

VS CodeでC言語のプロジェクトをデバッグする簡単ガイド

この文書は Ver.0.3 を変更して作成していますので、
節番号が抜けたりしている箇所があります。

Version 0.4.1 (with CodeLLDB)

2026-09-09 荻原剛志

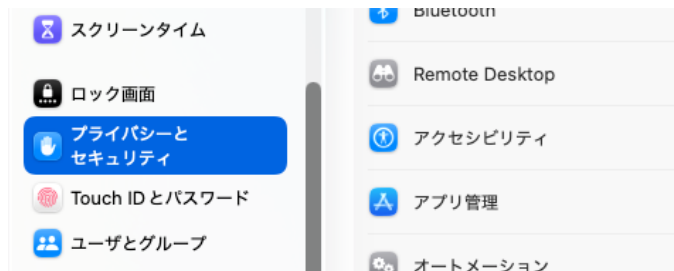
Visual Studio Code を使ってC言語のソフトウェア開発を行う方法、特にデバッガの基本的な使い方とそのため設定について説明する。ただし、簡単なエディタとしてVS Codeを使ったことがあることを前提とする。この文書の内容は導入レベルであって、ソースファイルが単一のディレクトリ内にあるような比較的簡単なプログラムが対象である。

VS Code自体の各種設定や操作の体系は必ずしも統一されておらず、利用方法を学んで開発ができるようになるまでの手間が大きい。利用者が多いため、使い方のノウハウや拡張機能の紹介の情報は多いが、複雑さに拍車をかける混沌とした状況とも言える。拡張機能の追加などに関しては、正確な情報に基づいて慎重に（自己責任で）行なってもらいたい。

1. 準備

1.1 アクセシビリティとフルディスクアクセスの確認

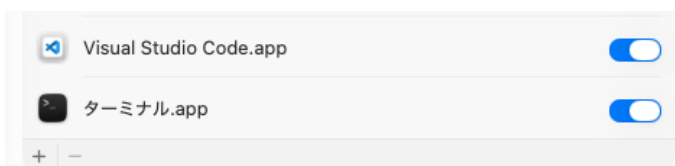
macOS では、アプリケーションが他のプログラムを操作するためにアクセシビリティ（操作権限）の設定を求められることがある。今回は VS Code がターミナルを操作するので、その設定を行う。なお、これまでに何らかの操作ですでに VS Code やターミナルに対する設定が行われている場合もある。



アクセシビリティを確認

macOSの「システム設定」で「プライバシーとセキュリティ」を選択し、さらに「アクセシビリティ」の項目を開く。「ターミナル.app」が登録されていなければ「+」をクリックして追加し、設定を ON にする。なお、ターミナルは「アプリケーション > ユーティリティ」の下にある。

さらに、アプリケーションが無条件で全てのファイルにアクセスできるとは限らないため、その設定を行っておく。「プライバシーとセキュリティ」の項目の中に「フルディスクアクセス」があるので、これを開き、「ターミナル.app」と「Visual Studio Code.app」を ON にする。



フルディスクアクセスを設定

1.2 ターミナルの設定

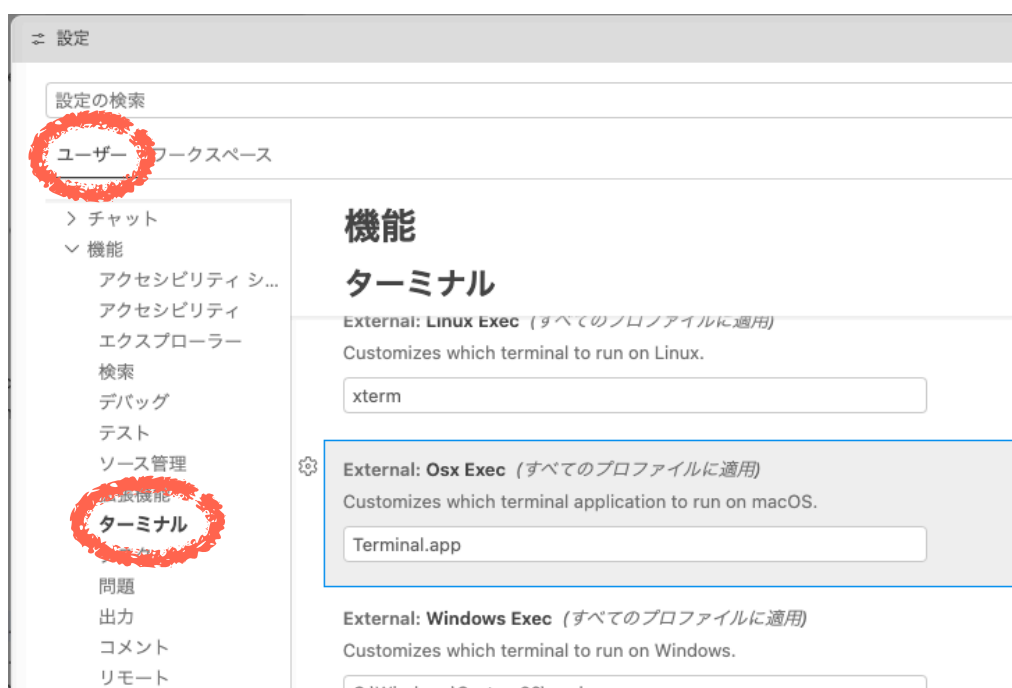
デバッガlldbと VS Codeを組み合わせで動作させた場合、デバッグ対象のプログラムとの入出力ができない、あるいは日本語の表示が化ける（16進数になる）といった問題が発生することがあることが知られている。このような現象に対応するため、以下ではデバッグ作業の際の入出力

をターミナル.appのウィンドウで行うための設定を行う。自分の VS Codeの環境で入出力に関する問題がない場合、ターミナルを使う必要はない。

ここではまず、ターミナル.app の場所が VS Codeに正しく設定されているかを確認しておく。

今回の設定ではデバッグ作業の際の入出力をターミナルのウィンドウで行う。そのため、ターミナル.app の場所が正しく設定されていることを確認しておく。

VS Code のメニューから「Code > 基本設定 > 設定 (⌘,)」を選択、またはウィンドウ左下の歯車アイコンから「設定」を選択すると図のような「設定」パネルを表示できる。左上の「ユーザー」というタブを選択し、左側の「機能」の項目から「ターミナル」を選択する。表示される設定項目の中に「External: Osx Exec」があるので、ここに「Terminal.app」と設定されていることを確認する（設定されていなければ記入する）。ここは「ターミナル.app」ではなく、英語の綴りで記入する（Finderに見えている名称は日本語UI向けのものであるため）。



ターミナルの設定を確認

なお、ターミナル自体が起動しないという現象に見舞われた場合、この指定をフルパス名にする必要があるかもしれない（/System/Applications/Utilities/Terminal.app）。特に、他のターミナルアプリ（iTermなど）を使ったことがある場合は注意が必要である。

1.3 拡張機能の確認

VS Codeのウィンドウの左側に拡張機能のアイコンがあるので、これをクリックして、インストールされている拡張機能を調べておく。现阶段では「C/C++」と「Japanese Language Pack」だけはインストールされていると仮定して説明を行う。

1.4 ディレクトリをオープン

多くの統合開発環境(IDE)では、あるソフトウェアを開発するためにまとめたソースファイルや関連するリソース（Makefileなどの開発管理用記述、UI設定ファイル、画像や音声データ、付属するテキストなど）の一式を**プロジェクト**と呼び、これらを使って実行可能なソフトウェアを作る作業（典型的にはコンパイル、リンク）を**ビルド**と呼ぶのが通例である。

VS Codeは一般のIDEと違い、プロジェクトの代わりに**ワークスペース**という用語を使う。ほとんどの場合（特に小規模な開発では）、ワークスペースはプロジェクトと同じと考えてよい。

VS Codeでソースファイルが存在するディレクトリを指定してオープンすると、そこがワークスペースのディレクトリとなる。起動時に表示されるウェルカムページの「開く...」を使うか、メニューから「開く...」（⌘O）、または「フォルダーを開く...」をクリックする（「このフォルダー内のファイルの作成者を信頼しますか？」というパネルが出るが、通常は問題ないはず）。

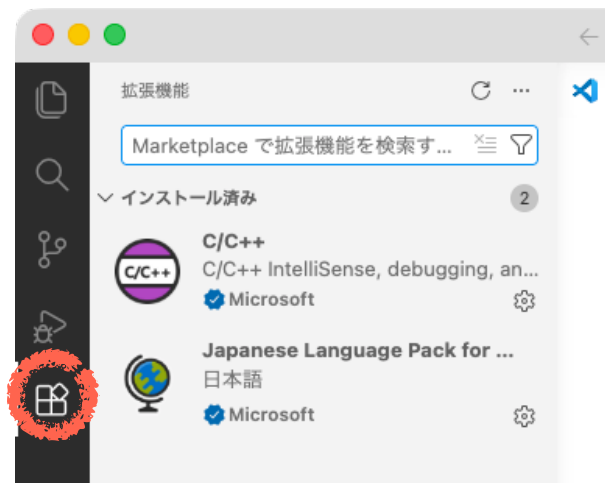
1.5 C言語の設定

左側の「エクスプローラー」（左上の書類の形のアイコンをクリック）で表示されているファイル名を選択すると、中央部分にコードが表示され、編集作業を行うことができる。

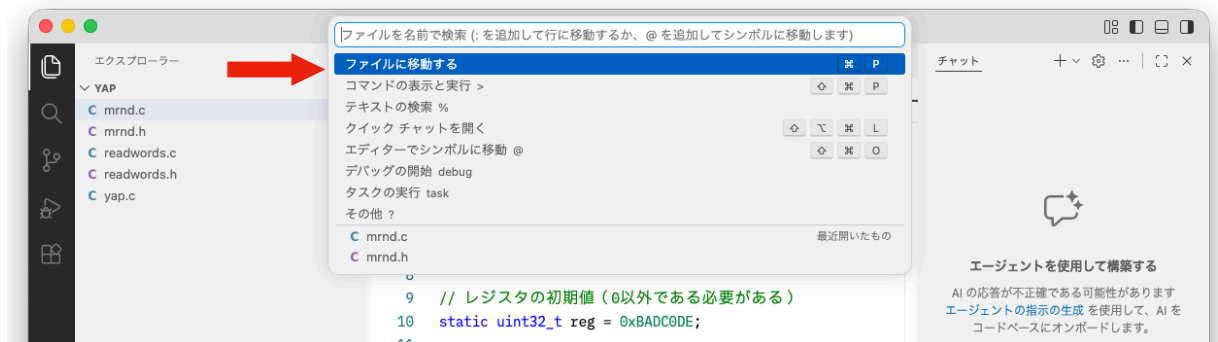
初期設定ではエディタやコンパイラなどの設定が C++ になっていることがあるため、設定を確認、変更する必要がある。ここではコマンドパレットから設定を確認する方法を説明する。

コマンドパレットを表示するには、上部にある文字入力フィールド（クイックアクセス）をクリックして「コマンドの表示と実行」を選択するか、文字 ">" をタイプする。あるいは、シフト+Command+P（⇧⌘P）をタイプしたり、ウィンドウ左下の歯車アイコンを使ってもよい。

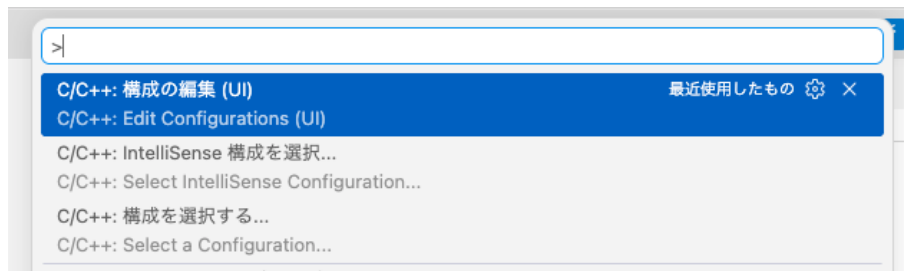
なお、クイックアクセスを表示したが何もせずに元に戻したい場合は ESCキーをタイプする。



拡張機能を確認

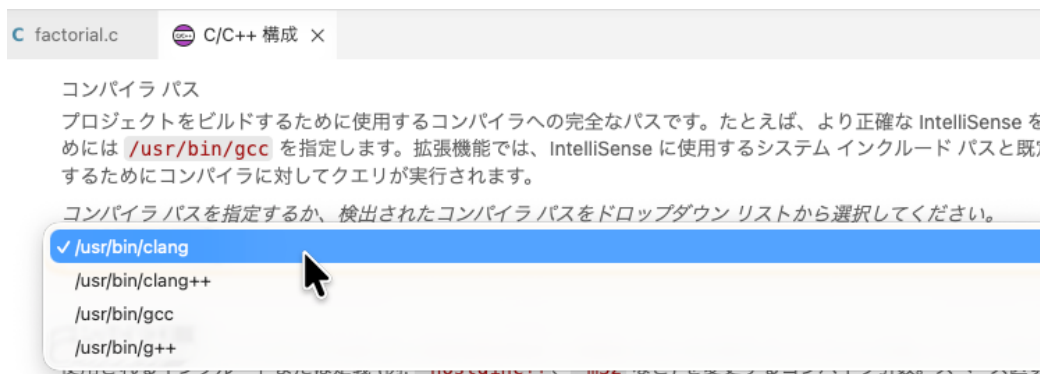


コマンドの表示と実行（コマンドパレットを表示 ⇧⌘P）



「C/C++: 構成の編集(UI)」をクリック

次に、「C/C++: 構成の編集(UI)」を探してクリックする。この中に「コンパイラパス」を指定する箇所があるので、C++ではなくCのコンパイラ、具体的には **clang** または **gcc** を選択する（通常はどちらでも同じである）。また、「C標準」は **c17** を選択すると良い。



コンパイラパスの設定

コマンドパレットには多くのコマンドが列挙されるので、目的のものをすぐには探すことができない場合がある。クイックアクセスにそのまま「>」を入力し始めてもコマンドパレットの機能に切り替わるが、さらにそのままコマンド名を入力すると、タイプした文字列にマッチするコマンドの一覧が表示されるのでそこから選ぶこともできる。マッチングは主に英語のコマンド名に対して行われるが、入力のコマンド名に正確に一致している必要はなく、一部分の文字列や先頭文字の羅列に近いと判定したコマンドを表示する。例えば「C/C++: 構成の編集(UI)」のコマンド名は「C/C++: Edit Configurations (UI)」なので「>editconf」や「>cui」のように入力して探し出すことができる。

拡張子が **.c** であるファイルを選択し、ウィンドウの右下を見ると、C++ となっていることがある。この場合、拡張子が **.c** のファイルを C++ のファイルと自動的に判断しているので、ここをクリックして設定を変更する必要がある（Cになっている場合は設定不要）。



この部分をクリックして、クイックアクセスに表示される「'.c'」に対するファイルの関連付けの構成」を選び、「C」を選択する。同様に拡張子が **.h** のファイルも確認し、C++ と見なされているのであれば、「'.h'」に対するファイルの関連付けの構成」を選び、「C」を選択する。

1.6 設定ファイル

VS Code は、挙動を指定する情報が書かれたファイルをいくつかの場所（ディレクトリ）に保存している。

(A) 現在のワークスペースの作業に適用されるカスタマイズ情報を記録する

上記 1.3 などの設定を行うと、ワークスペースのディレクトリ直下に `.vscode` というディレクトリが作成され、この中に各種の設定情報が書かれたファイルが作られる。なお、先頭が `"` であるファイル名やディレクトリ名は `mac` の `Finder` や `UNIX` の `ls` コマンドでは表示されない。`ls` コマンドで表示するためには `-a` オプションが必要である。

<code>settings.json</code>	文字の大きさ、インデント幅、カラーテーマなどの基本設定
<code>tasks.json</code>	コンパイル、テストなど、コマンドをショートカット化
<code>launch.json</code>	プログラムを実行、デバッグする際の設定
<code>c_cpp_properties.json</code>	そのプロジェクトに関する C/C++ の設定

設定ファイルの多くは **JSON形式** で記載されている（拡張子は `.json`）。設定はエディタのインタフェースから操作することも、テキストファイルとして編集することもできる。JSON形式については「5. 付録: JSONの記法」を参照すること。

(B) ユーザのワークスペースに共通するカスタマイズ情報を記録する

`macOS` の場合、`~/Library/Application Support/Code/User/` というディレクトリに、このパソコンで編集されるユーザのワークスペース全てに共通する設定（キーバインドなど）を保存するファイルが置かれる。以下、この文書では「ユーザ設定」と呼ぶ。代表的なファイルには以下のものがある。

<code>settings.json</code>	文字の大きさ、インデント幅、カラーテーマなどの基本設定
<code>keybindings.json</code>	キーの割り当てやショートカットの設定

`settings.json`、`tasks.json` などはユーザ設定として上記の共有のディレクトリに置くことも、ワークスペースに固有の定義として `.vscode` の中に置くこともできる。両方のディレクトリに記述があった場合、ワークスペース側の定義が優先される。従って、自分が比較的によく使う設定はユーザ設定としてまとめて記述し、特定のケースで変更したい設定だけをワークスペースの `.vscode` の中に記述するという使い方ができる。

また、ある開発で使ったワークスペースの設定ファイル（`.vscode` のディレクトリ以下）を別の新しいワークスペースにコピーすれば、多少の修正で同じような作業環境を利用できる。

(C) プロファイルを作って設定を切り替える

文字サイズや色使いなどに関し、いくつかの設定を作成しておいて切り替えることができる。この設定のセットのことを **プロファイル** と呼ぶ。プロファイルの情報は上記の共通のディレクトリに記録される。

プロファイルの作り方と利用方法は「2.1 プロファイルを作成」で説明する。

(D) VS Codeの拡張機能を管理するために作られる

ホームディレクトリ直下に `~/.vscode` というディレクトリが自動的に作成される。また、`~/.vscode-shared` などのディレクトリが作成されることもある。これらのディレクトリは VS Codeが管理しているため、消去してはならず、内容を自分で操作する必要もない。

1.7 設定の変更方法

(A) VS Codeのインタフェースから設定する

「設定」というパネルを表示して設定を確認、変更できる。



「設定」のパネル

VS Code のメニューから「Code > 基本設定 > 設定」を選択するか、ウィンドウ左下の歯車アイコンから「設定」を選択するか、またはコマンドキーとカンマ (⌘,) をタイプすると図のような「設定」パネルを表示できる。左上の「ユーザー」というタブを選択したするとユーザー設定、「ワークスペース」のタブを選択すると現在編集中的ワークスペースに関する設定を表示、変更できる。

例えば、「ワークスペース」に切り替え、左側で「テキストエディター > フォント」を選択して、Font Size を見てみる。デフォルトは12ポイントだが、これを違う値（例えば16など）に変更すると、エディタのフォントが大きくなる。

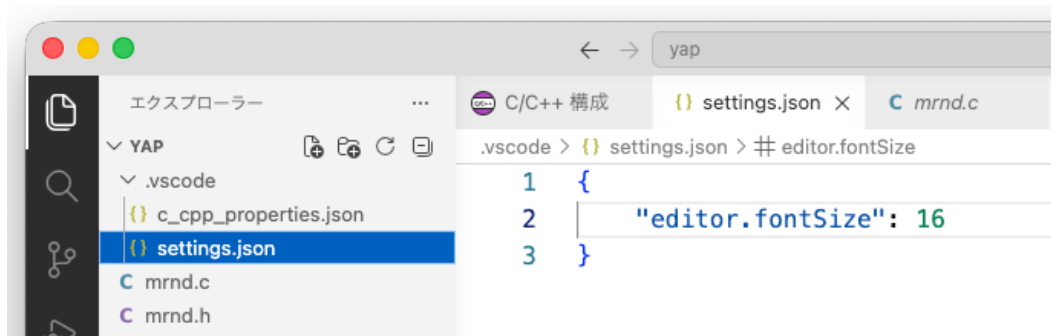
なお、「設定の検索」に設定名をタイプすると、コマンドパレットと同様にマッチするものが検索されて表示される。

(B) 設定ファイルを直接編集する

「ワークスペース」に関する設定はそのワークスペースのディレクトリにある `.vscode` というディレクトリ内の設定ファイルに反映される。このファイルを直接書き換えることによって、設定を変更できる。

例えば、上の(A)の説明のようにパネルを使ってワークスペースのフォントを16ポイントに設定したとすると、`.vscode/settings.json` は以下のようにになっている（他にも設定を変更し

ていればその内容も含まれる)。この16を12に変更してファイルを保存すると、文字の大きさは元に戻る。



.vscode/settings.json を編集する

利用する文字の大きさをどのソフトウェア開発でも同じにしたいなら、フォントサイズはワークスペースではなく、ユーザ設定で指定しておくのが良いだろう。そこで次に、「ユーザー」の設定について試してみる。まず、上で作成したファイル `settings.json` を削除するか、2行目の設定を削除しておく（ワークスペースの設定が優先してしまい、効果が確認できないため）。

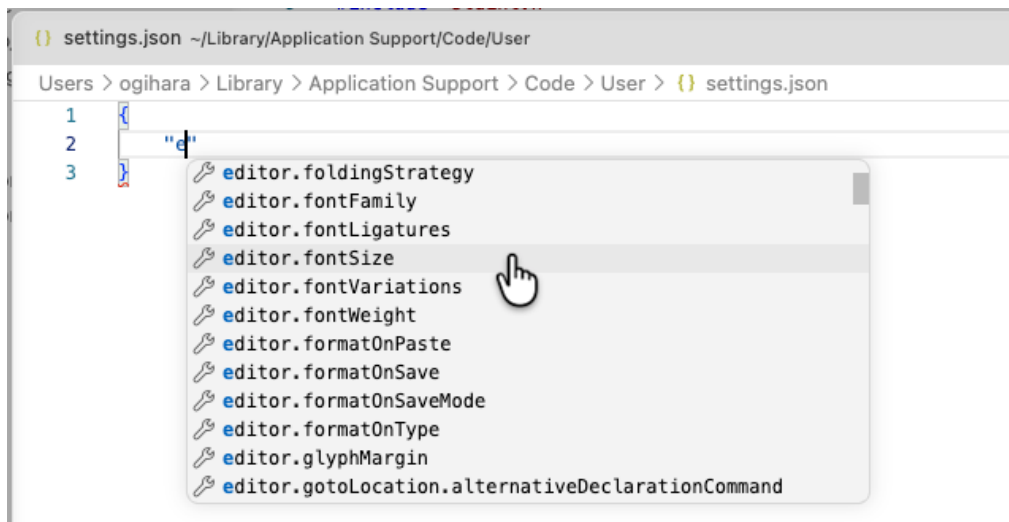
まず、上で説明したように「設定」パネルを表示（歯車アイコンや⌘,を使用）して、「ユーザー」タブに切り替える。このまま設定パネルからフォントサイズを変更することもできるが、ここではパネル右上にある「設定(JSON)を開く」というボタンをクリックする。



「設定」パネルから「設定(JSON)を開く」をクリックする

すると、`~/Library/Application Support/Code/User/settings.json` というテキストを編集するパネルが表示される。これまでに何かの設定をしていればその内容が反映されている（この例では何も設定していないと仮定）。

ここに、上の `editor.fontSize` の設定と同様な内容を書き込むことができるが、入力を始めると候補となる文字列が表示されるので、ここから選択することができる。JSONでフォントサイズを指定し（例えば10）、ファイルを保存すれば設定終了である。ただし、上でも述べたように、ワークスペースの `.vscode/settings.json` に同じ指定が存在していればそちらが有効となる。



ユーザ設定(JSON)の変更

1.8 makefileを編集する設定

makefileはタブ文字を入力しなければならないなどの点で一般のプログラミング言語と異なっている。ここでは、VS Codeで makefileを編集する際にのみ有効な設定を行う方法を示す。

どのワークスペースでもmakefileでの編集は共通しているため、ここでは「ユーザー」に対して設定を行う方法を示すが、各ワークスペースで settings.json に設定を追加したり上書きしたりすることもできる。

「設定」パネルを表示し、右上の「設定 (JSON) を開く」をクリックし、「ユーザー」の設定ファイルが編集できる状態にする。ここに、以下の記述を追加する。

新しい定義を追加する場合、前の行の最後に「,」を忘れずに追加すること。

makefileと同様に、拡張子 .c のファイルに特有の設定を行いたい場合には、「[c]」という文字列をキーとして同様な設定を行えばよい。

```
{
  "workbench.colorTheme": "Visual Studio Light", // 別の設定。「,」に注意！
  // ここから下を追加
  "[makefile]": { // makefile に対してのみ有効
    "editor.insertSpaces": false, // TABで空白を入力しない
    "editor.detectIndentation": false, // 自動インデントをしない
    "editor.renderWhitespace": "all" // TABや空白を特別に表示する
  },
}
```

なお、本来のJSON形式には上記のようなコメントは記入できないが、VS Codeの設定ファイルは特別に記入できるファイル形式を利用している（/* ... */ の形式も記述可能）。同様に、列挙された要素の最後に余計に「,」が付いていた場合も許容される（本来はエラー）。

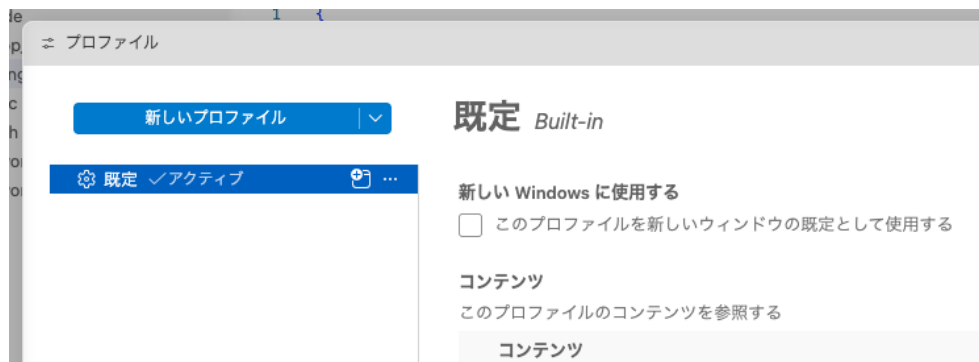
JSON形式については「5. 付録: JSONの記法」を参照すること。

2. ビルドの設定とmakefileとの連携

2.1 プロファイルを作成

状況に応じて各種設定の使い分けをしたい場合、それらの設定をプロファイルという単位にまとめておき、いつでも切り替えられるようにできる。プロファイルの作成は必須というわけではないが、今回は、既にインストールした拡張機能とCodeLLDBの混在によるトラブルを避けるという意味がある。

メニューから「Code > 基本設定 > プロファイル」を選択するか、ウィンドウ左下の歯車アイコンをクリックして「プロファイル」を選択すると図のようなパネルを表示できる。



プロファイルのパネル

ここでは、C言語のデバッグ環境に切り替えるためにプロファイル「Cデバッグ」を作る。

パネルの「新しいプロファイル」をクリックし、「名前」の欄に「Cデバッグ」と入力する。



プロファイル「Cデバッグ」を作成

「コピー元」は指定しない。

「コンテンツ」の箇所の設定は注意が必要である。「設定」と「キーボードショートカット」は「既定値」を指定して、上で設定した通常の設定を引き継いで使えるようにする。さらに、「拡張機能」はこれまでにインストールしたものを使わないようにしたいので、「なし」にする。

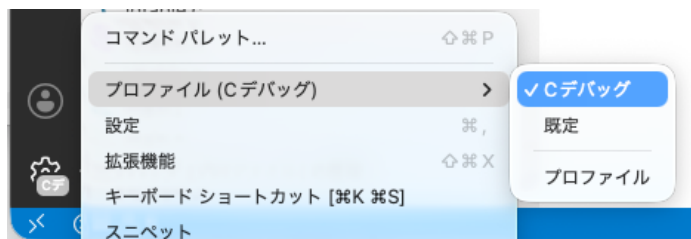
これで「作成」ボタンを押す。



コンテンツで「拡張機能」を「なし」に

2.2 拡張機能をインストールする

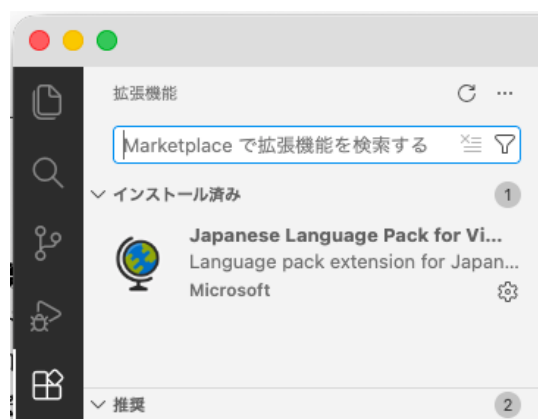
ウィンドウ左下の歯車アイコンから「プロファイル」を選択して「Cデバッグ」に切り替える。他の作業をする時は「既定」に戻せばよい。



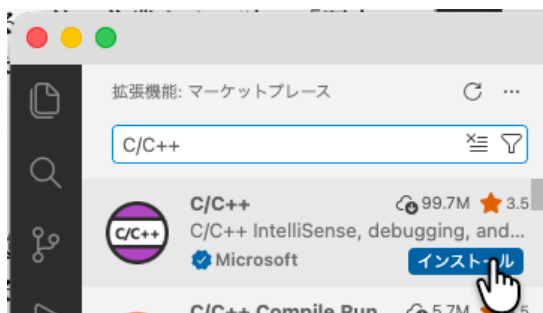
プロファイルを切り替える

この状態で拡張機能を確認すると、これまでにインストールしたものが見えなくなっていることが分かる。そこで、まず「C/C++」をインストールする。拡張機能の上部にある検索フィールドに名称をタイプすれば一致するものが表示される（C/C++ Extension Packではないので注意）。もし日本語言語パックも存在しなかったら追加しておく。

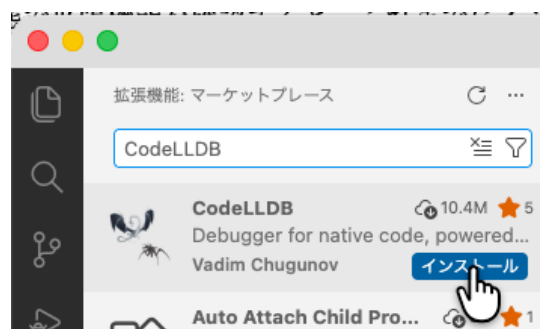
同様に Code LLDB も検索してインストールする。



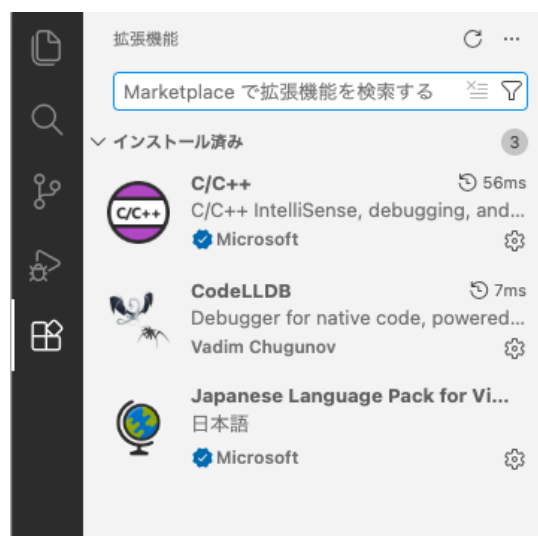
プロファイル作成直後の拡張機能一覧



“C/C++”をインストール



“CodeLLDB”をインストール

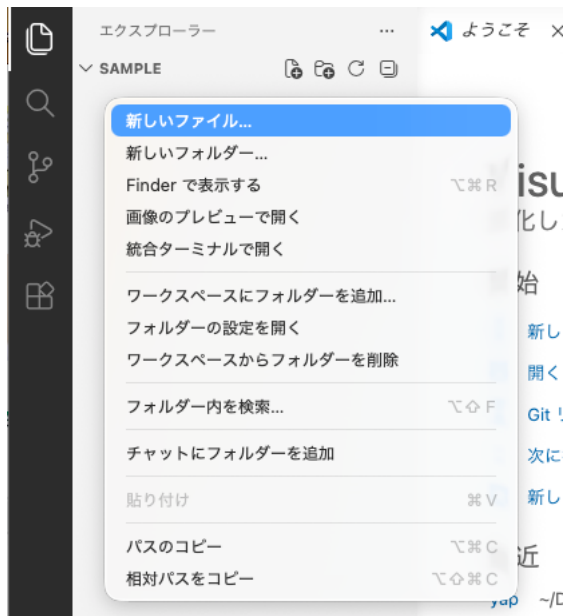


プロファイル「Cデバッグ」の拡張機能

2.3 ソースファイルが1つしかない（makefileを使わない）場合

VS Codeでは、ビルドやプログラムの実行などの操作を、ボタンのクリックや1回のキー入力（ショートカット）だけで行えるように設定できる。この単純化された操作を**タスク**と呼ぶ。以下の説明で「タスクを構成する」というのは、例えばビルド、プログラムの実行、デバッグの開始といった操作を、単純な操作で行えるタスクという仕事の単位として定義することを指す。

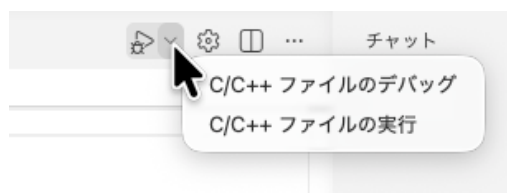
ここではまず、main.c というファイルしかない場合を考え、そのファイルをビルドするタスクを構成する方法を示す（ファイル名は任意で、そのファイルだけで実行可能であるとする）。



エクスプローラーでコンテキストメニューを開く

あらかじめディレクトリ内にソースファイルを用意しておいてもよいが、エクスプローラーを表示させてからコンテキストメニュー（アプリケーションのそれぞれの位置で状況に合わせて表示されるメニュー。macOSでは Control+クリック、またはトラックパッドを2本指でクリックして表示される）でファイルの作成、名称変更、削除などができる。

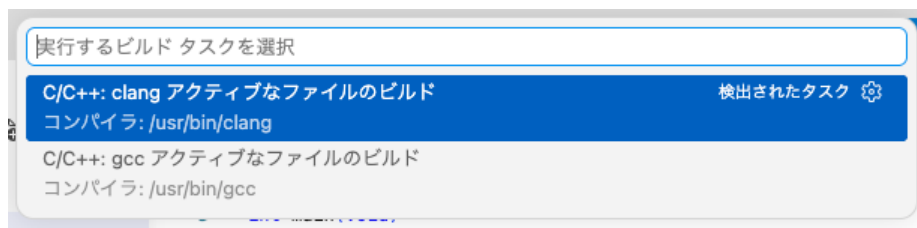
完成した main.c を選択し、ウィンドウの右上の「>」（虫の印が付いていることがある）から「ファイルの実行」を選択すると次の表示が出るので clang の項目を選択すればコンパイルと実行（デバッグ）が行える。



ソースファイルを選択して「ビルドとデバッグ」を行う

この時、ワークスペースのディレクトリに `.vscode/tasks.json` というファイルが作成される（これについては下で述べる）。

実行まではせず、ビルドだけを行うには、`main.c`を選択してシフト+Command+B（⇧⌘B）をタイプする。あるいはメニューから「ターミナル > ビルド タスクの実行...」を選択すると「実行するビルド タスクを選択」という次のような表示が出るので、「clang アクティブなファイルのビルド」をクリックすればビルド（コンパイル）だけが実行される。



ソースファイルを選択して ⇧⌘B をタイプ

ただし、ここまでの方法では常にデフォルトの動作しかできない。コンパイル時のオプションなどを指定するには、`.vscode/tasks.json` という設定ファイルを変更する。

上記の「ビルドとデバッグ」を行なってもこの設定ファイルは作成されるが、ビルドを行わずにファイルだけ作成することもできる。このためには `main.c` を選択しておいて、メニューの「ターミナル > タスクの構成...」を選択する。すると今度は「構成するタスクを選択」というメッセージが表示されるので、「C/C++: clang ビルド アクティブ ファイル」を選択する。あるいは、先ほどの「実行するビルド タスクを選択」の表示が出た際に、右側の歯車アイコンをクリックしても同じことができる。



メニューの「ターミナル > タスクの構成...」を選択

これによって、ワークスペースのディレクトリに `.vscode/tasks.json` というファイルが作成される。このファイルの内容はデフォルト時のビルドの動作を `tasks.json` に書き出したもので、この記述を変更することでビルド時の動作を変更できる。

定義では、`tasks` というキーが付いた配列に1つのタスクが定義されていることを表している。タスクが実行するコマンドは `clang` で、その引数が `args` に列挙されていることが分かる。なお、`clang` のオプション `-fcolor-diagnostics` と `-fansi-escape-codes` はターミナルでエラーや警告を色付きで表示させるための指示である。もし、色を付けたくないという場合にはこのオプションの代わりに `-fno-color-diagnostics` とする。

```
{
  "tasks": [
    {
      "type": "cppbuild",
      "label": "C/C++: clang アクティブなファイルのビルド",
      "command": "/usr/bin/clang",
      "args": [
        "-fcolor-diagnostics",
        "-fansi-escape-codes",
        "-g",
        "${file}",
        "-o",
        "${fileDirname}/${fileBasenameNoExtension}"
      ],
      "options": {
        "cwd": "${fileDirname}"
      },
      "problemMatcher": [
        "$gcc"
      ],
      "group": "build",
      "detail": "デバッガーによって生成されたタスク。"
    }
  ],
  "version": "2.0.0"
}
```

.vscode/tasks.json の内容

ビルドを行う前に、clangか gccかという選択が毎回表示されるが、これを出さないようにする（選択肢のデフォルトであることを指定する）には、上記の設定の "group" の箇所を次のように変更すればよい。

```
"group": {
  "kind": "build",
  "isDefault": true
},
```

2.4 ソースファイルをまとめてコンパイルする（makefileを使わない）場合

makefileなどは使わず（依存関係などに関係なく）、とにかく *.c というファイルだけをまとめてコンパイルさせるという指定もできる。このためには上記の .vscode/tasks.json の "args" の箇所で複数ファイルを指定すればよい。またこの例では、生成される実行可能形式のファイル名は "doit" に固定している。

上(2.1)で述べた、単一のファイルを選択してからコンパイルする tasks.json の記述には、\${fileDirname} あるいは \${fileBasenameNoExtension} という記法を使用したか、これはファイルが何か選択されていない場合には意味がないので注意が必要である。

```
{
  "tasks": [
    {
      // 中略

      "args": [
        "-fcolor-diagnostics",
        "-fansi-escape-codes",
        "-g",
        "${workspaceFolder}/*.c", // ← ここに注目
        "-o",
        "doit" // 実行可能ファイルの名前は固定
      ],
      "options": {
        "cwd": "${workspaceFolder}" // ← ここも変更
      },

      // 後略
    }
  ]
}
```

複数のファイルから実行形式ファイルを作成するための tasks.json

2.5 makefileを使ってビルドする方法

ターミナルから、あるいはVS Codeのウィンドウでターミナル領域を開いてmakeコマンドを直接入力してもビルドはできるが、VS Codeから makefile を使ってビルドできるようにも設定できる。その設定は .vscode/tasks.json に記述する。

makefileはあらかじめ作成しておき、エクスプローラーから .vscodeの下に tasks.json を以下のような内容で作成すればよい。色を付けた箇所は開発対象に応じて書き換える部分である。

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "shell",
      "label": "make build", // デフォルトとは違う名前にするとよい
      "command": "make",
      "args": [], // 特定のターゲット（例: ["all"]）やオプションを記述できる
      "group": {
        "kind": "build",
        "isDefault": true // デフォルトのビルド方法にする
      },
      "problemMatcher": ["$gcc"] // エラー位置をVS Codeに教える設定
    }
  ]
}
```

makeを起動するための tasks.json

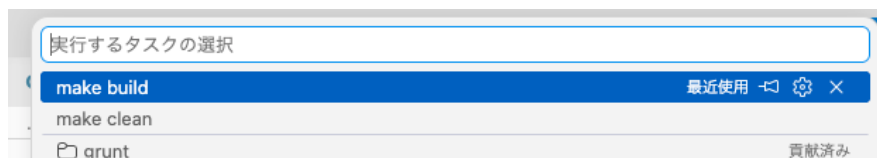
シフト+Command+B (⌘B) をタイプ、あるいはメニューから「ターミナル > ビルド タスクの実行...」を選択するとビルドできる。

設定ファイルのキー "tasks" の値は配列だが、ここには複数のタスクを列挙することもできる。例えば、makeのターゲットに clean が指定できるとすると、以下のように記述できるだろう。

```
{
  "version": "2.0.0",
  "tasks": [
    {
      // 上記の tasks の内容がここにあるとする
    },
    {
      "type": "shell",
      "label": "make clean", // 何を行うかわかる名前にする
      "command": "make",
      "args": ["clean"], // makeのターゲット
      "group": "build", // デフォルトにはしない（選択肢として出す）
      "problemMatcher": ["$gcc"]
    }
  ]
}
```

複数のタスクを tasks.json に記述する例

メニューから「ターミナル > タスクの実行...」を選択すると、以下のような表示が出されるので、目的とするものを選択すればよい（表示の順序は、前回選択したもの、よく選択するものが上に来るようにになっている）。



「ターミナル > タスクの実行...」を選択

タスクの中でよく使うものはピンの形のアイコンをクリックして「ピン留め」することができる。また、使わない項目は「×」をクリックして表示しないようにもできる。

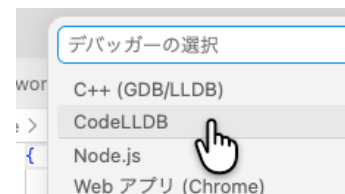
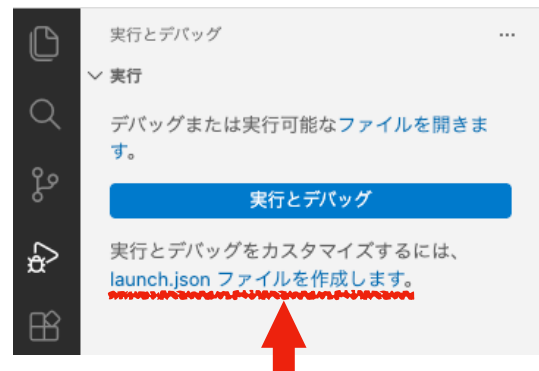
なお、makefileを使ってプロジェクトのビルドを行うために、Makefile Tools のような拡張機能をインストールして使うこともできるが、本文書では扱わない。

3. デバッグ

3.1 launch.jsonを用意する

VS Code でデバッグ作業を行うには、`.vscode/launch.json` というファイルでデバッグの起動方法を設定しておく必要がある。自分で `.vscode` の中にファイルを作るか、あるいはウィンドウ左側の▷と虫を組み合わせたアイコンをクリックして「`launch.json` ファイルを作成します」の箇所をクリックする。

あるいは「実行とデバッグ」に「構成がありません」と表示される場合は「構成の追加...」を選択し、「デバッガーの選択」で CodeLLDB を選択する。



デフォルトで作成される `launch.json` に、さらに推奨する設定を追加したものを下に示す。

ここでは VS Codeのウィンドウ上のターミナルを使う設定の他、に Macのターミナル.appをオープンする設定も選択できるように表示している。文字化けなどの問題がなければ VS Codeのターミナルを使えばよいし、別のウィンドウに表示された方が使いやすいならターミナル.appに変更してもよい。

設定の `preLaunchTask` には `tasks.json` のビルド動作のラベル名を記述でき、デバッグ開始前にもう一度ビルドを行うように指定できる。必要なければ記述しなくてよいが、記述する場合には `launch.json` に記述した `label`名と一致させなければならない。

`configurations` の値も配列であり、設定の組を複数個列挙することができる。その場合、`name` に指定した文字列でどのデバッグ作業を開始するかを選択できる。

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "(lldb) 起動",
      "type": "lldb",
      "request": "launch",
      "program": "${workspaceFolder}/main", // 実行形式のファイル名を指定
      "args": [], // 実行時にコマンドライン引数があるなら、ここに列挙する
      "cwd": "${workspaceFolder}",
      "env": { }, // 環境変数を指定する場合はここに記述
      "console": "integratedTerminal", // VS Codeのターミナルを使う
      // "console": "externalTerminal", // ターミナル.app を使う
      "preLaunchTask": "make build", // デバッグ前にtasks.jsonの動作を実行
    }
  ]
}
```

標準的な`launch.json`

なお、デバッグ時にリダイレクションの指定もできる。キー "stdio" に対して文字列の配列を値として指定し、リダイレクション先として指定したいファイルパスを標準入力、標準出力、標準エラー出力の3つ分列挙すればよい。設定を変更する必要がなければ null を指定しておく。

例えば、ワークスペースにあるファイル input.dat を入力としたい場合は次の指定を書き加えればよい。

```
"stdio": [  
    "${workspaceFolder}/input.dat", null, null  
],
```

標準入力へのリダイレクションの指定

3.2 デバッグのためのコンパイルオプション

VS Codeではデバッガとして macの lldb またはgdbを利用するが、どちらの場合でも実行ファイルを作成するための一連のコンパイルで、オプション -g を付けなければならない。これによって、デバッグに必要な情報を生成できる。

オプション -g を付けてコンパイルすると、ワークスペースに .dSYM という拡張子のディレクトリが作成される。この内部のファイルはデバッグに必要な情報を格納しているため、デバッグ作業中は操作してはならない。ただし、デバッグが終われば消去しても問題ない。

また、-O0 というオプションを付けることも推奨される（大文字の「オー」と数字のゼロ）。このオプションは生成される機械語の最適化(optimization)の度合いを指定するものであり、通常は -O3 または -O4 などでも最も高速に実行できる機械語が生成される。ただし、最適化されていると、必ずしもソースファイルに記述された順序通りに実行が行われず、デバッグ作業を行う上では障害になることがある（例えば、機械語のレベルで実行順を入れ替えたり、同じ式の評価を一箇所にまとめてレジスタで保持したりなど、ソースコードからは分からない動作を行うことがある）。-O0 というオプションは最適化を行わない指示のため、基本的にソースファイルに記述された通りに実行が進む。

なお、VS Codeでデバッグを行う際は同時にビルド方法の設定もしておくのが望ましいが必須ではない。makefileを使う必要がないのであれば、単にソースファイルを指定してコンパイルできればよく、必須なのはオプション -g を付けてコンパイルしておくことである。

3.4 例題

以下では例として、下に示すプログラム factorial.c を用いる。このプログラムは標準入力から整数値nを入力し、n! を計算して示す。

一方、この例では tasks.json と launch.json は下に示したものを利用し、makefile は使用しないものとする。この2つの設定ファイルは、エクスプローラーで選択したファイルをコンパイルし、デバッガを起動するために利用できる。ビルド対象のソースファイルは1つに限定されるが、そのようなシンプルなプロジェクトであればどこにでも ./vscode ごとコピーし、変更なく利用することが可能である。

makeが起動するように設定するには、「2.3 makefileを使ってビルドする方法」で説明したように tasks.json を記述し、必要に応じてlaunch.jsonを変更すればよい。

```

1  #include <stdio.h>
2
3  static int f(int n)
4  {
5      if (n > 1) {
6          return n * f(n - 1);
7      }
8      return 1;
9  }
10
11 int main(void)
12 {
13     int m = 0;
14     printf("入力 ? ");
15     scanf("%d", &m);
16     printf("結果: f(%d) = %d\n", m, f(m));
17     return 0;
18 }

```

factorial.c

```

{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "cppbuild",
      "label": "build myprog",
      "command": "/usr/bin/clang",
      "args": [
        "-fcolor-diagnostics",
        "-fansi-escape-codes",
        "-g", "-O0",
        "${file}",
        "-o", "${fileDirname}/myprog"
      ],
      "options": {
        "cwd": "${fileDirname}"
      },
      "problemMatcher": [ "$gcc" ],
      "group": "build",
      "detail": "コンパイラ: /usr/bin/clang"
    }
  ]
}

```

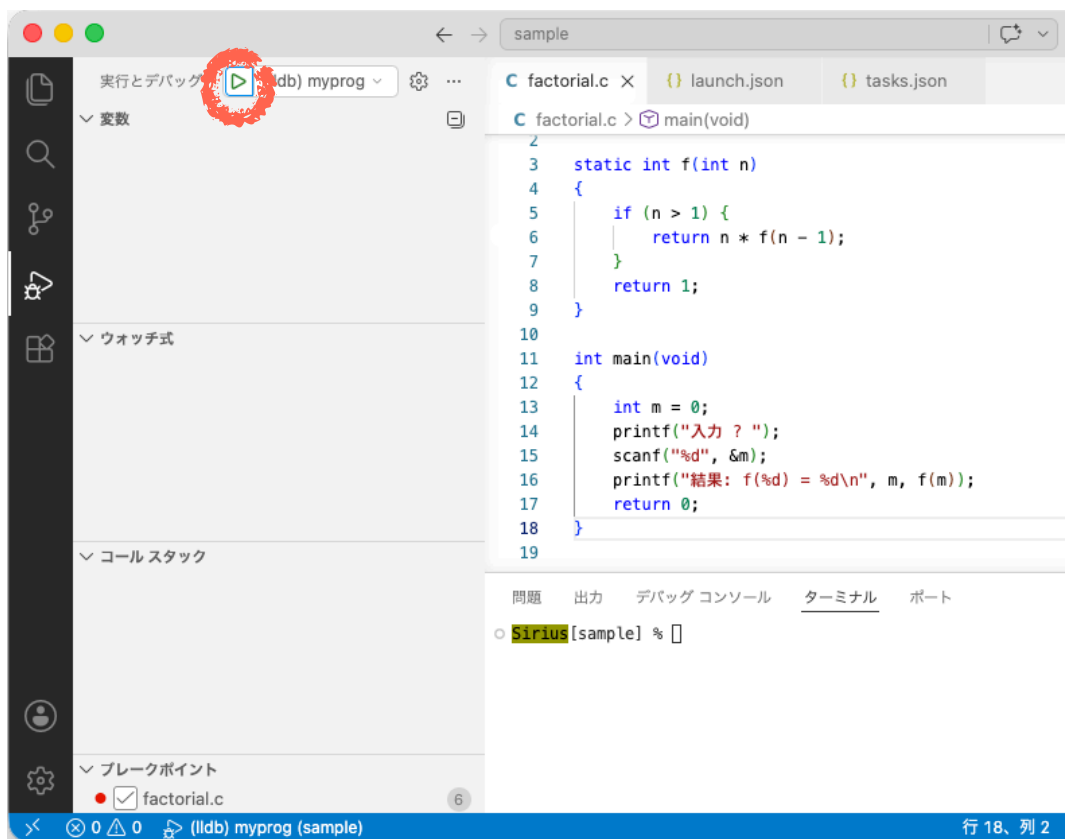
tasks.json

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "(lldb) myprog",
      "type": "lldb",
      "request": "launch",
      "program": "${workspaceFolder}/myprog",
      "cwd": "${workspaceFolder}",
      "console": "integratedTerminal", // または "externalTerminal",
      "preLaunchTask": "build myprog", // デバッグ前にビルドを実行
    }
  ]
}
```

launch.json

3.5 実行とデバッグ

デバッガを起動して、プログラムの実行だけを行う方法を説明する（正常に終了すればデバッグ作業には入らないことになる）。



「実行とデバッグ」の表示

ウィンドウ左側の「▷」に虫が付いているアイコンをクリックすると、ウィンドウの左側に「実行とデバッグ」という領域が表示される。ウィンドウ左上に「(lldb) myprog」(launch.jsonに記述した名前であることを注意)と表示されているはずなので、その「▷」をクリックすると、ビルドが開始される。

```
問題 出力 ターミナル デバッグ コンソール ポート
Console is in 'commands' mode, prefix expressions with '?'.
Launching: /Users/ogihara/Doc/京産大/プロジェクト演習/pj2026/Doc/work/sample/myprog
Launched process 46537 from '/Users/ogihara/Doc/京産大/プロジェクト演習/pj2026/Doc/work/sample/myprog'
Process exited with code 0.
```

デバッグコンソールの表示

ビルドが問題なく終わると VS Codeのターミナル（またはターミナル.appのウィンドウが1つオープンされる）で、ビルド対象のプログラムが動作する。このプログラムは入力を要求するので、ここでは5を入力して結果を表示させている。

```
問題 出力 ターミナル デバッグ コンソール ポート
入力 ? 5
結果: f(5) = 120
○ Sirius[sample] %
```

ターミナル領域でビルドが行われる

3.7 ブレークポイントの指定

ソースプログラムのある箇所をあらかじめ指定しておき、プログラムの実行がそこに及んだら中断させることができる。中断中は変数の値を調べたり、実行を継続させたり、あるいは1行ずつ実行してプログラムの振る舞いを調べることができる。この停止位置を**ブレークポイント**と呼ぶ。ブレークポイントは何箇所も指定できる。

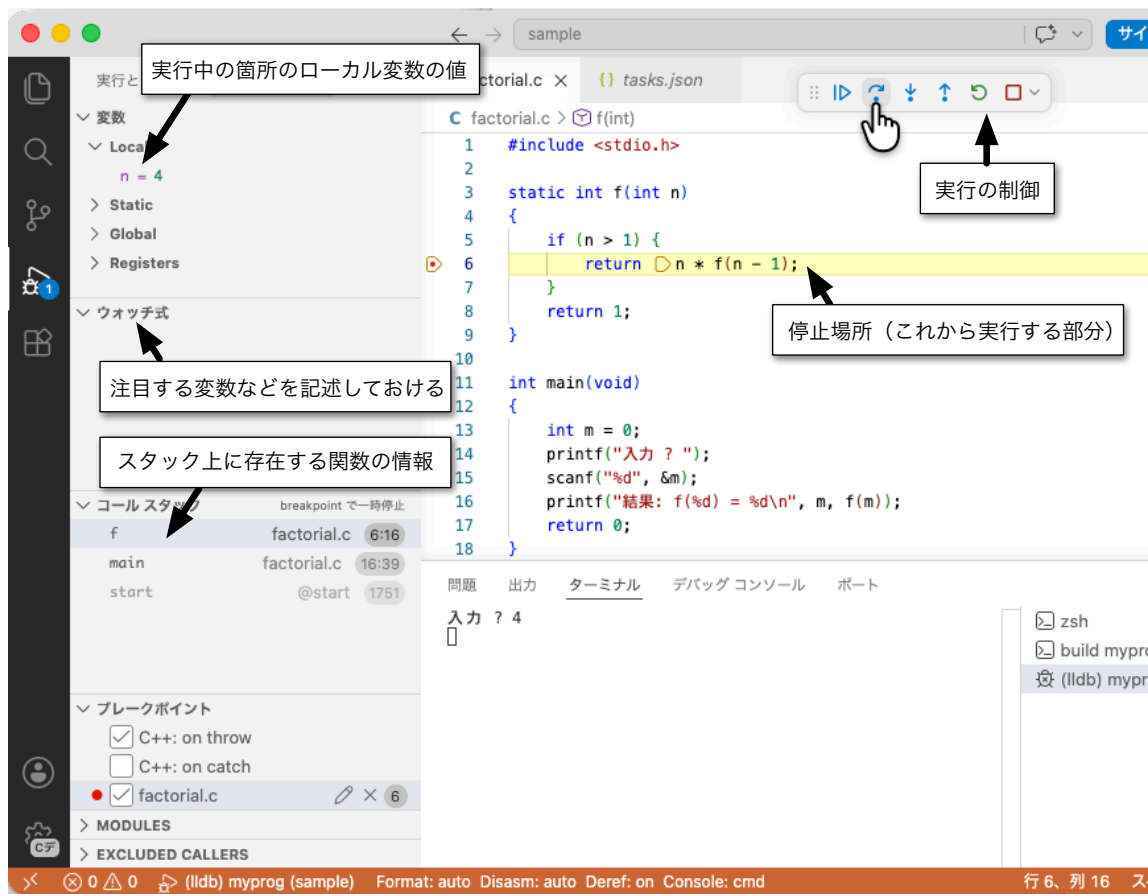
VS Code では、エディタの行番号の左側にマウスを持っていくと薄い赤丸が現れ、クリックすると濃い赤い丸になる。これでソースファイルのその行をブレークポイントとして設定できる（ただし、プログラムとして実行される箇所でなければ意味がない）。もう一度クリックするとブレークポイントを解除できる。

右図では、再帰呼び出しされるたびに引数nの値がどう変化するか調べるために6行目にブレークポイントを設定している。

```
C factorial.c > f(int)
1  #include <stdio.h>
2
3  static int f(int n)
4  {
5      if (n > 1) {
6          f(n-1);
7      }
8      return 1;
9  }
10
11 int main(void)
12 {
```

ブレークポイントの設定

ブレークポイントを設定してからもう一度プログラムを実行させると、ターミナルから値を読み込んだ後でプログラムの動作は停止し、VS Code のウィンドウは次のようになる。



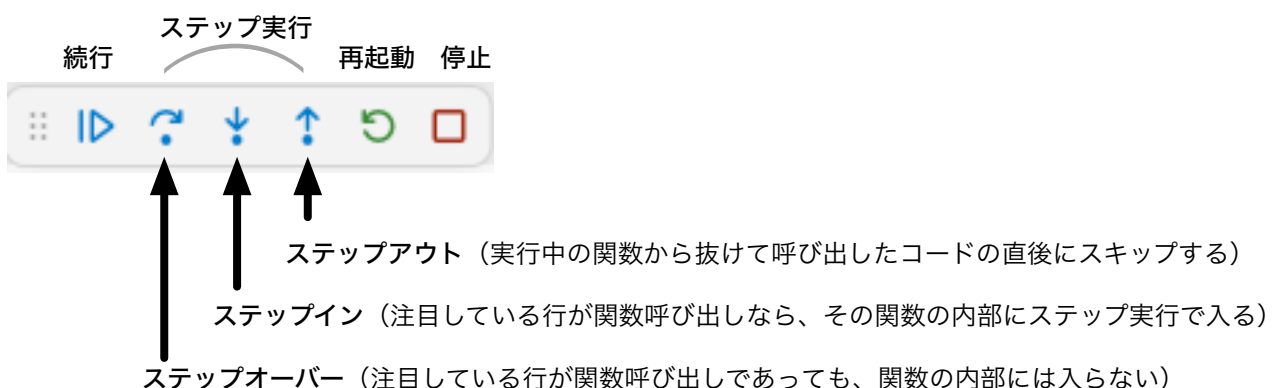
実行停止中のウィンドウの表示

図は、ターミナルから 4 を入力した直後にブレークポイントで停止した状況を示している。

まず、ソースコード上で黄色い色が付いている部分が現在プログラムが停止している箇所で、その行を実行する直前であることを示している。この時、仮引数の `n` は値が 4 なので、左側の変数ビューの「Locals」には `n = 4` と表示されている。

3.8 実行の制御

ウィンドウの上部のあたりに、下の図のような小さなパネルが表示される（これは自由に場所を移動できる）。



このボタンで、現在デバッグ中のプログラムの実行を制御できる。まず、「続行」はブレークポイントで停止しているプログラムの実行を再開させる。再開後、またブレークポイントに実行が及べば実行は中断される。

「停止」はプログラムの実行そのものを止める。「再起動」は同じプログラムをもう一度最初から起動させる。

矢印のボタンが3つあるが、これを使ってソースコードの1行ずつプログラムを実行させることができる。これを**ステップ実行**と呼ぶ。

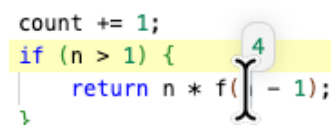
ステップ実行には、見た目の1行ずつ実行するステップオーバーという動作がある。ステップオーバーでは、注目している行（これから実行される行）が関数呼び出しであっても、それを1ステップとして実行する。関数の中をステップ実行で追うことはしない。一方、ステップインは関数呼び出しの場合にその関数の内部にステップ実行で入って行く。ステップアウトはその逆で、関数の内部で実行を停止している時、その関数を抜け出して呼び出し側に戻る。

ブレークポイントはデバッグ中でも設定・解除ができるため、この3種類のステップ実行と、ブレークポイント、続行を使い分けることによって、プログラムの必要な箇所の動作だけを詳しく調べることができる。

3.9 変数、式の値の参照

ウィンドウ左側の変数ビューには主にローカル変数の値が表示されるが、グローバル変数の値などを調べたい場合、ソースコードのその変数の上にマウスポインタを持って行くと小さなポップアップでその時の値が表示される。これをホバー表示と呼ぶ。

```
count += 1;
if (n > 1) {
    return n * f(n - 1);
}
```



ホバー表示の例

また、ウィンドウ左側の「ウォッチ式」には変数だけでなく、式を登録しておき、プログラムの実行に伴う式の値の変化を知ることができる。デバッグで注目している値や関係を式として列挙しておいて観察することができ、大変役立つだろう。

デバッグコンソールの一番下の行にも、式を記述してその時の値を表示させることができる。なお、C言語では代入も演算子であるため、ここに変数への代入を記入でき、実行中の変数の値を書き換えることが可能である。

3.10 コールスタック

実行中に呼び出された関数は、スタック上にローカル変数などの情報を保持するが、関数から関数を呼び出すことを繰り返すと、その情報はスタックに次々に積まれて行くことになる。

今、6行目のブレークポイントを解除し、ソースコードの8行目（return）に別のブレークポイントを設定して「続行」のボタンをクリックすると、コールスタックの表示は図のようになる（関数f()の再帰呼び出しで8行目が実行されるのは引数が1になった時だけ）。

コールスタック		breakpoint で一時停止
f	factorial.c	8:5
f	factorial.c	6:20
f	factorial.c	6:20
f	factorial.c	6:20
main	factorial.c	16:39
start	@start	1751

コールスタックの表示例

表示の内容は、プログラム名と関数名、呼び出された場所（ソースファイルの何行目、何文字目か）である。

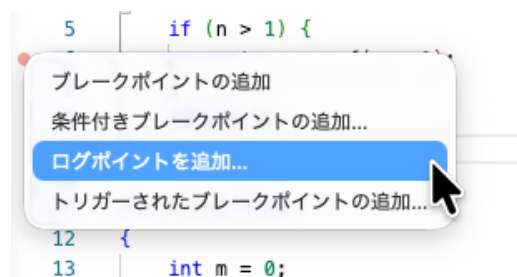
ここで興味深いのは、コールスタックの途中の行をクリックして選択すると、その呼び出しが起きていた時の状況（ローカル変数の値）が有効になるということである。変数ビューの値だけではなく、ウォッチ式の表示もこのスタックの選択によって変化する。実際に、コールスタックのいずれかの行をクリックした時、「変数」に表示されている `n` の値がどうなるかを観察するとよいだろう。

3.11 ログポイントと条件付きブレークポイント

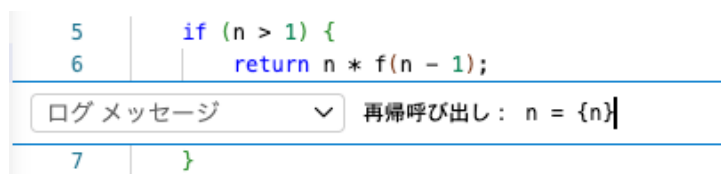
エディタの行番号の左部分をクリックするとブレークポイントを設定、解除できるが、ここで右クリック（Control-クリック、または指2本でトラックパッドをクリック）すると図のようなメニューが表示され、他の設定を行うことができる。

ログポイントとは、その箇所に実行が及んだ場合に、登録されているメッセージや式の値をデバッグコンソールに表示できるものである。

メッセージには通常の文字列を登録できるが、`{ }` で囲んだ式を含めると、実行がそこに及んだ時の値がその位置に埋め込まれる。



ログポイントなどの設定



ログポイントの設定例

条件付きブレークポイントとは、その箇所に実行が及んだ場合、登録されている条件式が真になった場合に実行が中断されるものである。例えば、何度も繰り返されるループの中にブレークポイントを設定したいという場合、調べたい状況を条件として指定すれば効率的にデバッグを進めることができる。

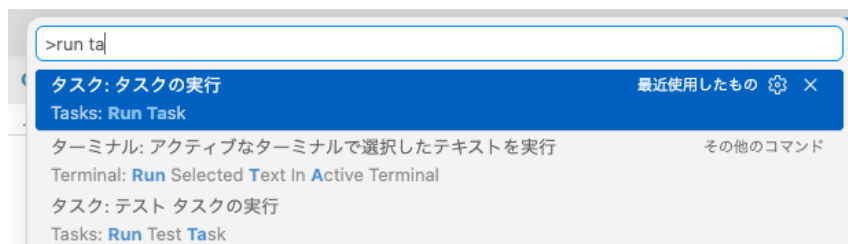
トリガーされたブレークポイントとは、他のブレークポイントを指定して、そのブレークポイントに実行が及んだ後に有効になるブレークポイントのことである。何かが連鎖的に起きる可能性がある場合に設定することが考えられるが、通常はあまり使われない機能のようである。

4. VS Codeの設定に関する追加事項

4.1 キーボードショートカットを設定する

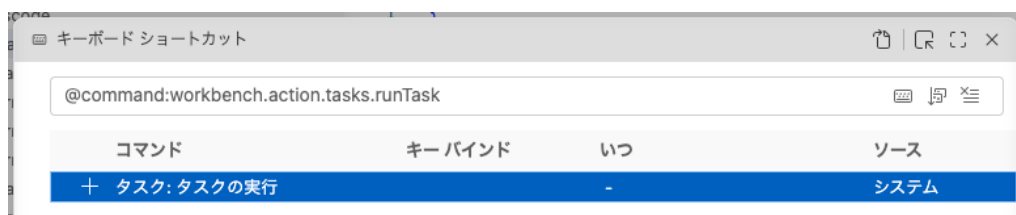
よく使う機能を特定のキーの組み合わせで呼び出すキーボードショートカットの設定について説明する。設定方法は色々あるが、ここでは「タスクの実行...」に、コマンドパレットからショートカットを設定する例を説明する。

コマンドパレットはウィンドウ上部の「クイックアクセス」を使ってシフト+Command+P (⌘P) でも表示できるが、クイックアクセスにそのまま「>」を入力し始めてもコマンドパレットの機能に切り替わる。VS Codeは「タスクの実行...」には「Run Tasks」という機能名を付けているので、「>」に続けて機能名を入力していくと綴りがマッチするものが選ばれて表示されてくる（もちろん、単純にコマンド一覧の中から探し出してもよい）。



コマンドパレットから目的のコマンドを探す

ここで右側の歯車アイコンをクリックすると「キーボードショートカット」というパネルが表示されるので、「+」をクリックするか行をダブルクリック、あるいは改行をタイプする。



キーボードショートカットの設定

表示されたパネルに、自分が使いやすいと思うキーの組み合わせを入力する。すでに使われている場合は下図のようなメッセージが出る。入力を取り消すにはESCキーを入力すればよい。（例えばこの場合、Control+Command+T (⌘T) などが使われていないと思われる）



キーの組み合わせを入力

設定は「ユーザー設定」として `keybindings.json` に書き込まれる。ファイルを直接編集するには、例えば「キーボードショートカット」のパネルの右上にある「キーボードショートカットを開く(JSON)」のアイコンをクリックすればよい。

4.2 変数展開

VS Codeは各種の設定の文字列に含めて使える既存の「変数」をいくつか用意している。それらは `tasks.json` や `launch.json` の文字列の中に `"${workspaceFolder}/a.out"` のように書いて展開できる。主なものを以下に示す。

変数	意味
<code>\${workspaceFolder}</code>	ワークスペースのルート
<code>\${workspaceFolderBasename}</code>	ワークスペース名
<code>\${file}</code>	現在のファイルのフルパス
<code>\${fileDirname}</code>	現在ファイルのディレクトリ
<code>\${fileBasename}</code>	ファイル名
<code>\${fileBasenameNoExtension}</code>	拡張子なしファイル名
<code>\${fileExtname}</code>	拡張子
<code>\${cwd}</code>	現在の作業ディレクトリ
<code>\${env:HOME}</code>	環境変数を参照
<code>\${config:editor.fontSize}</code>	設定値を参照
<code>\${command:...}</code>	コマンド実行結果
<code>\${input:...}</code>	ユーザー入力

VS Code の主な変数展開 (Variables Reference)

ただし、様々な設定の中で自由に展開できるわけではなく、利用可能な範囲は極めて限定されている。シェルスクリプトや `makefile` のように比較的自由に「変数」の値の設定や展開ができるわけではない。という点に注意が必要である。

まず、`settings.json` の設定の中では展開できない。また、`tasks.json` や `launch.json` の文字列の中で展開できる場合とできない場合がある。例えばタスクの名前を決める `"label"` の文字列に含めても展開されないが、一方、`"args"` の引数として指定すると展開できる。これらは試してみないと分からないことが多い。

4.3 独自の設定項目の利用

`settings.json` などに記述される `"editor.fontSize"` のような文字列をVS Codeでは設定項目、あるいはプロパティと呼ぶ。設定項目はJSONとして見るとオブジェクトのキーとして使われる文字列である。

文字列中のピリオドは単なる文字だが、擬似的に設定項目をグループ化する目的で使われている。例えば、`"editor.fontSize"` と `"editor.fontFamily"` はどちらも `editor` という擬似的なグループのサブ項目であるように扱われる。

VS Codeがあらかじめ知っている設定項目（例えば上で示した `"editor.fontSize"` の記述例のように）を `settings.json` に記述しておく、それらは `tasks.json` や `launch.json` の記述に先んじて読み込まれた値のように利用できる。

一方、VS Codeには存在していない独自の設定項目を `settings.json` に記述することもできる。例えば、以下のように記述できる。

```
{
  "myMake.debug": "debug",
  "myMake.target": "release",
  "myMake.clean": "cleanAll",
}
```

これらはそのままでは使い道がないが、"`${config:myMake.target}`" のように `config:` という文字列を前に付けて、他の定義の文字列の中で展開できる。ただし、上の変数展開でも述べたように、利用できる箇所には制限がある。

また、変数展開は文字列の展開にしか対応していない。例えば、複数箇所で指定するフォントの大きさを 11ポイントに統一したい、と考えて "`niceFontSize`": 11 のように `settings.json` に記述することはできるが、この値を他の設定ファイル内で数値として参照することはできない。

5. 付録: JSONの記法

以下は、Wikipedia日本語版に記載のJSON（ジェイソン、JavaScript Object Notation）の項目から記法に関する部分を抜粋、編集したものである(<https://ja.wikipedia.org/wiki/JSON>)。追加的な注釈も含む。JSONの記法はシンプルであり、他にも解説文書は多数存在する。

4.1 JSONのデータ型

JSONで表現するデータ型は以下の通りで、これらを組み合わせてデータを記述する。true, false, null などは全て小文字でなくてはならない。文字コードはUTF-8でエンコードすることが必須となっている。

- ・ **オブジェクト**（順序づけされていないキーと値のペアの集まり）
- ・ **配列**（データのシーケンス）
- ・ **数値**（整数、浮動小数点数）
- ・ **文字列**
- ・ **真偽値**（true と false。引用符を付けないことに注意）
- ・ **null**（値が存在しないことを表す）

数値は10進法表記に限り、8進、16進法表記などはできない。0そのものを除き、先頭に0を置いてはいけない。浮動小数点数としては 1.0e-10 といった指数表記もできる。ただし、小数点に先行して数字を置く必要がある。精度にかかる仕様や要求はなく、表記された値の取扱精度は実装に依存する。無限大およびNaNの表記はできない。

文字列は（JSONそれ自体と同じく）ダブルクォーテーション「"」でくくったUnicode文字列である。基本的にはJavaScriptの文字列リテラルと同様だが、囲むのにシングルクォートは使えない。バックスラッシュによるエスケープがある。

配列はゼロ個以上の値をコンマで区切って、角かっこでくくることで表現する。例えば以下のよう表現する：

```
["milk", "bread", "eggs"]
```

オブジェクトは**キー**と**値**のペアをコロンで対にして、これらの対をコンマで区切ってゼロ個以上列挙し、全体を波かっこでくくることで表現する。JSONでは連想配列と等価である。例えば以下のよう表現する：

```
{"name": "John Smith", "age": 33}
```

ここで注意することはキーとして使うデータ型は文字列に限ることである。したがって、

```
{name: "John Smith", age: 33}
```

という表記は許されない。この後者の表記はJavaScriptのオブジェクトの表記法としては正しいが、JSONとしては不正な表記である。

4.2 オブジェクトと配列による構造

配列にはゼロ個以上の任意の値を含めることができる。データ型は同じでも、それぞれに異なってもよい。要素が0個の配列は `[]` と表される。以下の例のように、配列の要素が配列であったり、オブジェクトであることが許され、入れ子構造を作ることができる。

```
[ "Varèse", [ 1883, 1965 ], [
    { "title": "Amériques", "year": 1921 },
    { "title": "Intégrales", "year": 1924 },
    { "title": "Ionisation", "year": 1931 }
  ]
]
```

オブジェクト（連想配列）は、キーとなる文字列とそれに対応する値の組を0個以上含む。1つのオブジェクトの中のキーは重複してはならない。値には任意のデータ型を指定できる。要素が0個のオブジェクトは `{}` と表される。以下はオブジェクトの例である。

```
{ "名前": "Varèse",
  "年代": { "生年": 1883, "没年": 1965 },
  "代表作": [
    { "title": "Amériques", "year": 1921 },
    { "title": "Intégrales", "year": 1924 },
    { "title": "Ionisation", "year": 1931, "備考": "打楽器のみ" }
  ]
}
```

配列の要素には順序が存在し、要素の順序が違う配列同士は別のデータである。例えば、配列 `[ 1, 2, 3 ]` と `[ 1, 3, 2 ]` は異なる値である。

一方、オブジェクトの要素には順序の概念は存在せず、プログラムによっては（あるいはその時々によって）要素が違う順序で出力されることもある。例えば次の2つは同じデータである。

```
{"name": "John Smith", "age": 33}

{"age": 33, "name": "John Smith"}
```

4.3 インデントや表示の形式について

それぞれの値や記号の間には、データの区切り（ホワイトスペース）として任意個の空白などを置くことができる。許されているのはいずれもASCII文字の空白、タブ、復帰(0x0D)、改行(0x0A)の4種類の文字だけである。

これらの区切り文字を使って、JSONのデータにインデント（字下げ）を付け、人間にとって読みやすくすることも多いが、これはデータが表現する内容自体には何の影響も及ぼさない。

4.4 文字コードに関する注意

引用符などの記号に ASCII文字以外の文字（例えば「“ ”」）を使ったり、全角の空白を使ってしまうと、分かりにくいエラーの原因になるので注意。

文字列は UTF-8で表記するが、合成文字の扱いについては取り決めがない。例えばNFCとNFD（あるいはそれ以外）が混在した場合、見た目が同じでも違う文字列と判断される可能性がある（例えば「じ」は e38198 というバイト列のこともあれば、e38197e38299 のこともありうる）。異なる開発環境との間で情報をやり取りする場合にはNFCなどに正規化することが望ましい。