラーのオプションに作成したソースファイル名を指定します（図 App1.13）。



❖ 図 App1.13 追加したソースファイルの指定

本体のプログラムを記述して実行するとすべてのソースファイルがコンパイル、リンクされ実行結果が表示されます（図 App1.14）。

The screenshot shows the Wandbox C++ compiler interface. On the left, there are compiler settings for 'gcc 9.2.0' (Warnings checked, Optimization and Verbose unchecked, Boost 1.71.0 selected), Compiler options (feature.cpp), and Runtime options. Below these are corporate sponsors. In the center, a code editor window displays the following C++ code:

```
#include "feature.h"
int main()
{
    show_message();
}
```

Below the code editor is a terminal window showing the command used to compile the program:

```
$ g++ prog.cc -Wall -Wextra -I/opt/wandbox/boost-1.71.0/gcc-9.2.0/include -s
```

The terminal output shows the program's execution:

```
Stdin
Run (or Ctrl+Enter)
#1  Share
Code
Start
Hello, ファイル分割
0
Finish
```

❖図 App1.14 分割コンパイルの実行

# App 2：開発環境のセットアップ

オンラインコンパイラは面倒なセットアップが不要ですぐにプログラミングを始められるので大変便利ですが、複雑なプログラムを作るには難があります。また、オープンソースソフトウェアなど、ソースコードが公開されているソフトウェアをビルド（書籍第4章参照）する場合にも、オンラインコンパイラを使うことはできません。

そういう場合にはオンラインコンパイラではなく、それぞれのプラットフォーム（OS）に応じた開発環境をセットアップする必要があります。多くの場合には、ビルドに必要なコンパイラやプロジェクト管理ツール、ソースコードエディターなどがまとめた、**統合開発環境**（Integrated Development Environment、IDE）を導入するのがよいでしょう。

## App 2.1 Windows 用統合開発環境

Windowsでは、Windowsを開発しているMicrosoft社が**Visual Studio**という統合開発環境を公開しているので、これを使うのが一般的です。

### App 2.1.1 ダウンロードとインストール

- Visual Studio のダウンロード：

URL <https://visualstudio.microsoft.com/downloads>

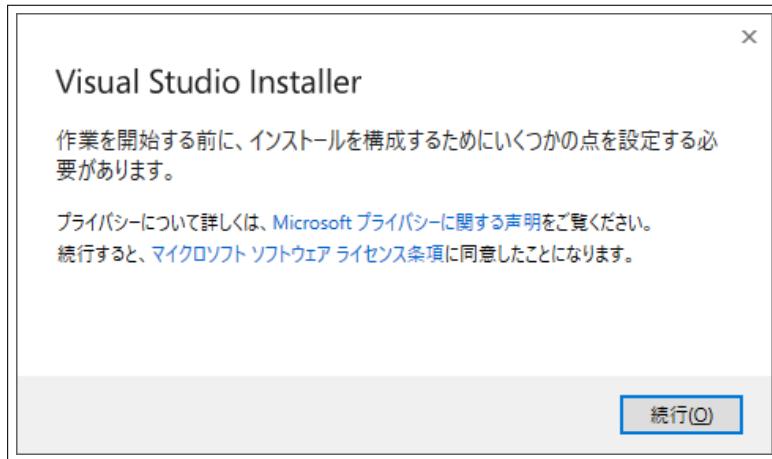
Visual Studioにはいくつかエディションがありますが、個人利用で開発するだけであれば無償のCommunity版で十分です。会社などでチームを組んで開発をするような場合には、Professional版やEnterprise版といったものが必要になることがあります、現時点ではそこまでのものは不要です。

## ダウンロード



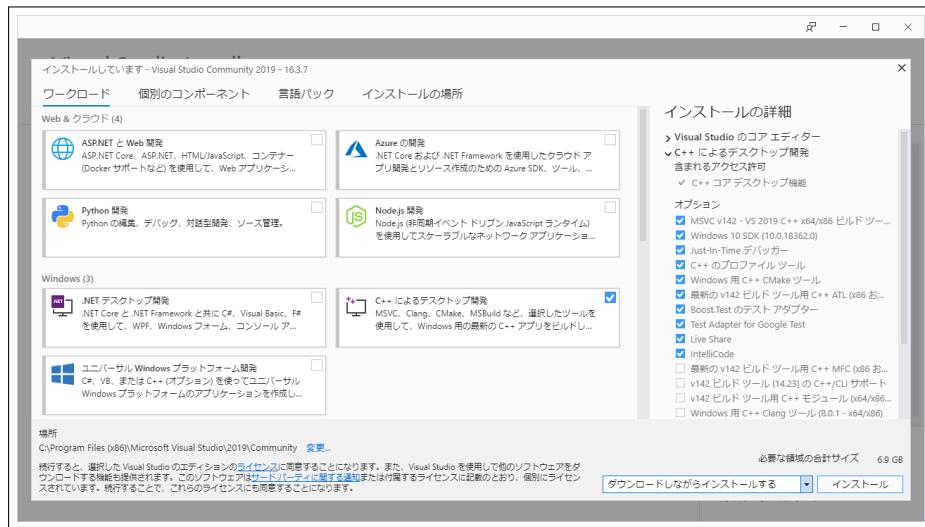
❖ 図 App2.1 Visual Studio のダウンロード

ダウンロードしたインストーラーを実行すると設定を求められますが、[続行] をクリックすると必要な設定が自動的に行われます（図 App2.2）。

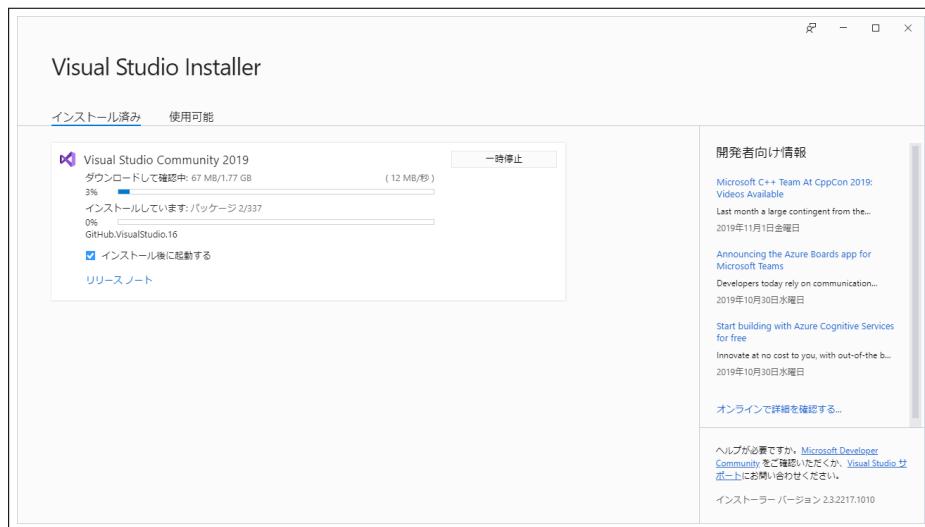


❖ 図 App2.2 事前設定を求めるダイアログ

設定が完了するとインストールする項目を選択する画面に移ります。今は C++ のプログラミングができれば問題ないので、[C++ によるデスクトップ開発] を選択して（図 App2.3）、右下の [インストール] をクリックします（図 App2.4）。もし他の言語や環境もインストールしたい場合にはこの時点では追加してもよいのですが、あとから追加することも可能です。



❖図 App2.3 インストールする項目の選択



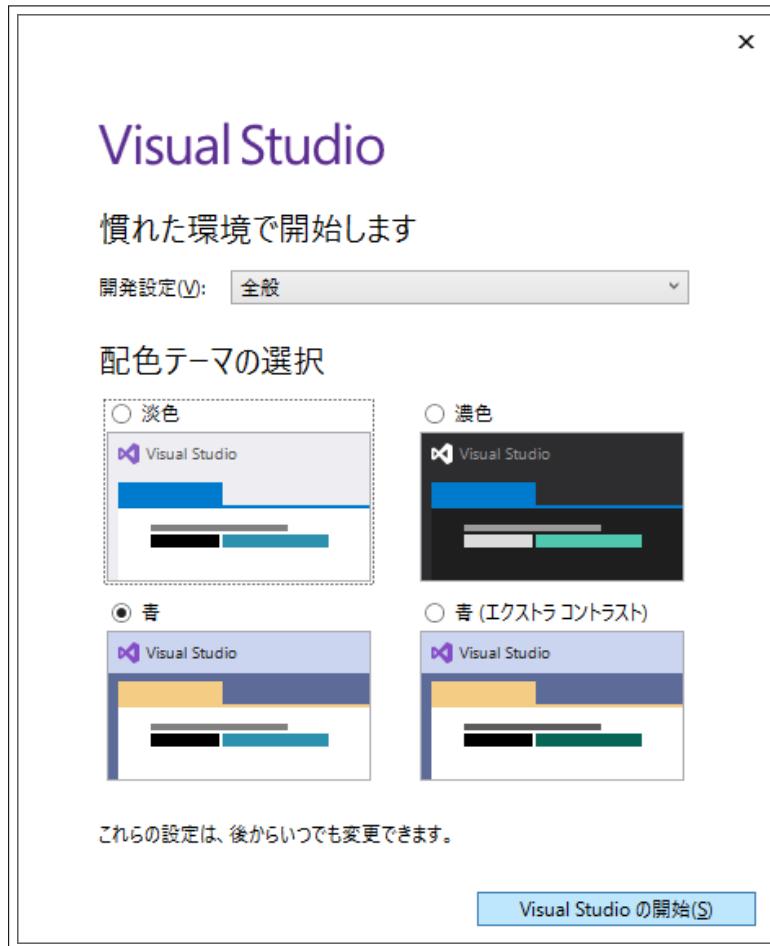
❖図 App2.4 インストールの進行中

インストールが完了すると自動的に Visual Studio が起動します（図 App2.5）。最初は Microsoft アカウントでのサインインを求められますが、[後で行う。] を選択してスキップすることもできます。



❖ 図 App2.5 初期起動画面

テーマの選択は好きなものを選んで問題ありません（図 App2.6）。

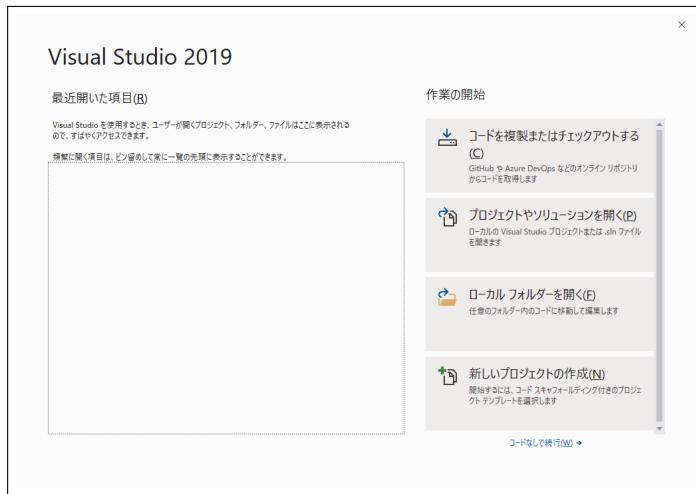


❖図 App2.6 テーマの選択

### App 2.1.2 ソリューション（プロジェクト）の作成

Visual Studio では最も大きな単位をソリューションと呼びます。プロジェクトはソリューションの中に複数作ることができ、それぞれ個別のプログラムをビルドします。最初は 1 プロジェクトだけのソリューションを作ることがほとんどだと思いますが、巨大なプログラムになると、プロジェクトを分けて管理することができます。

プロジェクトを作るには【新しいプロジェクトの作成(N)】を選択します(図 App2.7)。



❖図 App2.7 新しいプロジェクトの作成

次の画面ではプロジェクトの種類を選びます。今回はコンソール上で動くプログラムを作るので、【コンソールアプリ】を選択します(図 App2.8)。



❖図 App2.8 コンソールアプリ

最後にプロジェクトの名前とプロジェクトを保存するディレクトリを入力します（図 App2.9）。



❖ 図 App2.9 プロジェクト名と保存場所の設定

### App 2.1.3 単一ファイルのコンパイル

コンソールアプリを作成した直後は `Hello, world` がデフォルトのプログラムとして記述されているのでこれを実行してみましょう（図 App2.10）。

プログラムをビルドして実行するには、画面上部中央にある [▶ローカル Windows デバッガ] をクリックします（図 App2.11）。

```
ConsoleApplication1.cpp  x
ConsoleApplication1
1 //////////////////////////////////////////////////////////////////////////////
2 // ConsoleApplication1.cpp : このファイルには "main" 関数が含まれています。プログラム実行の開始点
3 //
4 #include <iostream>
5
6 int main()
7 {
8     std::cout << "Hello World!" << std::endl;
9 }
10
11 //////////////////////////////////////////////////////////////////////////////
12 // プログラムの実行: Ctrl + F5 または [デバッグ] > [デバッグなしで開始] メニュー
13 // プログラムのデバッグ: F5 または [デバッグ] > [ティックの開始] メニュー
14
15 // 作業を開始するためのヒント:
16 // 1. ソリューション エクスプローラー ワインダウを使用してファイルを追加/管理します
17 // 2. タスク ワインダウを使用して出力と他のメッセージを表示します
18 // 3. 出力 ワインダウを使用してビルド結果を表示します
19 // 4. ヘルプ ワインダウを使用してヘルプ情報を表示します
20 // 5. [プロジェクト] > [新しい項目の追加] を選択して新しいコード ファイルを作成するか、[プロ
21 // 6. 後ほどこのプロジェクトを再び開く場合、[ファイル] > [開く] > [プロジェクト] と移動して
```

❖図 App2.10 プロジェクト作成直後



❖図 App2.11 実行ボタン

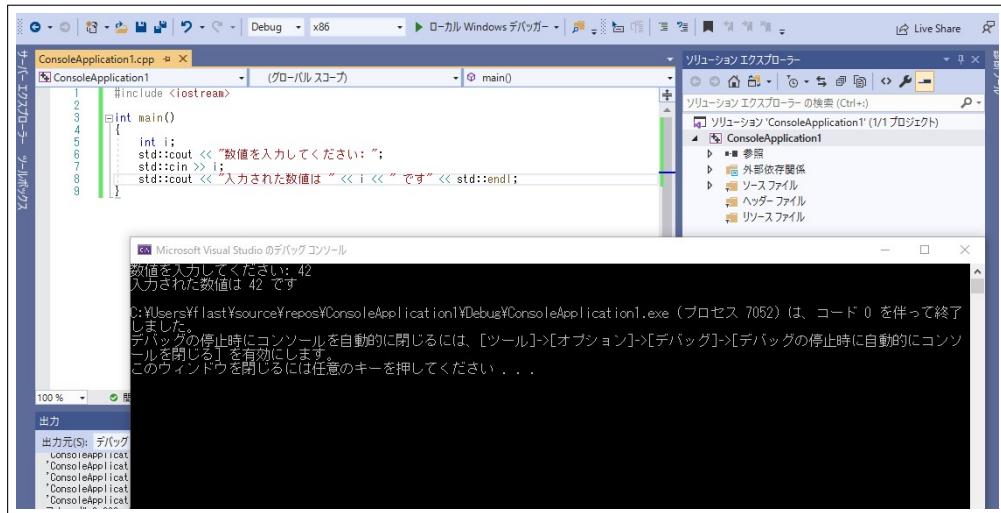
実行結果は図 App2.12 のようになります。

```
Microsoft Visual Studio のデバッグコンソール
Hello World!
C:\Users\last\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe (プロセス 1452) は、コード 0 を伴って終了しました。
デバッグの停止時にコンソールを自動的に閉じるには、「ツール」->「オプション」->「デバッグ」->「デバッグの停止時に自動的にコンソールを閉じる」を有効にします。
このウィンドウを閉じるには任意のキーを押してください . . .
```

❖図 App2.12 Hello, world 実行結果

## App 2.1.4 標準入力

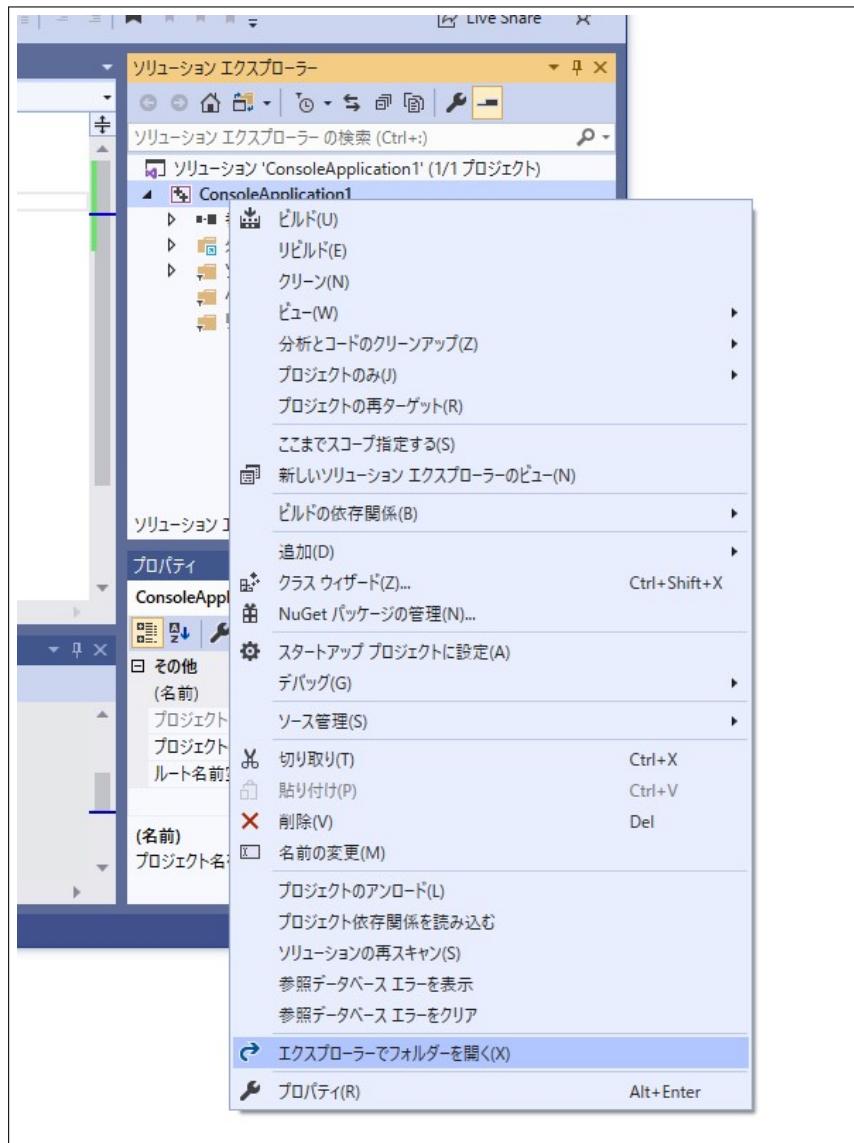
標準入力は実行時に出現したコンソールを使って入力します（図 2.13）。標準入力も標準出力も同じコンソールを使うので、Wandbox のときのように表示が崩れることはありません。



❖図 App2.13 標準入力

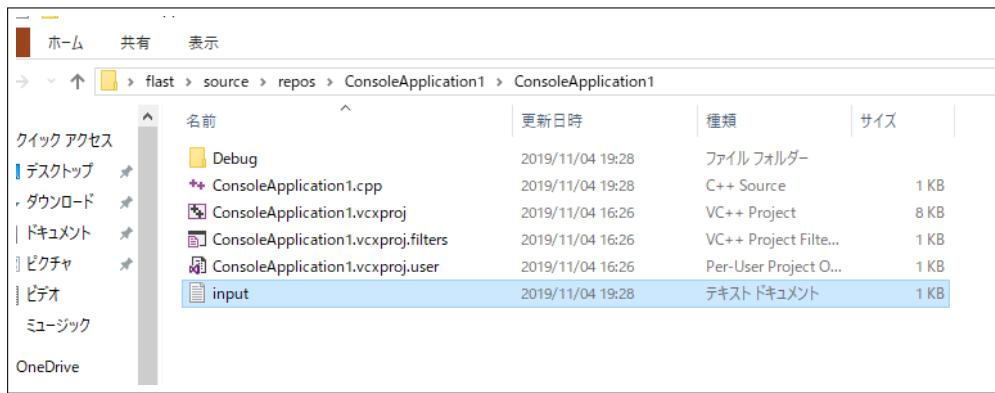
## App 2.1.5 ファイル I/O

ファイル I/Oを行うときの相対パスはプロジェクトのディレクトリが起点になります。そのディレクトリを簡単に開くには、ソリューションエクスプローラーのプロジェクトを右クリックして出てきたメニューから【エクスプローラーでフォルダーを開く】を選択します（図 App2.14）。

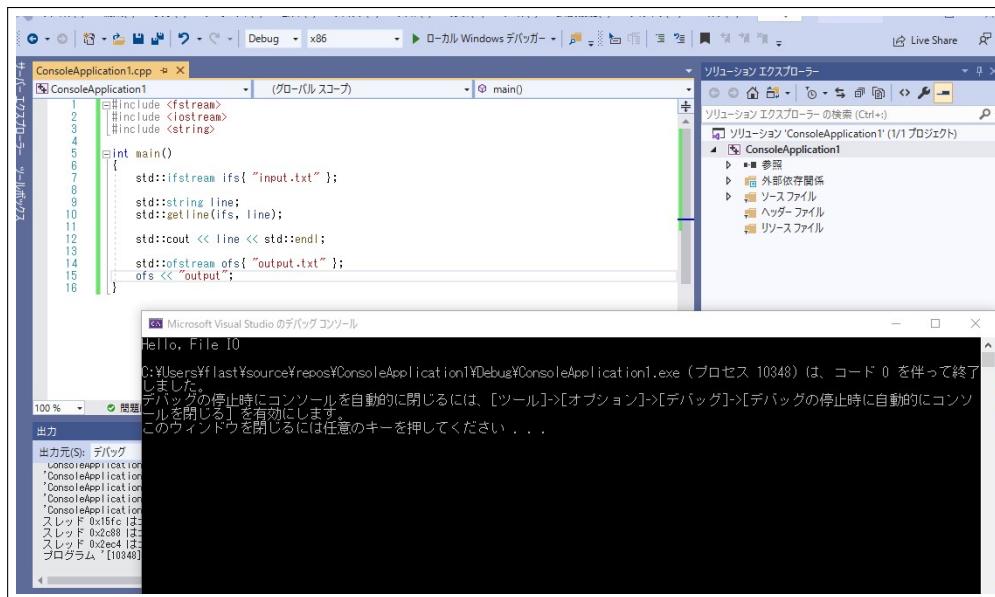


❖図 App2.14 エクスプローラーでフォルダーを開く

開かれたディレクトリに入力を使うファイルを配置したら（図 App2.15）、プログラムを実行すればファイル入力することができます（図 App2.16）。



❖図 App2.15 入力ファイルの配置



❖図 App2.16 ファイル I/O の実行

出力ファイルも入力ファイルと同じ場所に出力されます（図 App2.17）。

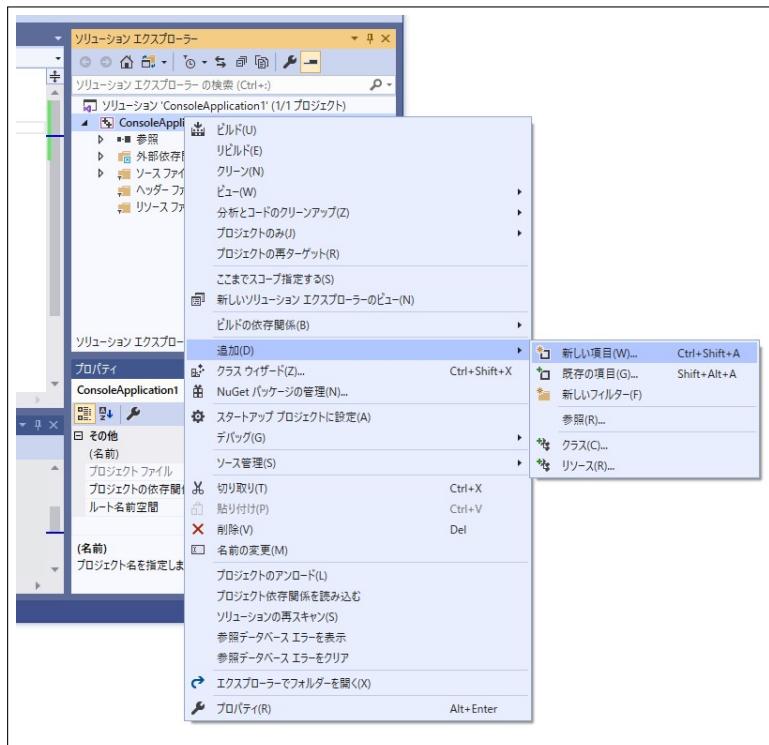
The screenshot shows a Windows File Explorer window. The path is displayed as: flast > source > repos > ConsoleApplication1 > ConsoleApplication1. The left sidebar includes icons for network drives, local drives, and project files. The main pane displays a list of files and folders:

名前	更新日時	種類	サイズ
Debug	2019/11/04 19:34	ファイル フォルダー	
ConsoleApplication1.cpp	2019/11/04 19:34	C++ Source	1 KB
ConsoleApplication1.vcxproj	2019/11/04 16:26	VC++ Project	8 KB
ConsoleApplication1.vcxproj.filters	2019/11/04 16:26	VC++ Project Filte...	1 KB
ConsoleApplication1.vcxproj.user	2019/11/04 16:26	Per-User Project O...	1 KB
input	2019/11/04 19:28	テキスト ドキュメント	1 KB
output	2019/11/04 19:34	テキスト ドキュメント	1 KB

❖図 App2.17 出力ファイル

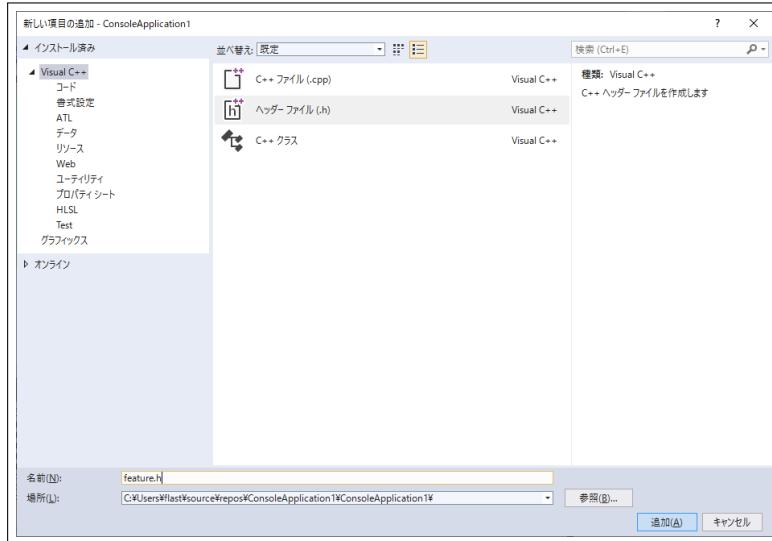
## App 2.1.6 分割コンパイル

分割コンパイルするにはまず追加のソースファイルを作成し、プロジェクトに追加する必要があります。ソリューションエクスプローラーのプロジェクトを右クリックして、コンテキストメニューから [追加 (D)] → [新しい項目 (W)] を選択します（図 App2.18）。



❖図 App2.18 新しい項目

「ヘッダー ファイル」と「C++ ファイル」をファイル名を入力してそれぞれ作成します（図 App2.19）。



❖図 App2.19 ファイルの作成

ソースファイルが追加できたらそれぞれのファイルに必要なプログラムを記述します（図App2.20～図App2.22）。

```
#include "feature.h"
int main()
{
    show_message();
```

❖図 App2.20 ConsoleApplication1.cpp

The screenshot shows a C++ IDE interface with the following details:

- Project name: ConsoleApplication1
- File tabs: feature.cpp, feature.h (highlighted), ConsoleApplication1.cpp
- Code editor content (feature.h):

```
1 #pragma once
2
3 void show_message();
```

❖図 App2.21 feature.h

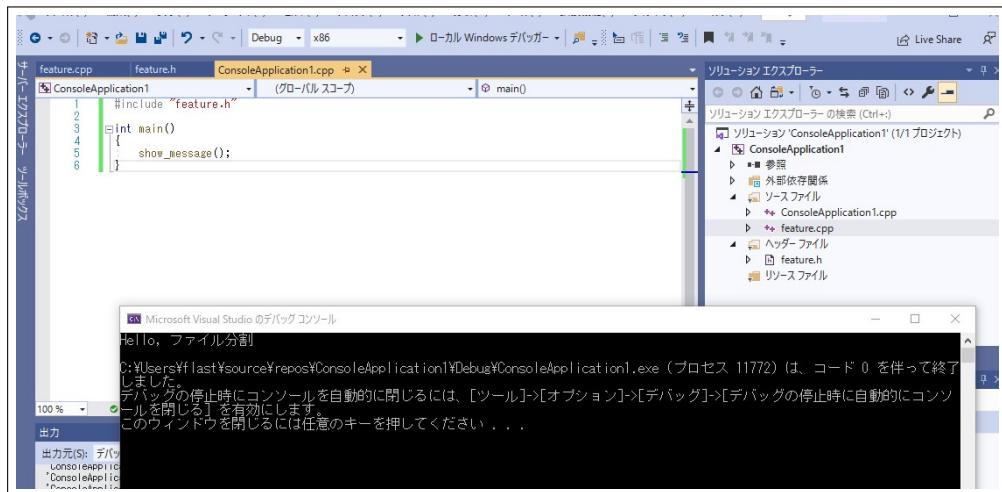
The screenshot shows a C++ IDE interface with the following details:

- Project name: ConsoleApplication1
- File tabs: feature.cpp, feature.h, ConsoleApplication1.cpp
- Code editor content (feature.cpp):

```
1 #include <iostream>
2
3 void show_message()
4 {
5     std::cout << "Hello, ファイル分割" << std::endl;
6 }
```

❖図 App2.22 feature.cpp

プログラムを記述できたら実行すると、すべてのソースファイルがビルドされ、プログラムの実行結果が表示されます（図 App2.23）。



❖ 図 App2.23 分割コンパイルの実行

## App 2.2 macOS 用統合開発環境

macOS では、Mac を開発している Apple 社が macOS 用の **Xcode** という統合開発環境を公開しているので、これを使うのが一般的です。Microsoft 社も macOS 用の Visual Studio を公開しているので、Visual Studio for Mac を選択することもできます。

本節では Xcode の使い方について説明します。

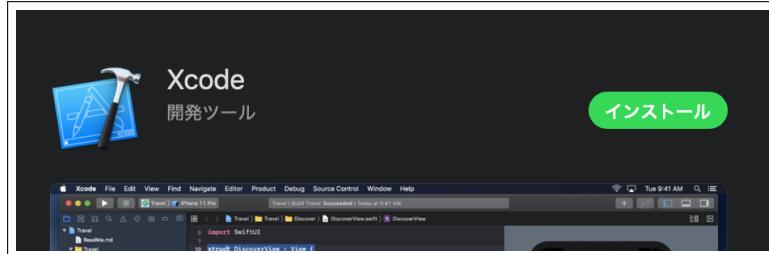
### App 2.2.1 ダウンロードとインストール

Xcode は App Store からインストールすることができます。検索ボックスに「xcode」と入力して検索すると、一番最初に出てくるので「入手」を選択します（図 App2.24）。



❖図 App2.24 App Store で「xcode」と検索

入手を選択するとそのボタンが【インストール】に変わるので、再度クリックします（図 App2.25）。



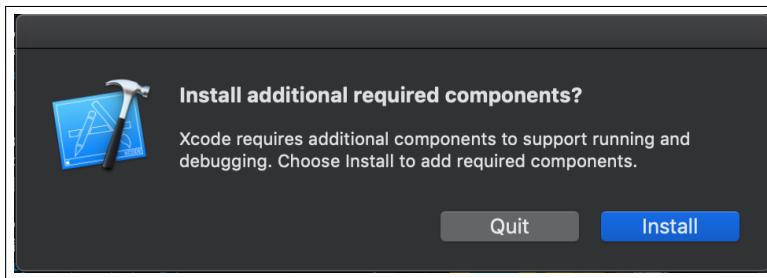
❖図 App2.25 インストールを選択

インストールが完了すると Launchpad から起動できます（図 App2.26）。



❖ 図 App2.26 Xcode の起動

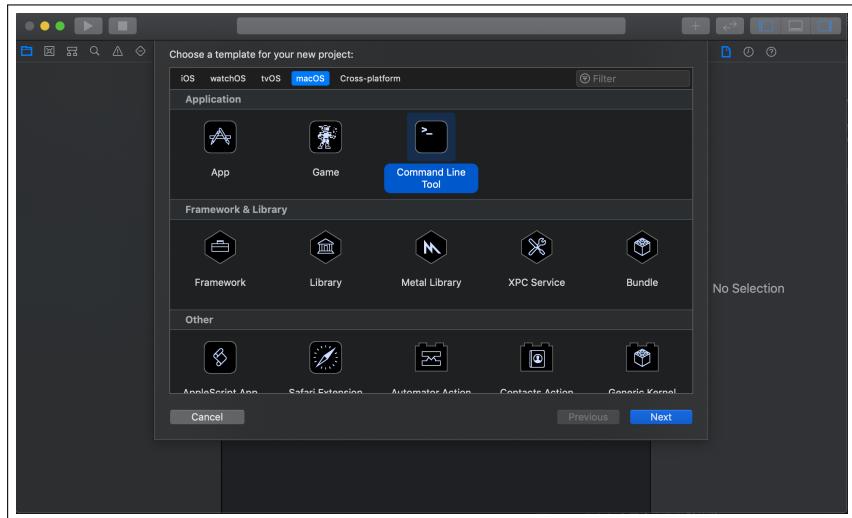
このとき必須コンポーネントのインストールを求められるので、[Install] を選択します（図 App2.27）。



❖ 図 App2.27 必須コンポーネントのインストール

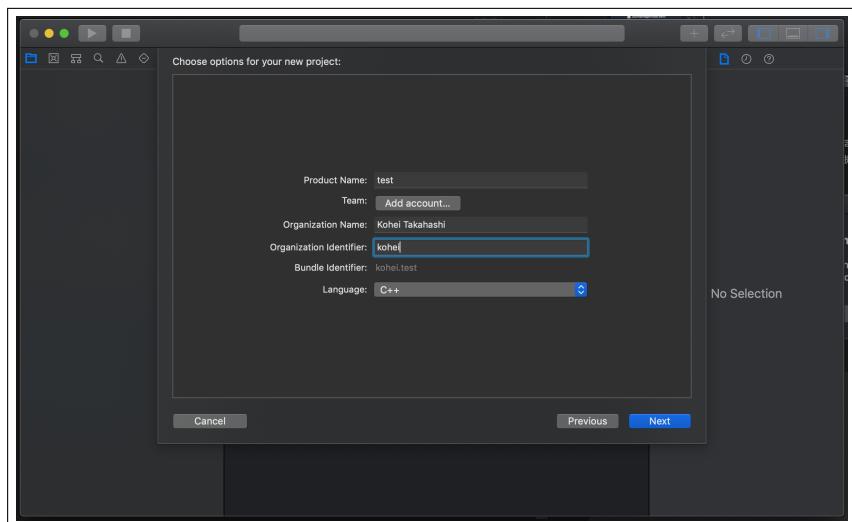
## App 2.2.2 プロジェクトの作成

プロジェクトを作成するには、中央の [Create a new Xcode project] を選択します。今は手元の macOS 上で実行するので、一覧から [macOS] を選択し、さらにその下から [Command Line Tool] を選んで [Next] をクリックします（図 App2.28）。



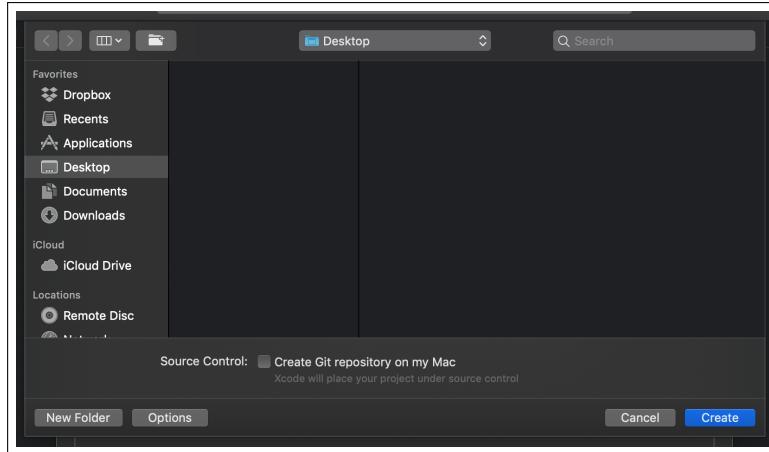
❖図 App2.28 プロジェクト種別の選択

次にプロジェクト名と組織名（Organization Identifier）を入力します。図 App2.29 ではプロジェクト名を「test」に、組織名を私の名前にしていますが、今はどちらも好きなものを入力してかまいません。プロジェクトの言語を C++ にするのは、もちろん忘れないようにしましょう。



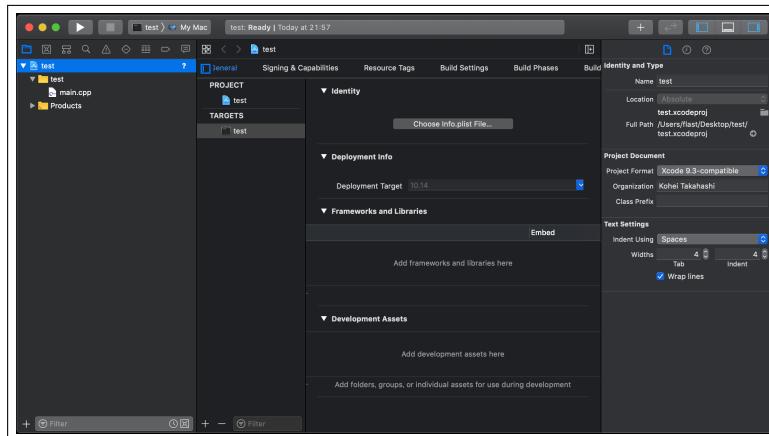
❖図 App2.29 プロジェクト名の入力

最後にプロジェクトを保存する場所を指定して [Create] をクリックします（図 App2.30）。いまのところ [Create Git repository on my Mac] のチェックは外しておいたほうがよいかかもしれません。これは **Git** というバージョン管理システム（Version Control System、VCS）の一つで、ある程度の大きさのプログラムを作る上では必ずといっていいほど必要になるソフトウェアです。今はまだわからなくても、できるだけ早く使い方を身に着けておくと、プログラムの開発をとても効率よくすすめることができます。



❖ 図 App2.30 保存場所の指定

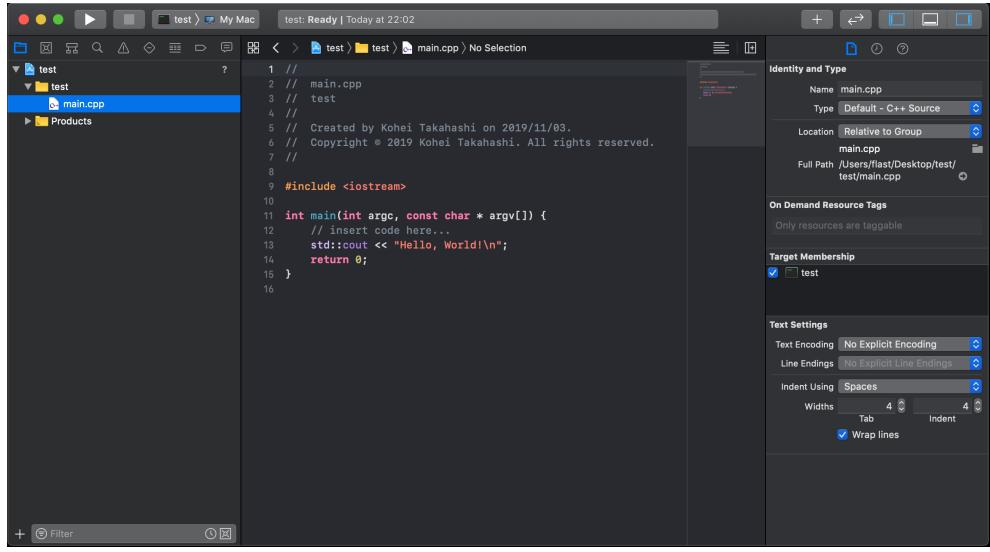
プロジェクトが作成されました（図 App2.31）。



❖ 図 App2.31 作成完了

### App 2.2.3 単一ファイルのコンパイル

プロジェクトを作成した直後は `main.cpp` というソースファイルだけが存在し、これの中身は `Hello, world` となっています（図 App2.32）。



The screenshot shows the Xcode interface with a project named "test". In the left sidebar, "main.cpp" is selected. The main editor area displays the following code:

```
1 //  
2 // main.cpp  
3 // test  
4 //  
5 // Created by Kohei Takahashi on 2019/11/03.  
6 // Copyright © 2019 Kohei Takahashi. All rights reserved.  
7 //  
8 //  
9 #include <iostream>  
10  
11 int main(int argc, const char * argv[]) {  
12     // insert code here...  
13     std::cout << "Hello, World!\n";  
14     return 0;  
15 }
```

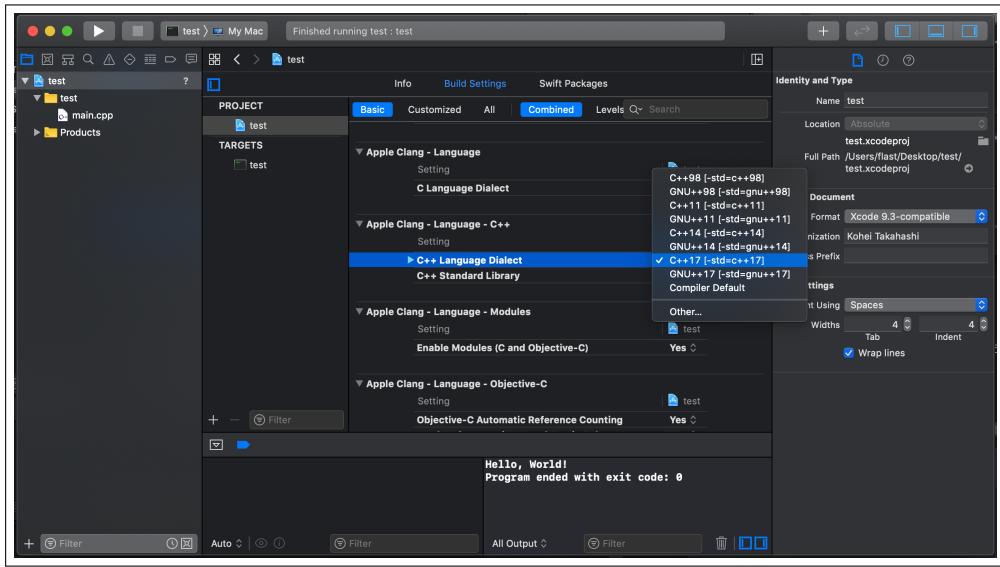
The right panel shows the file's properties:

- Identity and Type**: Name is "main.cpp", Type is "Default - C++ Source".
- Location**: Relative to Group, main.cpp, Full Path: /Users/kohei/Desktop/test/test/main.cpp.
- On Demand Resource Tags**: Only resources are taggable.
- Target Membership**: The checkbox for "test" is checked.
- Text Settings**: Text Encoding: No Explicit Encoding, Line Endings: No Explicit Line Endings, Indent Using: Spaces (Widths: Tab 4, Indent 4), Wrap lines is checked.

❖図 App2.32 main.cpp

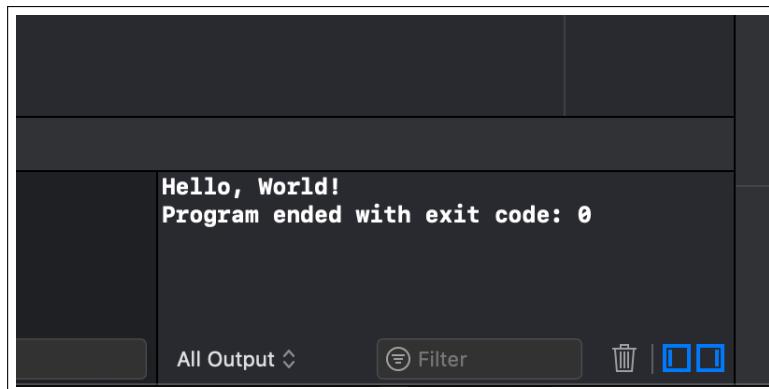
取りあえず今はこれを実行して試してみましょう。

ただ、その前に C++17 モードでコンパイルされるようにプロジェクトの設定を変更する必要があります。左側にあるツリー状になっている部分をプロジェクトナビゲーターといいますが、その一番上にあるプロジェクトを選択します。すると中央のエディター領域が設定画面に変化します。そこから [PROJECT] から現在のプロジェクトを、さらに上部の [Build Settings] を選択します。[Basic] の項目の真ん中あたりにある [Apple Clang - Language C++] を探して、さらにその下にある [C++ Language Dialect] の値として「C++17」を選びます（図 App2.33）。



❖図 App2.33 言語仕様を C++17 に変更

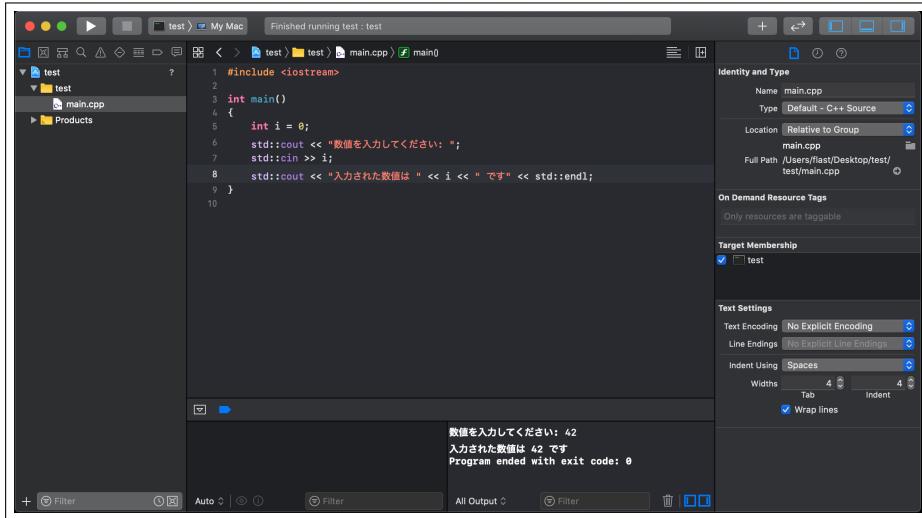
言語仕様を変更できたら、プログラムのコンパイルと実行までを一気にするには左上にある [▷] をクリックします。コンパイルに成功して実行されるとウィンドウ下に新たな領域（デバッガ領域）が出現して、出力結果を表示します（図 App2.34）。



❖図 App2.34 プログラムのコンパイルと実行

## App 2.2.4 標準入力

標準入力についても、Xcode 上から入力することができます。デバッグ領域は標準出力が表示される領域ですが、ここで何か入力すると、その内容は標準入力へと渡されます（図 App2.35）。



The screenshot shows the Xcode interface with a project named 'test'. In the left sidebar, 'main.cpp' is selected. The main editor window contains the following C++ code:

```
#include <iostream>
int main()
{
    int i = 0;
    std::cout << "数値を入力してください: ";
    std::cin >> i;
    std::cout << "入力された数値は " << i << " です" << std::endl;
}
```

The bottom right corner of the editor shows the output of the program:

```
数値を入力してください: 42
入力された数値は 42 です
Program ended with exit code: 0
```

The right side of the screen displays the 'Identity and Type' inspector for 'main.cpp', showing it is a 'C++ Source' file.

❖ 図 App2.35 標準入力

## App 2.2.5 ファイル I/O

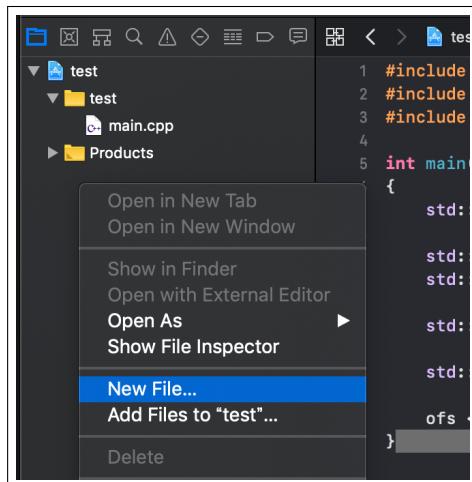
まずはファイル I/O をするプログラムを準備しておきます（図 App2.36）。

```
1 #include <iostream>
2 #include <fstream>
3 #include <string>
4
5 int main()
6 {
7     std::ifstream ifs("input");
8
9     std::string line;
10    std::getline(ifs, line);
11
12    std::cout << line << std::endl;
13
14    std::ofstream ofs("output");
15
16    ofs << "file output" << std::endl;
17 }
```

hoge  
Program ended with exit code: 0

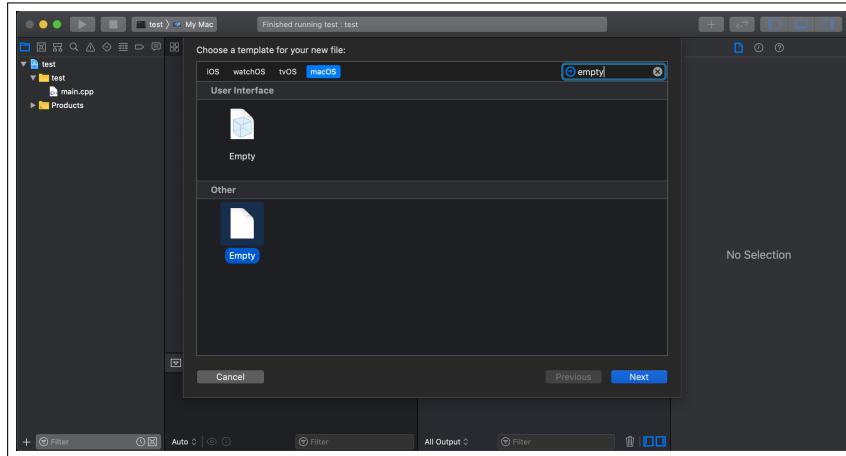
❖図 App2.36 ファイル I/O をするプログラム

次に入力を使うファイルを作成します。これはプロジェクトナビゲーター上で [control] キーを押しながらクリックします。出てきたコンテキストメニューから [New File...] を選択します（図 App2.37）。

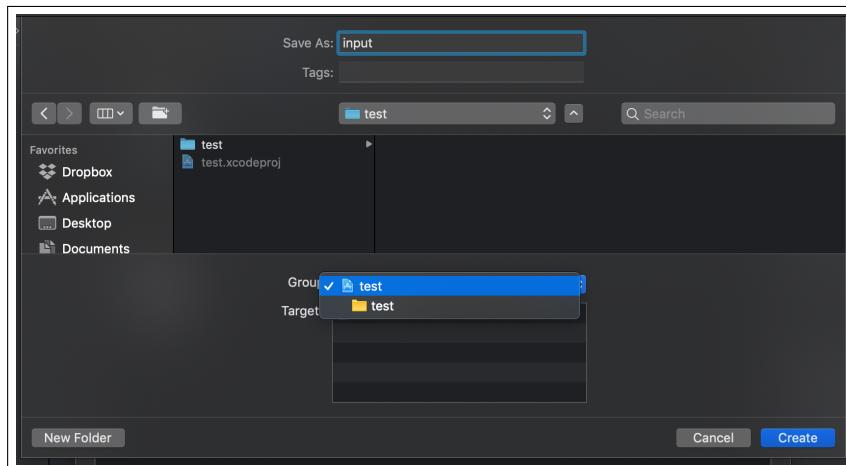


❖図 App2.37 コンテキストメニュー

ファイル作成画面の検索ボックスで「empty」と入力して空ファイルを探し出し、ファイル名や格納先を指定してファイルを作成します（図 App2.38）。このときファイルの作成先では、中央の Group というところから一番上を選択してください（図 App2.39）。

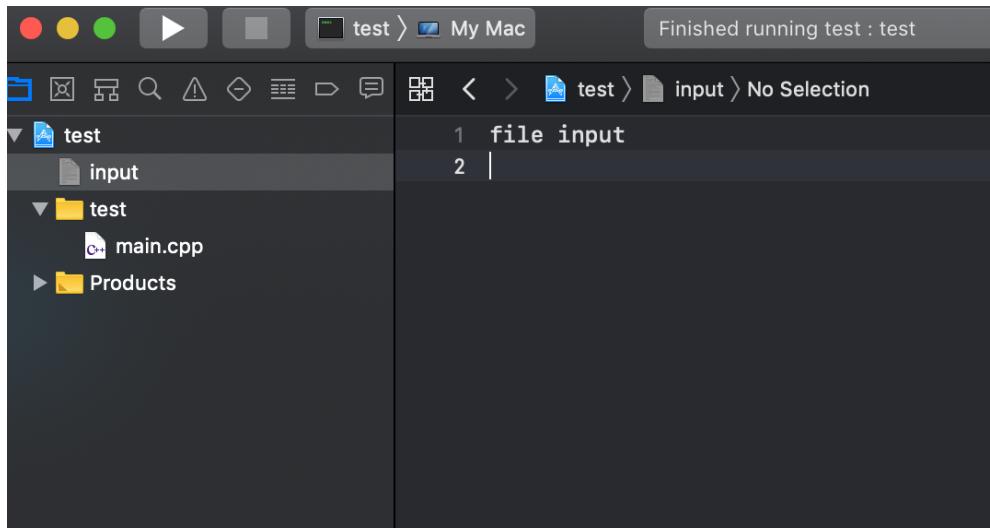


❖図 App2.38 空ファイルの作成



❖図 App2.39 ファイル名の指定

入力ファイルの中身を記述します（図 App2.40）。



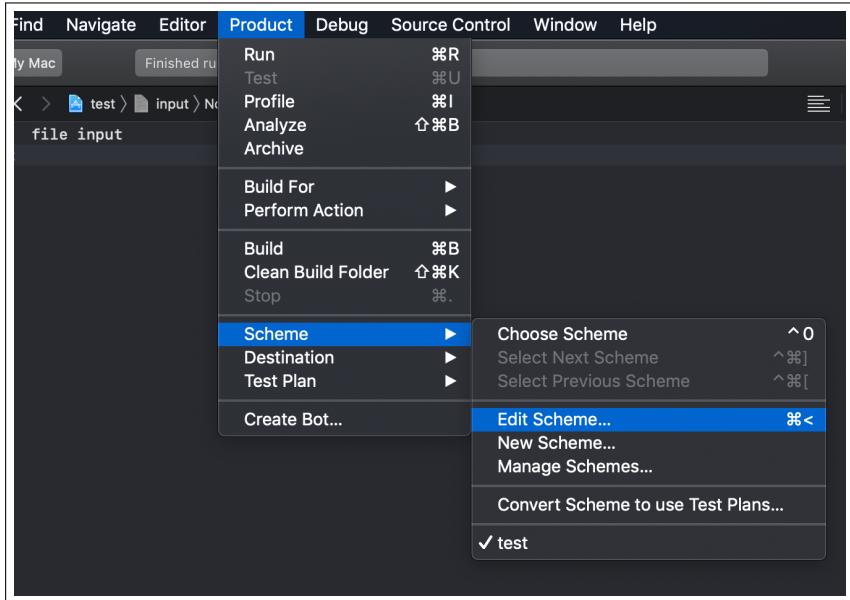
The screenshot shows the Xcode interface. The top bar displays the project name "test" and the message "Finished running test : test". The left sidebar shows the project structure: a "test" group containing an "input" folder, which itself contains a "test" folder and a "main.cpp" file. The right pane is a code editor with the following content:

```
1 file input
2 |
```

❖図 App2.40 入力ファイルの中身

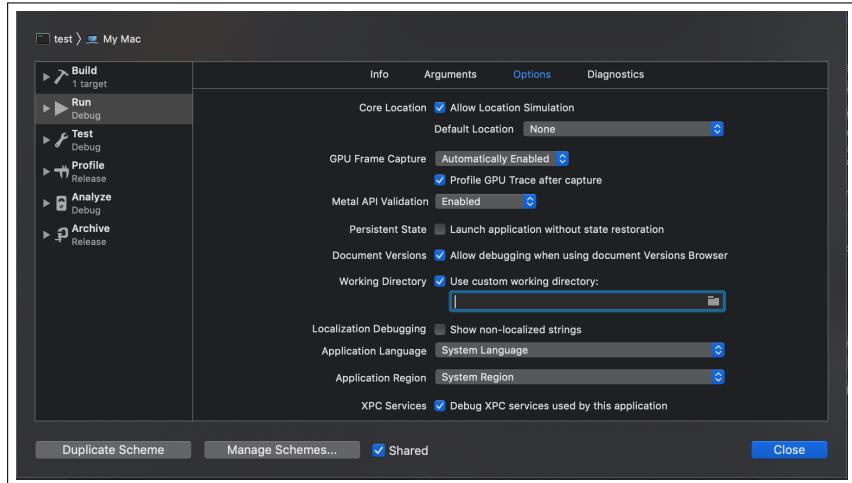
実はこれだけではまだうまくいかず、実行ディレクトリ（working directory）を変更する必要があります。ファイル I/O のときにファイルパスをちゃんと指定すればこの設定は不要なのですが、何度もプログラムを書き換えてテストするのであれば、実行ディレクトリを変更するほうが楽です。

[Product] メニューから [Scheme] → [Edit Scheme] を選択します（図 App2.41）。



❖ 図 App2.41 Edit Scheme の選択

[Run] を選択しさらに [Option] を選択すると、プログラムの実行に関する設定項目が現れます。この中の [Working Directory] をプロジェクトのディレクトリにします（図 App2.42）。



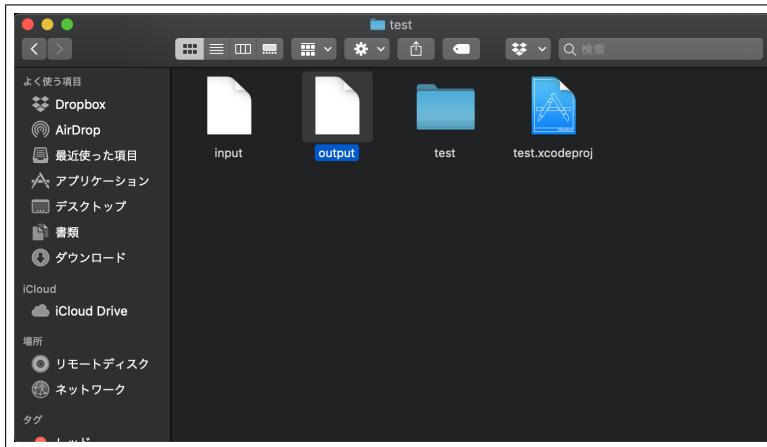
❖図 App2.42 実行ディレクトリの設定

ここまで設定するとファイル I/O ができるようになります（図 App2.43）。出力ファイルはプロジェクトのディレクトリに作成されます（図 App2.44）。

```
file input
Program ended with exit code: 0
```

Below the output window are buttons for 'All Output' (with a dropdown arrow) and 'Filter' (with a magnifying glass icon). To the right are icons for trash, copy, and paste.

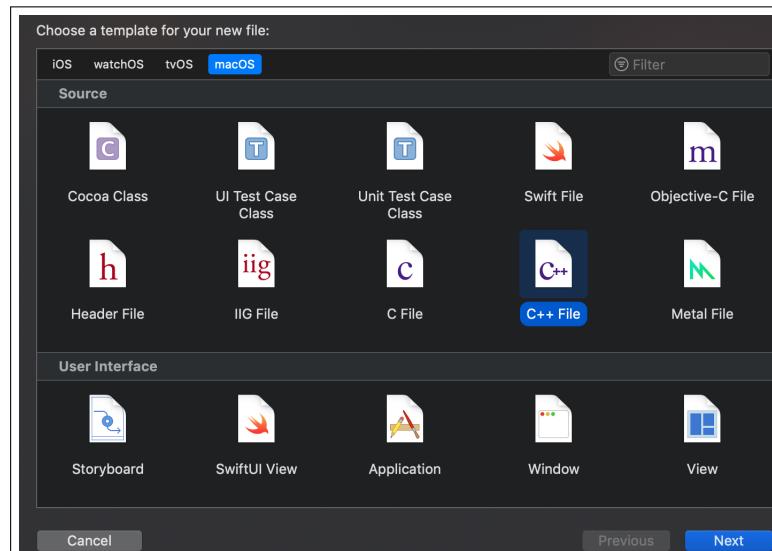
❖図 App2.43 ファイル I/O の実行結果



❖ 図 App2.44 出力ファイル

## App 2.2.6 分割コンパイル

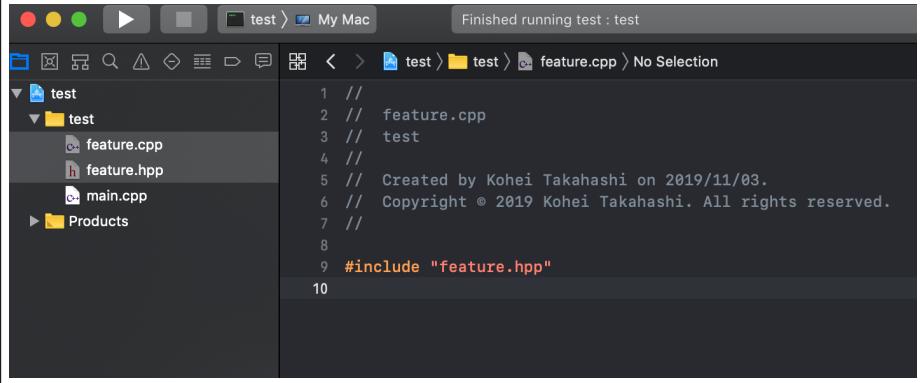
Xcode で分割コンパイルをするには、まず空ファイルを作成したのと同じ要領で追加のソースファイル (C++ File) を作成します (図 App2.45)。



❖ 図 App2.45 C++ ソースファイルの追加

そしてソースファイル名を入力します。このとき拡張子まで記入する必要はありません。また、[Also create a header file] のチェックが付いていることを確認します。

ソースファイルの保存先を指定します（図 App2.46）。これは `main.cpp` が格納されているフォルダーを指定すればよいです。

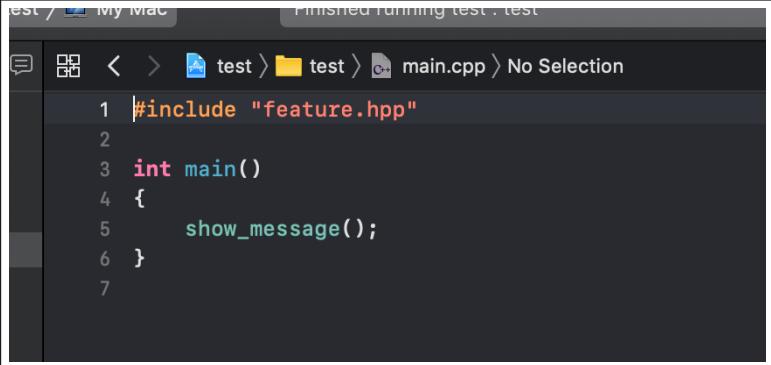


The screenshot shows the Xcode interface. The left sidebar displays a project structure with a 'test' group containing 'feature.cpp', 'feature.hpp', and 'main.cpp'. The right pane shows the code for 'main.cpp':

```
1 //  
2 // feature.cpp  
3 // test  
4 //  
5 // Created by Kohei Takahashi on 2019/11/03.  
6 // Copyright © 2019 Kohei Takahashi. All rights reserved.  
7 //  
8  
9 #include "feature.hpp"  
10
```

❖ 図 App2.46 ソースファイルが作成された

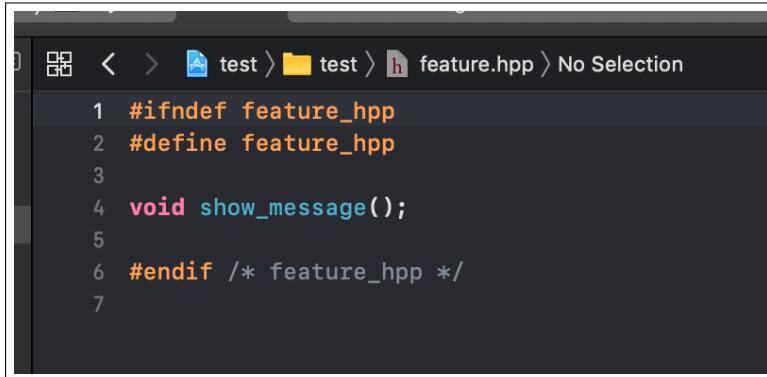
それぞれのソースにプログラムを記述します（図 App2.47～図 App2.49）。このとき、作成したヘッダーファイルをインクルードするには、`#include "filename"` の形式を使用する必要があります。



The screenshot shows the Xcode interface with the code editor open to 'main.cpp'. The code contains a single line of code that includes the 'feature.hpp' header:

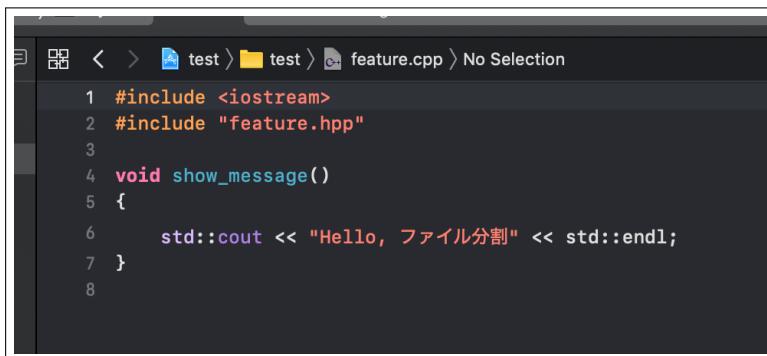
```
1 #include "feature.hpp"
```

❖ 図 App2.47 main.cpp



```
1 #ifndef feature_hpp
2 #define feature_hpp
3
4 void show_message();
5
6 #endif /* feature_hpp */
7
```

❖図 App2.48 feature.hpp



```
1 #include <iostream>
2 #include "feature.hpp"
3
4 void show_message()
5 {
6     std::cout << "Hello, ファイル分割" << std::endl;
7 }
```

❖図 App2.49 feature.cpp

これまでどおり実行すると、プログラムのコンパイル、リンクが行われて実行結果がデバッグ領域に表示されます（図 App2.50）。



❖図 App2.50 分割コンパイルの実行

## App 2.3 Linux 用開発環境

Linux ではディストリビューションごとに提供されている環境が異なるので、それぞれお使いのディストリビューションによってセットアップ方法が異なります。Linux で利用できる IDE では Eclipse (エクリプス) や Qt (キュート、キューティー) Creator、KDevelop などがあります。

ただ、Linux で開発をするという場合には、コンパイラーやビルドツールを端末エミュレーター (コンソール) からそれぞれ操作することが多いはずなので、ここではほとんどのディストリビューションで提供しているコンパイラである **GCC** の使い方について説明します。

### App 2.3.1 インストール

GCC はパッケージマネージャーからインストールできます。RedHat 系 (RHEL、CentOS、Fedora) では `yum` や `dnf` といったコマンド、Debian 系 (Debian や Ubuntu) では `apt` コマンドを使います。それ以外のディストリビューションについては各自で適宜読み替えてください。

#### ▶ RHEL / CentOS (7 まで)

```
$ sudo yum install gcc-c++
```

### ▶ CentOS (8 以降) / Fedora

```
$ sudo dnf install gcc-c++
```

### ▶ Debian / Ubuntu

```
$ sudo apt update  
$ sudo apt install g++
```

インストールが完了したらコマンドを実行してインストールされていることを確認してください。

### ▶ インストールの確認

```
$ g++ --version  
g++ (GCC) 9.2.0  
Copyright (C) 2019 Free Software Foundation, Inc.  
This is free software; see the source for copying conditions. There is NO  
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

## App 2.3.2 単一ファイルのコンパイル

ソースコードを何かしらのテキストエディターで作成したら、GCC のコンパイル引数でそのファイルを指定すればコンパイルされた実行形式ファイルが作成されます。このとき、デフォルトでは作成されたファイルは `a.out` という名前になります。

▶ リスト App2.1 hello\_world.cpp

```
#include <iostream>

int main()
{
    std::cout << "Hello, world" << std::endl;
}
```

▶ コンパイルと実行

```
$ g++ hello_world.cpp

$ ./a.out
Hello, world
```

a.out という名前はあまりにわかりづらいのでファイル名を変えたい場合には、コンパイラに-o オプションを与えると出力ファイル名を変更できます。

▶ ファイル名を変えてコンパイル

```
$ g++ -o hello_world hello_world.cpp

$ ./hello_world
Hello, world
```

2019年11月現在最新のバージョンであるGCC 9.2では、デフォルトでC++14モードでコンパイルされます。C++17やそれ以降の標準でコンパイルするには-stdオプションを追加してどの言語仕様を使うか指定する必要があります。デフォルトで使われる言語仕様はバ

ジョンによって異なるので、常に指定するようにすると間違いが少なくなります。

#### ▶ リスト App2.2 C++17 で書かれた cpp17.cpp

```
#include <iostream>
#include <vector>

int main()
{
    std::vector int_vector = {0, 1, 2};
    for (auto v : int_vector)
    {
        std::cout << v << std::endl;
    }
}
```

#### ▶ 言語仕様を指定してコンパイル

```
$ g++ -o cpp14 cpp17.cpp
cpp17.cpp: In function 'int main()':
cpp17.cpp:6:17: error: missing template arguments before 'int_vector'
  6 |     std::vector int_vector = {0, 1, 2};
     |             ^~~~~~
cpp17.cpp:7:19: error: 'int_vector' was not declared in this scope
  7 |     for (auto v : int_vector)
     |
$ g++ -std=c++17 -o cpp17 cpp17.cpp

$ ./cpp17
0
1
2
```

#### ◆ Column : 警告レベルの引き上げ

GCC のオプションに何も与えなければ、デフォルトではほとんど警告を出さないようになっています。しかし警告は間違えている可能性が高い部分をコンパイラが指摘してくれる機能なので、基本的に多く出力されるようにしておいて、警告がなくなるまでプログラムを修正していくのが望ましいです。

GCC に-Wall というオプションを渡すと基本的な警告がほとんど有効になります。少なくとも普段から-Wall を付けてプログラムを書く癖を付けるのがよいでしょう。

より細かい警告はさらに-Wextra オプションを付けると有効になりますが、こちらは今の段階では付けなくてもかまいません。

### App 2.3.3 標準入力

コンパイルと実行を端末エミュレーター上で行っているのであれば、標準入力もそのまま端末エミュレーター上から入力できます。

#### ▶ リスト App2.3 標準入力 (stdin.cpp)

```
#include <iostream>

int main()
{
    int i;
    std::cout << "数値を入力してください: ";
    std::cin >> i;
    std::cout << "入力された数値は " << i << " です" << std::endl;
}
```

## ▶ 実行結果

```
$ g++ -Wall -std=c++17 stdin.cpp  
$ ./a.out  
数値を入力してください: 42  
入力された数値は 42 です
```

## App 2.3.4 ファイル I/O

ファイル I/O をするときのファイルパスは、プログラムを実行したときのディレクトリからの相対パスもしくはファイルの絶対パスとなります。相対パスでプログラムを作る際には実行するときのディレクトリが異なれば違うファイルを扱ってしまうことに注意してください。

### ▶ リスト App2.4 fileio.cpp

```
#include <iostream>  
#include <fstream>  
#include <string>  
  
int main()  
{  
    std::ifstream ifs{"input"};  
  
    std::string line;  
    std::getline(ifs, line);  
  
    std::cout << line << std::endl;  
  
    std::ofstream ofs{"output"};  
    ofs << "Hello, output" << std::endl;  
}
```

## ▶ ファイル I/O

```
$ echo "Hello, File I/O" > input  
  
$ g++ -Wall -std=c++17 fileio.cpp  
  
$ ./a.out  
Hello, File I/O  
  
$ cat output  
Hello, output
```

### App 2.3.5 分割コンパイル

コマンドラインから GCC を呼び出して分割コンパイルするには 2 通りの方法があります。

まず 1 つ目は、すべてのソースファイルを 1 回のコマンドで指定してコンパイルする方法です。この場合コンパイラーがリンクまですべて行って、実行可能ファイルを直接出力します。

もう 1 つはソースファイルをひとつひとつコンパイルしてオブジェクトファイルを作成し、最後にそれらをリンクする方法です。この方法では変更のあったソースファイルのみをコンパイルし直して再度リンクすればよいですが、ソースファイルがたくさんある場合には管理が大変になります。

本来はビルドを効率的にするために GNU (グニュー) Make などを使ってツールに任せるのがよいですが、ここでは直接コマンドを実行してビルドすることにします。

最初にソースファイルを用意します。

## ▶ リスト App2.5 main.cpp

```
#include "feature.h"  
  
int main()  
{  
    show_message();  
}
```

▶ リスト App2.6 feature.h

```
void show_message();
```

▶ リスト App2.7 feature.cpp

```
#include <iostream>
#include "feature.h"

void show_message()
{
    std::cout << "Hello, 分割コンパイル" << std::endl;
}
```

まずはコンパイラにすべてのソースファイルを渡して実行形式ファイルを直接作成します。このときヘッダーファイルは渡す必要ありません。

▶ 1回で実行可能ファイルを作成する

```
$ g++ -Wall -std=c++17 main.cpp feature.cpp
$ ./a.out
Hello, 分割コンパイル
```

次にソースファイルごとにオブジェクトファイルを作成して最後にリンクする方法です。オブジェクトファイルを作るにはオプションとして-c を付けてコンパイルします。このときやはりヘッダーファイルはコンパイルしません。オブジェクトファイルは、.cpp の部分を.o に置き換えた名前で出力されます。

リンクするときには、コンパイラにリンクしたいオブジェクトファイルを渡すだけで、それらをリンクした実行可能ファイルが作成されます。

▶ オブジェクトファイルを作成してリンクする

```
$ g++ -c -Wall -std=c++17 main.cpp  
  
$ g++ -c -Wall -std=c++17 feature.cpp  
  
$ ls  
feature.cpp feature.o main.cpp main.o  
  
$ g++ -std=c++17 main.o feature.o  
  
$ ./a.out  
Hello, 分割コンパイル
```