* .
* 内容
* [**前書き　Visual Studioの使い方** 7](#_Toc14871730)

[**プログラムの作成手順** 7](#_Toc14871731)

[**・章末テスト** 13](#_Toc14871732)

* [**Lesson 1 はじめの一歩** 14](#_Toc14871733)

[**1.1 C++のプログラム** 14](#_Toc14871734)

[**1.2 コードの入力** 14](#_Toc14871735)

[**1.3プログラムの作成** 14](#_Toc14871736)

[**1.Ex C言語の標準出力関数を知ろう(補足)** 15](#_Toc14871737)

[**Lesson 1 章末テスト** 15](#_Toc14871738)

* [**Lesson 2 C++の基本** 16](#_Toc14871739)

[**2.1 画面への出力** 16](#_Toc14871740)

[**2.2 コードの内容** 16](#_Toc14871741)

[**2.3 文字と数値** 17](#_Toc14871742)

[**Lesson 2 章末テスト** 19](#_Toc14871743)

* [**Lesson 3 変数** 20](#_Toc14871744)

[**3.3 型** 20](#_Toc14871745)

[**3.5 変数の利用** 20](#_Toc14871746)

[**3.5ex(補足) printf関数を使用して、変数の値を出力する。** 21](#_Toc14871747)

[**Lesson 3 中間テスト1** 21](#_Toc14871748)

[**3.6 キーボードからの入力** 22](#_Toc14871749)

[*3.6.ex(補足) C言語の標準入力関数scanf\_s* 23](#_Toc14871750)

[*3.7 定数* 23](#_Toc14871751)

[*Lesson 3 章末テスト* 23](#_Toc14871752)

* [**Lesson 4 式と演算子** 24](#_Toc14871753)

[**4.1 式と演算子** 24](#_Toc14871754)

[*Lesson4 中間テスト１* 25](#_Toc14871755)

[*4.2 演算子の種類* 25](#_Toc14871756)

[**Lesson 4 中間テスト２** 26](#_Toc14871757)

[**Lesson 4 中間テスト3** 29](#_Toc14871758)

[**4.3 演算子の優先順位** 29](#_Toc14871759)

[**4.4 型変換** 29](#_Toc14871760)

[**Lesson 4 中間テスト４** 32](#_Toc14871761)

[**Lesson 4 章末テスト** 33](#_Toc14871762)

* [**Lesson 5 場合に応じた処理** 34](#_Toc14871763)

[**5.1　関係演算子と条件** 34](#_Toc14871764)

[**5.2　if文** 34](#_Toc14871765)

[**5.2 実習** 34](#_Toc14871766)

[**5.3 if～else文** 34](#_Toc14871767)

[**5.3 実習** 34](#_Toc14871768)

[**5.4 if～else if～else** 35](#_Toc14871769)

[**5.4 実習** 35](#_Toc14871770)

[**5.ex\_1** 35](#_Toc14871771)

[**Lesson 5 中間テスト１** 37](#_Toc14871772)

[**5.5 switch文** 37](#_Toc14871773)

[**5.6　論理演算子** 37](#_Toc14871774)

[**Lesson 5 章末テスト** 39](#_Toc14871775)

* [**Lesson 6 何度も繰り返す** 40](#_Toc14871776)

[**Lesson6を始める前に** 40](#_Toc14871777)

[**6.1 for文** 40](#_Toc14871778)

[**Lesson 6 中間テスト１** 42](#_Toc14871779)

[**6.2 while文** 43](#_Toc14871780)

[**6.3　do～while文** 44](#_Toc14871781)

[**6.4　文のネスト** 44](#_Toc14871782)

[**6.4 ex デバッガの機能のウォッチを使ってみよう。** 44](#_Toc14871783)

[*Lesson 6 中間テスト２* 45](#_Toc14871784)

[*6.5 処理の流れの変更* 46](#_Toc14871785)

[**Lesson6 章末テスト** 48](#_Toc14871786)

* [**Lesson 9 配列　基本** 49](#_Toc14871787)

[**9.1 配列の基本** 49](#_Toc14871788)

[**9.2 配列の宣言** 51](#_Toc14871789)

[**9.3 配列の利用** 51](#_Toc14871790)

* [*Lesson\_09 実習課題\_1* 51](#_Toc14871791)

[*課題 1* 51](#_Toc14871792)

[**課題２** 52](#_Toc14871793)

[**課題3** 52](#_Toc14871794)

[*課題４* 52](#_Toc14871795)

* [**Lesson 9 中間テスト１** 52](#_Toc14871796)
* [**Lesson 7 配列** 53](#_Toc14871797)

[**7.1 関数の仕組みを知る** 53](#_Toc14871798)

[**7.2 関数の定義と呼び出し** 55](#_Toc14871799)

[**7.2 Ex\_1関数化を行う理由** 57](#_Toc14871800)

[**7.2 Ex\_2関数は分かりやすい名前をつける。** 61](#_Toc14871801)

[**Lesson\_07 ハンズオン 1** 62](#_Toc14871802)

[**Lesson\_07 実習課題\_1** 62](#_Toc14871803)

[**Lesson\_07 中間テスト1** 62](#_Toc14871804)

[**7.3 引数** 63](#_Toc14871805)

[**7.3 Ex\_3 引数はローカル変数** 63](#_Toc14871806)

[**7.3 Ex\_2 引数の値渡し** 64](#_Toc14871807)

[**7.3 Ex\_2 引数の参照渡し** 65](#_Toc14871808)

[**7.3 Ex\_2.1 参照って何？？？** 65](#_Toc14871809)

[**Lesson\_07 ハンズオン 2** 66](#_Toc14871810)

[**Lesson\_07 実習課題\_2** 66](#_Toc14871811)

[**Lesson\_07 中間テスト2** 66](#_Toc14871812)

[**7.4 戻り値** 67](#_Toc14871813)

[**7.4.1 サンプルプログラム** 67](#_Toc14871814)

[**7.4.2 サンプルプログラム(浮動小数点を返す関数)** 68](#_Toc14871815)

[**7.4.3 サンプルプログラム(bool型を返す関数)** 68](#_Toc14871816)

[**7.4.4　サンプルプログラム(if文の中で関数呼び出し)** 69](#_Toc14871817)

[**7.4.5 サンプルプログラム(if文の中で関数の戻り値をそのまま使う)** 69](#_Toc14871818)

[**7.4 Ex 関数名は分かりやすい名前を考えよう！** 70](#_Toc14871819)

[**Lesson\_07 ハンズオン 3** 71](#_Toc14871820)

[**Lesson\_07 実習課題\_3** 71](#_Toc14871821)

[**Lesson\_07 中間テスト3** 71](#_Toc14871822)

[**7.6 関数宣言** 71](#_Toc14871823)

* [**Lesson 11 いろいろな型** 73](#_Toc14871824)

[**11.3 構造体** 73](#_Toc14871825)

[**11.4 構造体の応用** 75](#_Toc14871826)

[**11.Ex 1 構造体の発展型としてのクラス** 76](#_Toc14871827)

[**Lesson 11 ハンズオン\_1** 79](#_Toc14871828)

[**Lesson 11 実習課題\_1** 79](#_Toc14871829)

[**Lesson 11 実習課題\_2** 79](#_Toc14871830)

[**Lesson 11 実習課題\_Ex\_1** 79](#_Toc14871831)

[**Lesson 11 中間テスト 1** 79](#_Toc14871832)

* [**Lesson 12 クラスの基本** 80](#_Toc14871833)

[**12.1 クラスの宣言** 80](#_Toc14871834)

[**12.1.Ex\_1 ベクトルクラスを作る** 82](#_Toc14871835)

[**12.1.Ex\_2 ベクトルクラスにx、y、zの値を記録するメンバ変数を追加する。** 82](#_Toc14871836)

[**12.1.Ex\_3 ベクトルクラスにx、y、zの値を表示するDisplayメンバ関数を追加する。** 83](#_Toc14871837)

[**12.1.Ex\_4 ベクトルクラスを利用する。** 84](#_Toc14871838)

[**12.1.Ex\_5 インスタンスを複数作る** 85](#_Toc14871839)

[**12.1.Ex\_6 ベクトルの長さを求めてみる。** 85](#_Toc14871840)

[**12.1.Ex\_7 ベクトルの長さを求めるメンバ関数を作成する。** 86](#_Toc14871841)

[**12.1.Ex\_8 ベクトルとスカラーの掛け算** 88](#_Toc14871842)

[**12.1\_Ex\_9 ベクトルとスカラーの掛け算を行うメンバ関数を追加。** 89](#_Toc14871843)

[**12.1\_Ex\_10 実習課題　ベクトルをスカラーで除算(割り算)するメンバ関数を追加** 91](#_Toc14871844)

[**12.Ex\_1 クラスと配列** 92](#_Toc14871845)

[**12.Ex\_1.1　何番目のベクトルが最も大きいか調べる** 94](#_Toc14871846)

[**12.Ex\_1.2 実習課題　何番目のベクトルが最も小さいか調べる** 95](#_Toc14871847)

[**12.Ex\_2 クラスと関数** 96](#_Toc14871848)

[**12.Ex\_2.1　演算子のオーバーロード** 97](#_Toc14871849)

[**12.Ex\_2.2 実習課題** 98](#_Toc14871850)

[**12.2 実習課題　ベクトルクラスに色々なメンバ関数を追加してみよう。** 100](#_Toc14871851)

* [**Lesson 8 ポインタ** 107](#_Toc14871852)

[**8.1　アドレス** 107](#_Toc14871853)

[**8.2 ポインタ** 107](#_Toc14871854)

[**8.2.1 実習　いろいろな型を使って、ポインタを使ってみよう。** 109](#_Toc14871855)

[**8.ex\_1 構造体型の変数とポインタ** 109](#_Toc14871856)

* [**Lesson 9 配列　応用** 110](#_Toc14871857)

[**9.4 配列の応用** 110](#_Toc14871858)

[**Lesson\_09 実習課題\_2** 114](#_Toc14871859)

[**Lesson 9 中間テスト 2** 114](#_Toc14871860)

[**Lesson\_09 実習課題\_3** 115](#_Toc14871861)

[**9.5 配列とポインタの関係** 125](#_Toc14871862)

[**9.5.Ex\_1　多次元配列もメモリ上並んで配置されている。** 126](#_Toc14871863)

[**Lesson\_09 実習課題\_4** 127](#_Toc14871864)

[**9.5.Ex\_1 ポインタ演算の仕組みをさらに詳しく見ていく。** 128](#_Toc14871865)

[**9.5.Ex\_2 ハンズオン** 135](#_Toc14871866)

[**Lesson 9 中間テスト3** 135](#_Toc14871867)

[**9.6 配列と引数** 136](#_Toc14871868)

[**9.7 文字列と配列** 137](#_Toc14871869)

[**Lesson 9 中間テスト4** 140](#_Toc14871870)

[**文字列を操作する** 140](#_Toc14871871)

[**文字列を扱う標準ライブラリ** 141](#_Toc14871872)

[**9.7.Ex\_1　ハンズオン** 141](#_Toc14871873)

[**Lesson\_09 実習課題\_5** 141](#_Toc14871874)

[**Lesson 9 中間テスト5** 141](#_Toc14871875)

[**9.8.Ex 文字列クラスを作ろう** 142](#_Toc14871876)

[**9.8.Ex.1 C言語は文字列をまともに扱えない** 142](#_Toc14871877)

[**9.8.Ex.2 python超入門！！！！** 142](#_Toc14871878)

[**9.8.Ex.3 C言語は文字列のコピーができない！！！** 143](#_Toc14871879)

[**9.8.Ex.4 C言語は文字列の連結ができない！！！** 145](#_Toc14871880)

[**9.8.Ex.5 C言語は文字列が同じかどうか判定できない！！！** 146](#_Toc14871881)

[**9.8.Ex.6 C++には文字列型がある！！！** 147](#_Toc14871882)

[**9.8.Ex.6.1 std::stringを使った文字列のコピー** 148](#_Toc14871883)

[**9.8.Ex.6.2 std::stringを使った文字列の連結** 148](#_Toc14871884)

[**9.8.Ex.6.3 std::stringを使った文字列の比較** 149](#_Toc14871885)

[**9.8.Ex.7 MyStringクラス** 149](#_Toc14871886)

[**9.8.Ex.7.1 HandsOn-1 MyStringクラス(文字列のコピー)** 150](#_Toc14871887)

[**9.8.Ex.7.1.1 メモリの動的確保** 152](#_Toc14871888)

[**9.8.Ex.7.1.3　動的に確保したメモリの解放** 157](#_Toc14871889)

[**9.8.Ex.7.1.4 文字列のコピーで代入演算子を使えるようにしてみよう。** 159](#_Toc14871890)

[**Lesson 9 中間テスト６** 161](#_Toc14871891)

[**9.8.Ex.7.2 HandsOn-2 MyStringクラス(文字列の連結)** 161](#_Toc14871892)

[**9.8.Ex.7.2.1 文字列の連結で+=演算子を使えるようにしてみよう。** 162](#_Toc14871893)

[**9.8.Ex.7.3　Hands-On3　MyStringクラス(文字列の比較)** 164](#_Toc14871894)

[**Lesson 9 実習課題\_6** 165](#_Toc14871895)

* [**Lesson 10 大規模なプログラムの作成** 166](#_Toc14871896)

[**10.1 変数とスコープ** 166](#_Toc14871897)

[Sample10.1 166](#_Toc14871898)

[Sample10.2 167](#_Toc14871899)

[Sample10.3 167](#_Toc14871900)

[**10.1.ex.1 for文のループ変数のスコープは？？？** 168](#_Toc14871901)

[Sample10.4 168](#_Toc14871902)

[Sample10.5 168](#_Toc14871903)

[**10.2 記憶寿命** 169](#_Toc14871904)

[Sample.10.6 169](#_Toc14871905)

[**10.2.ex.2 メンバ変数の記憶寿命は？？？** 169](#_Toc14871906)

[Sample.10.7 169](#_Toc14871907)

[**Lesson 10 中間テスト 1** 170](#_Toc14871908)

[**10.3.ex.1 new/deleteとmalloc/free** 171](#_Toc14871909)

[Sample10.8(new/delete版) 171](#_Toc14871910)

[Sample10.9(malloc/free版) 171](#_Toc14871911)

[**10.3.ex.2 C++ではnew/deleteを利用する** 171](#_Toc14871912)

[10.3.ex.3 new/deleteとmalloc/freeの違い 172](#_Toc14871913)

[Sample10.10(new/delte版) 172](#_Toc14871914)

[Sample10.11(maloc/free版) 173](#_Toc14871915)

[Sample10.12 173](#_Toc14871916)

[**Lesson10 中間テスト２** 173](#_Toc14871917)

* [**Lesson 11 いろいろな型** 174](#_Toc14871918)

[11.3.ex.1 enumの典型的な使われ方 174](#_Toc14871919)

[Sample11.1(enumを配列の要素数、添え字として活用するサンプル) 174](#_Toc14871920)

[**11.3.ex.2 enumを配列で利用するメリット** 175](#_Toc14871921)

[Sample11.2 175](#_Toc14871922)

[**11.3.ex.2.1 可読性の向上** 179](#_Toc14871923)

* [**Lesson 12 クラス(発展)** 180](#_Toc14871924)

[**12.2 メンバへのアクセス制限** 180](#_Toc14871925)

[**12.2.ex\_1 privateなメンバ変数の値を参照する** 183](#_Toc14871926)

[**実習課題\_2** 184](#_Toc14871927)

[**実習課題\_3** 185](#_Toc14871928)

[**中間テスト\_1** 185](#_Toc14871929)

[**12.ex.1 関数のオーバーロード** 186](#_Toc14871930)

[Sample12.1 186](#_Toc14871931)

[**12.3 引数とオブジェクト** 186](#_Toc14871932)

[**Sample12.2** 187](#_Toc14871933)

[**実習課題\_4** 188](#_Toc14871934)

[**中間テスト\_2** 188](#_Toc14871935)

* [Lesson 13 クラスの機能 189](#_Toc14871936)

[13.1 コンストラクタの基本 189](#_Toc14871937)

[13.2 コンストラクタのオーバーロード 189](#_Toc14871938)

[Sample13.1 189](#_Toc14871939)

[Sample13.2 190](#_Toc14871940)

[**13.ex.1 デフォルトメンバ初期化子** 191](#_Toc14871941)

[Sample13.3 191](#_Toc14871942)

* [**13.ex.2 Singletonパターン** 192](#_Toc14871943)

[インスタンスを一つしか作れなくする 192](#_Toc14871944)

[**グローバルなアクセスポイントを提供する** 192](#_Toc14871945)

[**典型的なSingletonパターンの実装方法** 192](#_Toc14871946)

[Sample13.4 192](#_Toc14871947)

[**Hands-On Singletonを実装してみる。** 193](#_Toc14871948)

[**中間テスト** 196](#_Toc14871949)

* [**Lesson 14 いろいろなクラス** 196](#_Toc14871950)
* [**14.ex.1 より良い設計を求めて** 196](#_Toc14871951)

[**14.ex.1.1　クラスの責任を考える** 196](#_Toc14871952)

[**14.ex.1.2 凝集度と結合度** 197](#_Toc14871953)

[**14.ex.1.3 インターフェースに対するプログラミング** 197](#_Toc14871954)

[14.ex.1.4 UpdateMethodパターン 198](#_Toc14871955)

[14.3x.1.4 Stateパターン 201](#_Toc14871956)

[Ex.1.2 Hands-On ゲームのデータをセーブしてみよう(テキスト版) 206](#_Toc14871957)

[Ex.1.3 Hands-On ゲームのデータをロードしてみよう(テキスト版) 207](#_Toc14871958)

[Ex.1.4 ファイルの書き込み(バイナリモード) 208](#_Toc14871959)

[Ex.1.4.1　ファイルを開く 208](#_Toc14871960)

[Ex.1.4.2　データの書き込み 208](#_Toc14871961)

[Ex1.4.3 ファイルを閉じる 208](#_Toc14871962)

[Sample6.cpp 209](#_Toc14871963)

[Ex.1.5 ファイルの読み込み(バイナリモード) 209](#_Toc14871964)

[Ex.1.5.1 ファイルを開く 209](#_Toc14871965)

[Ex.1.5.2 データの読み込み 209](#_Toc14871966)

[Ex 1.5.3 ファイルを閉じる 210](#_Toc14871967)

[Sample7.cpp 210](#_Toc14871968)

[Ex.1.6 Hands-On ゲームデータをセーブしてみよう(バイナリ版) 211](#_Toc14871969)

[Ex.1.7 Hands-On ゲームデータをロードしてみよう(バイナリ版) 211](#_Toc14871970)

# **前書き　Visual Studioの使い方**

## **プログラムの作成手順**

プロジェクトLesson0を作ってみましょう。大文字の入力はshiftキーを押しながらで行えます。

**１. プロジェクトの作成**

StartメニューからVisualStudioを起動する。



ファイル→新規作成→プロジェクトを選択



ドキュメントの下にc\_purapura\_1というフォルダを作って、そこにプロジェクトを作りましょう。\_(アンダースコア)はshift + /で入力できます。図１の赤枠を参照。

図１



**2. プロジェクトのプロパティを変更する。**

ソリューションエクスプローラーでプロジェクトを選択→右クリック→プロパティ



**3.ソースファイルの追加**

ソリューションエクスプローラーでプロジェクトを選択→右クリック→追加→新しい項目を選択。



**4. ソースコードの入力**

　追加したソースファイルを選択。



選択が出来たら、下記のようにコードを入力する。



次の４点に注意する。

①　スペースは半角。入力を忘れないように。図２，３の青枠。

　②　大文字、小文字に注意。

　③　#　、<　、（　、)、{ 、}、”(ダブルクォーテーション)などはShiftキーを押しながら入力する。図２，３の赤枠。

　④　;(セミコロン)を忘れない。上記コード、図２，３の緑枠。

⑤ 日本語入力→英字入力は半角/全角ボタンで切り替えることができる。図２の黄色枠。

**図２**



**図３**



**5. ソースコードのビルド→実行**

　Ctrl+F5でデバッガなしで実行できます。下記図の赤枠。



**６. コマンドプロンプト(黒いウィンドウ)がすぐに消える**

　　この現象が発生した場合は、2をやり直す。

## **・章末テスト**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSfWXuo1SqrWla4WEOIGEVUOJlqHg_vBP6p-5FZtuyqzRLNs5A/viewform?usp=sf_link>

# **Lesson 1 はじめの一歩**

## **1.1 C++のプログラム**

**機械語(p.3)**

　　→こんなの。



## **1.2 コードの入力**

**統合環境を使ってみる(p.5)**

本校の授業ではVisual Studio(統合環境)を使います。

**テキストエディタにコードを入力する。(p.6)**

　新しいプロジェクト、Lesson1を作成して、ソースファイルを追加して、p.7のSample1.cppの内容を入力する。

**コンパイルエラーを起こしてみる**

コードを書くことができて、実行出来たら、わざとコンパイルエラーを起こしてどうなるか確認してみましょう。

## **1.3プログラムの作成**

**コンパイラを実行する(P9)**

オブジェクトファイルを確認してみる。

**オブジェクトファイルをリンクする(P9)**

実行ファイルができていることを確認する。

## **1.Ex C言語の標準出力関数を知ろう(補足)**

**printf関数**

　coutはC++の標準出力関数ですが、printfはC言語の標準出力関数となります。C++はC言語のスーパセット言語となっており、C言語の機能はすべて使えます。では、下記のコードを入力してみてください。

|  |
| --- |
| #include <stdio.h>  int main()  {  printf(“ようこそ C++へ！”);  return 0; } |

C言語検定では標準出力関数として、printf関数が使用されていることと、ほかの言語でも似たようなprint関数が用意されているので、こちらを使えるようになることも重要です。

## **Lesson 1 章末テスト**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/1NPbcFTvH37P5kJ3pHvOIB0GWXZoNYUG93aCOaEYAXYA/edit>

# **Lesson 2 C++の基本**

## **2.1 画面への出力**

**新しいコードを入力する(p.16)**

新しいプロジェクトLesson2を作成して、Sample.1.cppの内容を入力して実行してください。

**色々な出力方法を知る(p.18)**

Sample2.cppの内容を入力して実行してください。

**改行には\nを使う**

　テキストにも書かれていますが、重要です。しっかりと覚えましょう。

## **2.2 コードの内容**

**main()関数**

　main関数はエントリーポイントと呼ばれる特殊な関数です。プログラムのスタート地点となる関数です。

**#include <iostream>をコメントアウトしてみる。**

#include <iostream>をコメントアウトして、コンパイルしてみましょう。下記のようなエラーが表示されたと思います。



エラーをダブルクリックすると、エラーが起きている箇所にジャンプできます。試してみてください。

**using namespace std;をコメントアウトしてみる。**

こちらもコメントアウトしてコンパイルしてみてください。先ほどと同様のエラーが表示されたと思います。



**重要**

　エラーをダブルクリックするとコンパイルエラーが起きている箇所にジャンプできることをしっかりと覚えましょう。

## **2.3 文字と数値**

**数値リテラル**

　123、579、30.0など

**文字リテラル**

　‘A’、’B’、’c’など。**シングルクォーテーションで囲まれたもの。**一文字を表す。文字列とは違う！！！

**文字列リテラル**

　“ABC”、”ようこそC++”など。**ダブルクォーテーションで囲まれたもの。**

**エスケープシーケンス**

　全部を覚える必要はない。重要なのは下記の３点です。

　\n 改行コード

　\0 文字列の終わり(詳細はLesson9.7)

\マークをつけることで、\や’や”などの特殊文字を文字列に組み込むことができる。

　　→どういうこと？

例えば、下記のような文字列を表示する場合を考えてみましょう。



ダブルクォーテーションは文字列リテラルを囲むものという、特殊文字となっているため、下記のようなプログラムではコンパイルエラーになります。



文字としてダブルクォーテーションを出力したい場合は、下記のように記述する必要があります。



その他のエスケープシーケンスが必要になったら、ネットで検索をすればＯＫです。ググりましょう。

**実習1 (時間 5分)**

　一度のcoutの実行で下記のような表示ができるようにしてみよう。



**printf関数でも同じ**

　エスケープシーケンスのルールはprintf関数でも同じです。



**実習２(時間 5分)**

一度のprintf関数の実行で下記のような表示ができるようにしてみよう。



**８進数**

10を8とする表記法。

**16進数**

10を16とする表記法

## **Lesson 2 章末テスト**

***下記のURLのテストを行いなさい。***

<https://docs.google.com/forms/d/e/1FAIpQLSdNjkkwZ4M0L5_ujSSc-GULWBWXiA_dRPSHZ8ct2XpUrocsjg/viewform?usp=sf_link>

# **Lesson 3 変数**

## **3.3 型**

数が多くて、いきなり覚えるのは難しいと思いますが、今は下記の4点を覚えてください。

　整数型　 　　　　int

浮動小数点型 　　　　float

文字型 　　　　char

unsignedをつけると　 符号なしになる。(頭の隅の方にでも置いておいてください。)

|  |
| --- |
| int hoge = －10; //hogeは符号付き整数型。負数も記憶できる。  unsigned int hoge2 = 20; //hoge2は符号なし整数型。負数は記憶できない。 |

## **3.5 変数の利用**

**変数に値を代入する**

|  |
| --- |
| int num;  num = 3; |

＝が代入であることに注意してください。数学であればイコールですが、プログラムでは代入です。右辺の値を左辺に代入します。簡単に思えるかもしれませんが、実は、ここの理解で躓く人が多いです。

**p.50のコードを入力して結果を確認しよう**

　新しいプロジェクトLesson3を作成して、p.50のコードを入力して確認しましょう。

**変数を初期化する**

　変数は宣言することで、数値を記録するための領域がメモリ上に確保されます。では下記のようなコードの場合、どのような値が表示されるのでしょうか？

|  |
| --- |
| int num; //numという変数を用意する。  cout << num << “\n”; 何が表示される？ |

**変数の値を変更する**

　p.53のコードを入力して、動作を確認しましょう。

**他の変数の値を代入する**

　p.55のコードを入力して、動作を確認しましょう。

**値の代入についての注意**

　p.56のコードを入力して、動作を確認しましょう。

　doubleはfloatでもＯＫです。

## **3.5ex(補足) printf関数を使用して、変数の値を出力する。**

coutを使用して変数の中身を出力する方法は見てきましたが、printf関数も変数の値を出力することができます。int型の変数の値を出力する場合は下記のように記述します。



また、下記のように記述することで、複数の変数の値を出力できます。



## **Lesson 3 中間テスト1**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLScDqY-iHav90m71Sr1uTCXsHrTW1t5TJLkW1oNA0xNZUAtfmg/viewform?usp=sf_link>

## **3.6 キーボードからの入力**

**キーボードから入力する**

最近使ったプロジェクトからLesson3を起動して、P.60のSample5.cppの内容を打ち込んで下さい。



**C++標準入力関数 cin**

cinを実行するとキーボードからの入力待ちになります。

**二つ以上の数値を入力する**

　p.61のSample6の内容を打ち込んでください。

## ***3.6.ex(補足) C言語の標準入力関数scanf\_s***

***C++の標準入力関数はcinですが、C言語にも同様の関数のscanf\_sがあります。下記のようなコードを入力することで、cinと同じ動作になります。入力して確認してください。***



***(注意)正確には、C言語の標準入力関数はscanf関数なのですが、この関数はセキュリティホールが存在しているため、マイクロソフトがより安全なscanf\_s関数を用意しています。使い方はscanfと同じなので、混乱しないようにしてください。***

## ***3.7 定数***

P.63のSample7.cppのコードを入力してみてください。

**変更することのできない変数にconstをつける。**

プログラマのミスで値を変更してしまうヒューマンエラーをなくす。

今、定数の利点を理解するのは難しいと思うので、これは後期に詳しくやります。

## ***Lesson 3 章末テスト***

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSfgUfWrA1P6vhlj-UqzQr3CBhHYmOxpzsFYOo0mdsmuQpSU9A/viewform?usp=sf_link>

# **Lesson 4 式と演算子**

## **4.1 式と演算子**

**式の値を出力する(p.71)**

新しいプロジェクトLesson4を作成して、Sample1.cppを入力して実行してください。

プログラムの世界では＊が掛け算になります。＊はshift＋：、と+はshift＋；で入力できます。下記の図の赤枠を参照。



**色々な演算をする(p.71)**

下記の２点をしっかりと意識しながら、Sample2.cppのコードを入力してください。

・ 変数と変数の演算を行える。

* 変数と数値リテラルとの演算を行える。

**変数num1の値に１を足し、その値を再度num1に代入する(p.73)**

　下記のコードに注目してください。

|  |
| --- |
| num1 = num1 + 1; |

数学的にはおかしな式です。 = の記号が等しいではなく、代入であったことを思い出してください。右辺の結果(num1+1)を左辺に代入しているため、このような記述が可能になります。

**キーボードから入力した値を足し算する(p.74)**

Sample3.cppの内容を入力して、実行してください。

**実習 1(10分)**

Lesson4***の内容を改造して、下記のような表示をできるようにしなさい。***

***また、除算の演算子は / です。***



## ***Lesson4 中間テスト１***

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSfyMQIL4RSN9mjOLdaeRnAJv2mDBi4U-WjkY4mywBdxOnw-uQ/viewform?usp=sf_link>

## ***4.2 演算子の種類***

**いろいろな演算子(p.76)**

非常に多くの演算子があります。残念なことに、ここに記載されている演算子はすべて重要です。世のプログラマーはすべてを覚えていると思います。でも、安心してください。覚えようとしなくてもプログラムを書いていると、自然と覚えてしまいます。

　まずは四則演算(＋－×÷)+αをマスターしましょう。

では、Lesson4を最近使ったプロジェクトから開いて、p.77のSample4.cppのコードを入力して、実行してください。

**実習2(10分)**

Lesson4を改造して、次の動画のような挙動になるプログラムを作成しなさい。

出席番号で０～４のグループに振り分けるプログラム。

<https://www.youtube.com/watch?v=lRwpa-nbEuI&feature=youtu.be>

**インクリメント・デクリメント演算子(p.79)**

これも四則演算子なのですが、少し変わっています。よく使う演算子なのでしっかりと覚えましょう。

**インクリメント・デクリメントの前置と後置(p.80)**

前置と後置で実行結果が変わることがあります。普段プログラムを書くときは、この規則を意識しなくてもいいように書く方が優れている場合がほとんどですが、資格・検定の試験でこれを問う問題がでることがあるので、覚えましょう。

Lesson4にp.81のSample.cppのコードを入力して、実行してください。

下記が前置と後置の挙動の違いの覚え方です。

**・前置なので、代入する前にインクリメント・デクリメントする。**

**・後置なので、代入した後でインクリメント・デクリメントする。**

ただし、これに依存するようなコードを書くことは可読性を下げることになるので、下記のようなコードを書くことを推奨します。

**代入する前にインクリメントしたい場合**

|  |
| --- |
| a++; 　//インクリメントしてから  b = a; //代入する。 |

**代入した後でインクリメントしたい場合**

|  |
| --- |
| b = a; //代入してから、  a++; //インクリメントする。 |

## **Lesson 4 中間テスト２**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSdAUAv0FP5S5zUoNbznYQUhUVoGBl82IVA6wdh3LZrc0I-oRA/viewform?usp=sf_link>

**代入演算子(p.83)**

Lesson4にp.85のSample6.cppを入力して実行してください。

**sizeof演算子**

　この演算子は、配列(Lesson9)の要素数を調べるときなどに使用されることがあります。国家試験でもよく出てくる演算子です。例えば下記のように使います。

|  |
| --- |
| int num = sizeof(int); |

この演算子は読んで字のごとく、「size of int」int型 の(of) サイズ(size)を求めてくれます。

では、int型のサイズはいくつだったでしょうか？教科書のp.43に戻って確認してみましょう。

**確認テスト**

　下記のプログラムの実行結果を答えなさい。

|  |
| --- |
| int main()  {  int hoge = 0;  int size = sizeof(hoge);  cout << size << “\n”;  return 0;  } |

答え

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| int main()  {  char hoge = ‘w’;  int size = sizeof(hoge);  cout << size << “\n”;  return 0;  } |

答え

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**4Hands-On 2**

Lesson 4に下記のコードを入力して動作を確認しなさい。



**シフト演算子(p.88)**

下記の２点を覚える。

　・１ビット左にシフトすると値は倍になる。

　・１ビット右にシフトすると値は半分になる。

**Hands-On 3**

Lesson4に下記のコードを入力して動作を確認しなさい。



## **Lesson 4 中間テスト3**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLScoCxdw4wk-kCYEglROeYF9pFD-tRO5MOZj9S-LSrc4pMY2Jw/viewform?usp=sf_link>

## **4.3 演算子の優先順位**

**演算子の優先順位とは(P.90)**

下記の２点を抑えましょう。

**・四則演算の優先順位は数学と同じ。**

**・優先順位が分からなかったら()を使えば良い。**

この２点を押さえておけば、p.91～p92の表は覚える必要はありません。この表を答えられるプログラマーなどいないのですから。

**同じ優先順位の演算子を使う(P93**

これも難しいことを考える必要はありません。抑える点は同じです。

**・四則演算の優先順位は数学と同じ。**

**・優先順位が分からなかったら()を使えば良い。**

## **4.4 型変換**

**大きなサイズの型に代入する(P.94)**

Lesson4にSample8.cppの内容を入力して実行してください。

**小さなサイズの型に代入する(P.95)**

小さなサイズの型に代入すると、値が失われるのは正しいですが、このテキストに書かれている内容は、ちょっと不正確です。このSample9は正しくは、**浮動少数点型の変数の値を整数型の変数に代入すると、小数点の値が失われる**です。

Lesson4に下記のコードを入力して実行してください。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  float fnum = 50.5f;　 //floatは32bitの浮動小数点。  int inum = fnum; 　　 //intも32bitなので、値は失われないはず？  cout << inum << '\n'; //教科書の説明は正しくない。小数点は失われる！  return 0;  } |

では、「小さなサイズの型に代入すると値が失われる」の正しい説明となるコードを見てみましょう。

Lesson4に下記のコードを入力して実行してください。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {    int inum = 100000; //intは32bitは符号付きの整数型。  //表現できる範囲は、-2,147,483,648 ～ 2,147,483,647  short snum; 　　　　　　//shortは16bitの符号付きの整数型。  //表現できる値の範囲は、-32,768 ～ 32,767  snum = inum; //snumに100000という値は表現できないので、代入すると・・・  cout << snum << '\n'; //値が失われる！  return 0;  } |

**キャスト演算子を使う(P.96)**

明示的な型変換を行う。

→教科書に書かれていることをしたところで、結果は何も変わらない。

　→え？じゃぁキャストってなんのためにするの？

型変換を行うと、小さなサイズの型に代入すると値が失われてしまいます。つまり下記のようなコードを書いてしまった場合、致命的な不具合を生み出すことがあります。

(このコードは書かなくていい)

|  |
| --- |
| #include <iostream>  using namespace std;  int g\_playerHp = 100000; //int型のプレイヤーのHPを表すグローバル変数。  int main()  {  short playerHp = g\_playerHp; //プレイヤーのHPをshort型のローカル変数に代入！  playerHp -= 100; //プレイヤーのHPを100減らすことが目的なのだが・・・。  g\_playerHp = playerHp; //変更したHPを書き戻す。  cout << g\_playerHp << '\n'; //なんてこった。  return 0;  } |

このようなヒューマンエラーを防ぐために、明示的ではない、小さな型への変換を行うコードを書いた場合にコンパイルエラーにすることができます。

次の設定を行ってから、コンパイルを行ってみてください。

プロジェクトのプロパティを開く



プロパティページ/C++/全般/警告レベルを4にする。



プロパティページ/C++/全般/警告をエラーとして扱うを「はい」にする。



この設定でLesson4のコンパイルを行うと、下記のようなエラーと警告が出てくると思います。



こうすることで、意図していない型変換を行うコードを書いてしまった場合はコンパイルエラーとなって、ヒューマンエラーをなくすことができます。そして、このエラーは下記のように明示的にキャストすることによって、消すことができます。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int inum = 100000; //intは32bitは符号付きの整数型。  //表現できる範囲は、-2,147,483,648 ～ 2,147,483,647  short snum; //shortは16bitの符号付きの整数型。  //表現できる値の範囲は、-32,768 ～ 32,767  snum = **(short)**inum; //snumに100000という値は表現できないので、代入すると・・・  cout << snum << '\n'; //値が失われる！  return 0;  } |

つまり、キャストというのは「小さな型変換で、値が失われるのは知っているけど、これは正しいコードだから、黙ってコンパイルしろ！」ということをコンパイラに教えてやる行為となります。

## **Lesson 4 中間テスト４**

　下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLScNMpAM-h4_Bek28g8_-tVNiXn8Rj-bos_NAHbE0QlagOpyTA/viewform?usp=sf_link>

**異なる型どうしで演算する(P.98)**

この説明もちょっと正しくありません。例えば下記のコードの場合、結果は小さい型の浮動小数点となります。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  long long llValue = 10; //long longは64bitの整数型。  float fValue = 0.5f; //floatは32bitの浮動小数点型。  long long llResult = llValue \* fValue; //教科書の説明なら64bitの整数型になるはずが、  　 //32bitの小数点型になっている！！！  return 0;  } |

型変換のルールは下記です。

　・どちらか一方のオペランドが浮動所数点なら、結果は浮動小数点になる。

　・オペランドが両方とも整数型　or　浮動小数点型なら、型サイズの大きい方になる。

## **Lesson 4 章末テスト**

　下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSeDBBADoB0WS-1l8sT8foEhJoobCE0-_bRu_UVdXioWAZqWgA/viewform?usp=sf_link>

# **Lesson 5 場合に応じた処理**

## **5.1　関係演算子と条件**

**条件の仕組みをしる(p.106)**

例えばゲームであれば、「**もしも**、コントローラーのＡボタンが押されたら」

　　　　　　　　　　　　　　→ジャンプする

**条件を記述する(p.107)**

　C++では==が数学の＝と同じになる。左辺と右辺が等しければtrueを返す演算子。

関係演算子は、条件が成立する時にtrue、成立しなければfalseを返してくる演算子です。

## **5.2　if文**

**if文の仕組みを知る(p.111)**

新しいプロジェクトLesson5を作成して、Sample1.cppの内容を入力して、実行してください。

**if文で複数の分を処理する(p.113)**

Lesson5を改造して、Sample2.cppのコードを入力して、実行してください。

**ブロックにしないと(p.116)**

if文を使うときは、条件を満たすときの処理が、たとえ１文であったとしても、必ず{}で囲むのをお勧めします。ただし、{}で囲んでいないコードを読むこともあるので、知識として知っておくことは重要です。

## **5.2 実習**

下記の動画のようなプログラムを実装しなさい。

<https://www.youtube.com/watch?v=EM0hp-jT15I&feature=youtu.be>

## **5.3 if～else文**

Lesson5を改造して、Sample3.cppの内容を入力して、実行してください。

## **5.3 実習**

下記の動画のようなプログラムを実装しなさい。

<https://www.youtube.com/watch?v=qEYbe7RDEkg&feature=youtu.be>

## **5.4 if～else if～else**

**if～else if～elseの仕組みを知る(p.122)**

Lesson5を改造して、Sample4.cppを入力して実行してください。

## **5.4 実習**

下記の仕様を満たすプログラムを実装しなさい。

・年齢の入力を促す。

・20歳未満なら、「未成年ですね」と表示する。

・20歳以上なら「成人ですね」と表示する。

・ただし、下記の年齢の場合は長寿の祝いを表示する。

60歳の場合は「還暦おめでとうございます。」

77歳の場合は「喜寿おめでとうございます。」

88歳の場合は「米寿おめでとうございます。」

99歳の場合は「白寿おめでとうございます。」

下記の動画を参考にして実装しなさい。

<https://www.youtube.com/watch?v=JmguNYN1aaU&feature=youtu.be>

## **5.ex\_1**

if文はbool型のtrue、falseの値によって分岐します。そのため、下記のようなコードも合法です。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int age;  cout << "年齢を入力してください。\n";  cin >> age;  bool result = age >= 20; //>=のような関係演算子はbool型の値を返す。  if (result){ //if文は単にbool型の値によって分岐するだけなので、これもＯＫ。  cout << "あなたは成人ですね。\n";  }    return 0;  } |

また、整数の０はfalseに０以外はtrueに暗黙的に変換されるため、下記のようなコードも合法です。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int age;  cout << "年齢を入力してください。\n";  cin >> age;  //int型の変数ageをif文にそのまま書く。  //整数型は暗黙的に、bool型に変換される。  if (age){  cout << "あなたは0歳ではないですね。\n";  }  else {  cout << "あなたは0歳ですね。\n";  }  return 0;  } |

！(否定演算子)を使うと、bool型の結果が反転します。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int age;  cout << "年齢を入力してください。\n";  cin >> age;  bool result = age >= 20; //>=のような関係演算子はbool型の値を返す。  　　　//resultの結果を反転しているので、このif文の条件が成立するということは、  //未成年だということになる。  if (!result){  cout << "あなたは未成年ですね。\n";  }  return 0;  } |

下記のようにも書ける。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int age;  cout << "年齢を入力してください。\n";  cin >> age;  　　　//age>=20の結果を反転しているので、このif文の条件が成立するということは、  //未成年だということになる。  if (!(age >= 20)){  cout << "あなたは未成年ですね。\n";  }  return 0;  } |

## **Lesson 5 中間テスト１**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSfu0l2a0RE_FFheu8jhl6mmcJVzDEyxFF6kIhvOvKfVe5RE6w/viewform?usp=sf_link>

## **5.5 switch文**

**switch文の仕組みを知る(p.126)**

switch文で出来ることは、if～else if ～else文でもできるので、どちらを使ってもＯＫ。

Lesson5を改造して、Sample5.cppを入力して実行してみてください。

**break文が抜けていると(p.129)**

***break文がないと下のケースが実行されます。このようなことを意図して行うコードもありますが、慣れないうちは必ずbreakを書くようにしましょう。***

## **5.6　論理演算子**

**論理演算子の仕組みを知る(p.131)**

***(条件Ａ)　&& (条件Ｂ)***

***→条件Ａがtrue かつ 条件Ｂがtrue***

***(条件Ａ)　|| (条件Ｂ)***

***→条件Ａがtrue または 条件Ｂがtrue***

***では、&&(論理積)を使えるケースを見てみましょう。***

***例) 20歳以上かつ男性かどうかを判断する場合の条件式(論理演算を使わない場合)***

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int age, gender;  cout << "年齢を入力してください。\n";  cin >> age;  cout << "性別を入力してください。0 : 男性、1 : 女性\n";  cin >> gender;  //&&はif文のネストでも表現できるが・・・  if (age >= 20) {  if (gender == 0) {  cout << "あなたは２０歳以上で、男性ですね。\n";  }  }    return 0;  } |

***例) 20歳以上かつ男性かどうかを判断する場合の条件式(論理演算を使う場合)***

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int age, gender;  cout << "年齢を入力してください。\n";  cin >> age;  cout << "性別を入力してください。0 : 男性、1 : 女性\n";  cin >> gender;  //&&を使って、条件をまとめることができる。  if (age >= 20 && gender == 0) {  cout << "あなたは２０歳以上で、男性ですね。\n";  }    return 0;  } |

***続いて、||(論理和)を使えるケースを見てみましょう。***

***例)飲食店で女性と子供は2割引のサービスを行っている場合の条件式(論理演算を使わない場合。)***

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  //例えば、飲食店で女性と子供は2割引のサービスを行っている場合。  int age, gender;  cout << "年齢を入力してください。\n";  cin >> age;  cout << "性別を入力してください。0 : 男性、1 : 女性\n";  cin >> gender;  //||は複数のif文でも表現できるが・・・  if (age < 13) {  cout << "あなたは２割引きのサービスを受けられます。\n";  }  if (gender == 1) {  cout << "あなたは２割引きのサービスを受けられます。\n";  }  return 0;  } |

***例)飲食店で女性と子供は2割引のサービスを行っている場合の条件式(論理演算を使う場合。)***

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  //例えば、飲食店で女性と子供は2割引のサービスを行っている場合。  int age, gender;  cout << "年齢を入力してください。\n";  cin >> age;  cout << "性別を入力してください。0 : 男性、1 : 女性\n";  cin >> gender;  //||を使って、条件をまとめる。  if (age < 13 || gender == 1) {  cout << "あなたは２割引きのサービスを受けられます。\n";  }    return 0;  } |

***更に論理演算を組みあわせると、じゃんけんの勝敗判定も行えます。***

|  |
| --- |
| //playerという変数にプレイヤーの手、comという変数にコンピュータの手の情報が入っている。  //0がグー、１がチョキ、２がパーです。  if (( player == 0 && com == 1 ) //プレイヤーがグー、コンピュータがチョキ。  || ( player == 1 && com == 2 ) //プレイヤーがチョキ、コンピュータがパー。  || ( player == 2 && com == 0 ) //プレイヤーがパー、コンピュータがグー。  ) {  cout << "あなたの勝ちです。\n";  } |

## **Lesson 5 章末テスト**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLScWLHDqPBaJc3ISuLyL13M93NhYJJwO010PPaCld7PK5EUltw/viewform?usp=sf_link>

# **Lesson 6 何度も繰り返す**

## **Lesson6を始める前に**

Lesson 6に記載されているサンプルコードですが、今後の勉強に悪い影響を与えかねない要素があるので、教科書のサンプルコードは打ち込まずに、PDFファイルのサンプルコードを打ち込んでください。この理由はLesson9で配列を勉強する時に説明ます。また、ループの繰り返しの処理は、for文、while文、do~while文の３つがありますが、for文だけ覚えてもらえばＯＫです。while文、do～while文で出来ることは、すべてfor文で実現可能なので、プログラムに慣れてきたときに覚えてもらえれば十分です。ただし、国家試験、検定にはすべて出てくる可能性があるので、国家試験を取りたい人は、すべて覚える必要があります。

## **6.1 for文**

**for分の仕組みを知る(p.142)**

VisualStudioでLesson\_6のプロジェクトを作成して、Sample1.cppを入力してください。

for文は一行に３つの処理が記述されていることに注意！

for( int i = 0 ; i < 5; i++)

ループブロックの処理を一度、実行した後で行われる処理。

ループ変数 iの宣言と初期化

ループの継続条件

Sample1.cpp

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  for (int i = 0; i < 5; i++) {  cout << "繰り返しています。\n";  }  cout << "繰り返しが終わりました。\n";  return 0;  } |

もう少し、見ていきましょう。下記のコードを見てください。

Sample1.cppは下記のようなコードと同義です。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int i = 0; //変数iの宣言と初期化。  LOOP: //これはラベルと呼ばれるもの。  if (i < 5) { //ループの継続判定。  cout << "繰り返しています。\n";  i++; //ループ変数のインクリメント。  goto LOOP; //goto文でラベルLOOPにジャンプする。  }  cout << "繰り返しが終わりました。\n";  return 0;  } |

**注意！今回for文の説明のためにgoto文を使用しましたが、goto文はスパゲッティコード(読みづらいコード)を生み出しやすくするものとして、多くの開発で使用することが非推奨となっている構文です。絶対に使用しないように！！！**

for文の３つの式の意味が分かれば、下記のようなコードが書けることが分かります。

下記のコードを入力して、F10キーでステップ実行を行い、処理の流れを確認してください。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  //ループ変数は0以外で初期化することもできるし、  //ループの継続判定もi>=0などにもできる。  //ループブロックの処理が終わった後の処理もデクリメントでも良い。  for (int i = 5; i >= 0; i--) {  //ループ変数はループ内で使用できる。  cout << i << "\n";  }  return 0;  } |

このコードをgoto文を使って書くと下記のようになります。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int i = 5; //変数iの宣言と初期化。  LOOP: //これはラベルと呼ばれるもの。  if (i >= 0) { //ループの継続判定。  cout << i << "\n";  i--; //ループ変数のインクリメント。  goto LOOP; //goto文でラベルLOOPにジャンプする。  }    return 0;  } |

**変数をループ内で使う(p.145)**

ここまでの例で見てきたように、ループ変数をループ内で使うことができます。Lesson\_6にSample2.cppを入力して実行してください。

**for文を応用する(p.146)**

Lesson\_6にSample3.cppとSample4.cppを入力して、動作を確認してください。

Sample3.cpp（入力した数だけ\*を表示する。）

|  |
| --- |
| #include <iostream>  ループの終了判定に変数が使えることに注目！  using namespace std;  int main()  {  int num;  cout << "いくつ\*を表示しますか？\n";  cin >> num; //数を入力させる。    for (int i = 0; i < num; i++) {  cout << "\*"; //入力した数だけ\*を繰り返し表示する。  }  cout << "\n";  return 0;  } |

Sample4.cpp (入力した数までの合計を求める。)

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int kazu;  int goukei = 0;    cout << "いくつまでの合計を求めますか？\n";  cin >> kazu; //数を入力させる。  for (int i = 1; i <= kazu; i++) {  goukei += i ; //例えば、kazuに5が入力されたら、iの値は１～５となる。  　　//なので、iの値を加算していくと1～5までの数値の合計になる。  }  cout << "１から" << kazu << "までの合計値は" << goukei << "です。\n";  return 0;  } |

## **Lesson 6 中間テスト１**

下記のＵＲＬのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLScAPHvnMY3l2-PuMblKSBgfP0XIBRcexBqBHIzURobXUUCUFA/viewform?usp=sf_link>

## **6.2 while文**

**while文の仕組みを知る(p.149)**

while文は、for文をgoto文に置き換えて記述したものとよく似ています。

Sample5.cppを入力して実行出来たら、F10を押して、ステップ実行をおこなって動作を確認しなさい。

Sample5.cpp

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int i = 0; //これがループ変数。  while (i < 5) { //これがループの継続条件。  cout << i << "番目の繰り返しです。\n";  i++; //これがループの処理が終わったときの処理。  }  cout << "繰り返しが終わりました。\n":  return 0;  } |

**条件の記述を省略する(p.151)**

p.152のようなコードを書くときはforではなく、whileが使われることが多いです(私もたぶんwhileを使います)。whileが使われる理由は、このようなコードを書くときは、for文に比べて、キータイプの量が減ることと(キータイプ量が減ると腱鞘炎に苦しむプログラマーが減りますよね？)、whileを覚えている人からすると、読みやすいコードになるからです。

　では、Sample6.cppを記入して、動作を確認してください。

Sample6.cpp

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int num = 1;  while (num == 1) { //numが1の間は処理を繰り返す。  cout << "整数を入力してください。(0で終了)\n";  cin >> num;  cout << num << "が入力されました。\n";  }  cout << "繰り返しが終わりました。\n";  return 0;  } |

## **6.3　do～while文**

　たぶん学生のうちに使うことはないと思います。do～while文は下記の一点だけを覚えておきましょう。

**ループの中の処理が必ず最低一回は行われる。**

## **6.4　文のネスト**

**文をネストする(p.157)**

Sample8.cppを入力して動作を確認したら、F10を押してステップ実行で処理の流れを確認しなさい。

Sample8.cpp

ループ変数がiとjで違うことに注意！

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  for (int i = 0; i < 5; i++) {  for (int j = 0; j < 3; j++) {  cout << "iは" << i << "、jは" << j << "\n";  }  }  return 0;  } |

## **6.4 ex デバッガの機能のウォッチを使ってみよう。**

プログラムを書いていくうえで、強力な武器となるデバッガの機能のウォッチ(変数の値を見るため)を使って、6.4のループ変数のiとjの中身を見てみましょう。ウォッチの使い方は下記の動画を参考にしてみてください。

<https://www.youtube.com/watch?v=dAwYii65J40&feature=youtu.be>

**if文などと組み合わせる(‘p.159)**

Sample9.cppを入力して動作を確認してください。動作が確認出来たら、下記の動画のように、ステップ実行を行い、変数i、j、chをウォッチに追加して、処理の流れを追いかけなさい。

***Sample9.cpp***

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int ch = 0;  for (int i = 0; i < 5; i++) {  for (int j = 0; j < 5; j++) {  if (ch == 0) {  cout << '\*'; //\*を出力したら、次は-を表示  ch = 1; //するように、chに1を代入。  }  else {  cout << '-'; //\*を出力したら、次は\*を表示  ch = 0; //するように、chに0を代入。  }  }  cout << "\n"; //内側のループが終わったら改行します。  }  return 0;  } |

***動画(音声が入っているので再生する時は注意してください。)***

<https://www.youtube.com/watch?v=MPCf2NUSCUA&feature=youtu.be>

## ***Lesson 6 中間テスト２***

***下記のURLのテストを行いなさい。***

<https://docs.google.com/forms/d/e/1FAIpQLSfPy-snsI_VVDKWYV18xdYtVq171je48_McWFPbH9udXML3sQ/viewform?usp=sf_link>

## ***6.5 処理の流れの変更***

**break文の仕組みを知る(p.161)**

Sample10.cppを入力して、動作を確認したら、F10キーでステップ実行して、処理の流れを確認しなさい。

Sample10.cpp

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int res;  cout << "何番目でループを注視しますか？(1～10)\n";  for (int i = 0; i < 10; i++) {  cout << i << "番目の処理です。\n";  if (i == res) {  break; //指定した回数で繰り返しを終了します。  }  }  } |

break文は、例えば下記のような仕様を実装する時などに使えます。RPGの製作を行っていて、一緒に戦う仲間のAIを作っている場合を考えて下さい。「ヒーラーのAIであれば、HPが500以下になった味方にヒーリングを行う。」といった仕様を実装することがあるかもしれません。そのような場合に、下記のコードのように味方からＨＰ500以下のキャラクターを検索する必要があります。

|  |
| --- |
| for (int i = 0; i < mikataCharacterNum; i++) {  if (mikataCharacterHP[i] < 500) {  //500以下のキャラクターを発見したので、ヒールをかけて  //ループを抜ける。  ・  ・  省略  ・  ・  break;  }  } |

**繰り返しをネストしている場合、その内側の文でbreak文を使うと、外側のブロックに処理が移る(p.162)**

これは下記のようなコードの話です。入力して動作を確認したのち、F10でステップ実行を行い、処理の流れを確認してください。

内側のループを抜ける。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  for (int i = 0; i < 3; i++) {  for (int j = 0; j < 5; j++) {  if (j == 3) {  break;  }  //jは2までしか出力されない！  cout << "i : " << i << "、j : " << j << "\n";  }  }  return 0;  } |

**continue分の仕組みを知る(p.165)**

Sample12.cppの内容を入力して動作を確認してください。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int res;  cout << "何番目の処理を飛ばしますか？(0～9)\n";  cin >> res;  for (int i = 0; i < 10; i++) {  if (i == res) {  continue;  }  cout << i << "番目の処理です。\n";  }  return 0;  } |

continue文は下記のような仕様を実装したい場合に使えます。

「1000人のグループの名簿から20歳の人にだけ、成人おめでとうと表示する。」

この仕様をcontinue文を使って、実装すると下記のようになります。

|  |
| --- |
| for (int i = 0; i < 1000; i++) {  if (ageList[i] != 20) {  //20歳以外はスキップ。  continue;  }  cout << "成人おめでとう。\n";  } |

ただし、このコードは下記のように書くこともできます。

|  |
| --- |
| for (int i = 0; i < 1000; i++) {  if (ageList[i] == 20) {  cout << "成人おめでとう。\n";  }    } |

このように、continue文を使用しなくても、ほかの方法があるため、今無理して覚える必要はないと思います。

## **Lesson6 章末テスト**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSe9Qi4I82H6LzhIYxmw-wPiGth-ihf3PCZ0zmdcEPjd4MANgg/viewform?usp=sf_link>

# **Lesson 9 配列　基本**

## **9.1 配列の基本**

**配列の仕組みを知る(p.256)**

「ただし、こうしてたくさんの変数が登場すると、コードが複雑で読みにくくなってしまう場合があります。(p.256)」

　→読みやすいコードを書くと言ことは、ソフトウェアを開発していくうえで、とても重要

要素となります。今のうちから意識してコードを書いていきましょう。

例えば、あなたがMMORPGの100人参加可能な、レイドバトルのボスキャラクターのAIを実装している場合を考えてみて下さい。そのボスキャラクターのAIは「HPが一番低いプレイヤーに対して攻撃を行う。」という仕様だったとします。配列を知らない不幸なプログラマー(プロのプログラマーに、そんな人はいないと思いますが)は、悪夢のようなコードを書いてしまうでしょう。

|  |
| --- |
| //プレイヤー構造体。  struct Player {  int hp;  int attack;  ・  ・  省略  ・  ・  };  //プレイヤー100人を表すグローバル変数を100個用意する。  Player player00;  Player player01;  Player player02;  Player player03;  Player player04;  Player player05;  Player player06;  Player player07;  Player player08;  Player player09;  Player player10;  ・  ・  省略  ・  ・  Player player98;  Player player99;  //攻撃対象となるプレイヤーを検索する関数。  Player\* FindAttackTargetPlayer()  {  Player\* attackTargetPlayer = nullptr;  //一番HPが小さい奴を探す。  int hp = 9999999; //とりあえずあり得ないくらい大きくしておく。40億超えたらダメだよ。  //プレイヤー00  if (player00.hp < hp) {  //プレイヤー00の方がHPが小さいので更新。  hp = player00.hp;  attackTargetPlayer = &player00;  }  //プレイヤー01  if (player01.hp < hp) {  //プレイヤー01の方がHPが小さいので更新。  hp = player01.hp;  attackTargetPlayer = &player01;  }  //プレイヤー02  if (player02.hp < hp) {  //プレイヤー02の方がHPが小さいので更新。  hp = player02.hp;  attackTargetPlayer = &player02;  }  //プレイヤー03  if (player03.hp < hp) {  //プレイヤー03の方がHPが小さいので更新。  hp = player03.hp;  attackTargetPlayer = &player03;  }  //プレイヤー04  if (player04.hp < hp) {  //プレイヤー04の方がHPが小さいので更新。  hp = player04.hp;  attackTargetPlayer = &player04;  }  //プレイヤー05  if (player05.hp < hp) {  //プレイヤー05の方がHPが小さいので更新。  hp = player05.hp;  attackTargetPlayer = &player05;  }  //プレイヤー06  if (player06.hp < hp) {  //プレイヤー06の方がHPが小さいので更新。  hp = player06.hp;  attackTargetPlayer = &player06;  }  //プレイヤー07  if (player07.hp < hp) {  //プレイヤー07の方がHPが小さいので更新。  hp = player07.hp;  attackTargetPlayer = &player07;  }  ・  ・  省略  ・  ・  //プレイヤー98  if (player98.hp < hp) {  //プレイヤー98の方がHPが小さいので更新。  hp = player98.hp;  attackTargetPlayer = &player98;  }  //プレイヤー99  if (player99.hp < hp) {  //プレイヤー99の方がHPが小さいので更新。  hp = player99.hp;  attackTargetPlayer = &player98;  }  //見つかった攻撃対象のプレイヤーを返す。  return attackTargetPlayer;  } |

もし、このプログラムを配列を知っていた幸福なプログラマーが書けばこうなるでしょう。

|  |
| --- |
| //プレイヤー構造体。  struct Player {  int hp;  int attack;  ・  ・  省略  ・  ・  };  //プレイヤー100人を表すグローバル変数を要素数100の配列として個用意する。  Player player[100];  //攻撃対象となるプレイヤーを検索する関数。  Player\* FindAttackTargetPlayer()  {  Player\* attackTargetPlayer = nullptr;  //一番HPが小さい奴を探す。  int hp = 9999999; //とりあえずあり得ないくらい大きくしておく。40億超えたらダメだよ。  for (int i = 0; i < 100; i++) {  if (player[i].hp < hp) {  //このプレイヤーの方がHPが小さいので更新する。  hp = player[i].hp;  attackTargetPlayer = &player[i];  }  }  //見つかった攻撃対象のプレイヤーを返す。  return attackTargetPlayer;  } |

とても短くシンプルで分かりやすいプログラムになりました。

## **9.2 配列の宣言**

**配列の添え字は0からはじまるので、最後の添え字は「要素数-1」にとなる。**

## **9.3 配列の利用**

　新しいプロジェクト、Lesson09を作成してSample1.cppを入力してください。

**配列を初期化する(p.262)**

# ***Lesson\_09 実習課題\_1***

***下記のURLからzipファイルをダウンロードして、次の１～4の実習を行いなさい。***

<https://1drv.ms/u/s!AnpsoGtakIkVi41brqVH5jmOnCTrVQ>

## ***課題 1***

サンプルプログラムのkadai\_09\_01を改造して、試験の合計点を求めて表示するプログラムを作成しなさい。

## **課題２**

　サンプルプログラムのkadai\_09\_02を改造して、試験の平均点を求めて表示するプログラムを作成しなさい。

## **課題3**

　サンプルプログラムのkadai\_09\_03を改造して、試験の平均点以上の点数を取っている学生の人数を表示するプログラムを作成しなさい。

## ***課題４***

　サンプルプログラムのkadai\_09\_04を改造して、試験の点数を昇順（小さい順）に表示できるようにしなさい。

課題４のヒント動画

　ヒント１　値の交換のアルゴリズムの解説と、ソースコードの解説

<https://www.youtube.com/watch?v=Y-nZ8dYaY-A&feature=youtu.be>

　ヒント２　バブルソートアルゴリズムの解説

<https://www.youtube.com/watch?v=xvbUuWzBOHY&feature=youtu.be>

　ヒント３　バブルソートのソースコード解説

<https://www.youtube.com/watch?v=gOmwRxBH8WM&feature=youtu.be>

# **[L](https://www.youtube.com/watch?v=gOmwRxBH8WM&feature=youtu.be)esson 9 中間テスト１**

　下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSe-NfkV14Qi6lv8wgo6qYVaa8ODC3LK89Qy35QdTaVa_7juCA/viewform?usp=sf_link>

# **Lesson 7 配列**

## **7.1 関数の仕組みを知る**

**「C++でも、複雑なコードを書くようになってくると、たびたび行わなければならない一定の処理が出てくる場合があります。(P.170)」**

例えば、これまでいくつかの関数をみなさんは使ってきています。

・コンソールに文字列を表示する処理

　printf関数、cout関数(実はこれも関数。ただしちょっと特殊な書き方をす

しているので注意が必要。)

・ユーザーからの入力を受け付ける関数。

scanf関数、cin関数(coutと同様に、これも関数。)

・キーボードの入力判定

GetAsyncKeyState関数。これはmicrosoftが提供している関数です。

・パックマン、落ち物などの文字表示関数。

kbcDrawMoji関数。この関数は私が作成した関数です。

もしこの関数という機能がなかったら、あなたはコンソールに文字を出力する時に下記のようなコードを毎回コピペで複製する羽目になります！！！

|  |
| --- |
| int main()  {  //Hello worldと表示する！  char\* fmt = "Hello world\n";  for (;;) {  unsigned long long ui;  long long i;  char \*s = NULL;  double d = 0.0;  int sign = 0;  int flags = 0;  int length = 0;  int precision = 0;  int tmp = 0;  INTEGER int\_type = 0;  while (\*fmt && \*fmt != '%') \_\_putc(\*fmt++);  if (\*fmt == '\0') { va\_end(ap); break; }  fmt++;  while (mystrchr("'-+ #0", \*fmt)) {  switch (\*fmt++) {  case '\'': flags |= THOUSAND\_GROUP; break;  case '-': flags |= LEFT\_JUSTIFIED; break;  case '+': flags |= WITH\_SIGN\_CHAR; sign = '+'; break;  case '#': flags |= ALTERNATIVE; break;  case '0': flags |= ZERO\_PADDING; break;  case ' ': flags |= WITH\_SIGN\_CHAR; sign = ' '; break;  }  }  if (\*fmt == '\*') { length = va\_arg(ap, int); fmt++; }  else { while (\_isnumc(\*fmt)) length = (length \* 10) + \_ctoi(\*fmt++); }  if (\*fmt == '.')  {  fmt++;  if (\*fmt == '\*') { fmt++; precision = va\_arg(ap, int); }  else { while (\_isnumc(\*fmt)) precision = precision \* 10 + \_ctoi(\*fmt++); }  }  while (mystrchr("hljzt", \*fmt)) {  switch (\*fmt++) {  case 'h': int\_type--; break;  case 'l': int\_type++; break;  case 'j': /\*intmax : long \*/  case 'z': /\*size : long \*/  case 't': /\*ptrdiff : long \*/  int\_type = L; break;  }  }  switch (\*fmt) {  case 'd':  case 'i':  i = get\_signed(ap, int\_type);  if (i < 0) { i = -i; sign = '-'; }  put\_integer(\_\_putc, i, 10, length, sign, flags);  break;  case 'u':  ui = get\_unsigned(ap, int\_type);  put\_integer(\_\_putc, ui, 10, length, sign, flags);  break;  case 'o':  ui = get\_unsigned(ap, int\_type);  put\_integer(\_\_putc, ui, 8, length, sign, flags);  break;  case 'p':  length = sizeof(long) \* 2;  int\_type = L;  sign = 0;  flags = ZERO\_PADDING | ALTERNATIVE;  case 'X':  flags |= CAPITAL\_LETTER;  case 'x':  ui = get\_unsigned(ap, int\_type);  put\_integer(\_\_putc, ui, 16, length, sign, flags);  break;  case 'c':  i = get\_signed(ap, C);  \_\_putc(i);  break;  case 's':  s = va\_arg(ap, char \*);  if (s == NULL) s = "(null)";  tmp = strlen(s);  if (precision && precision < tmp) tmp = precision;  length = length - tmp;  if (!(flags & LEFT\_JUSTIFIED))  {  while (length > 0) { length--; \_\_putc(' '); }  }  while (tmp--) { \_\_putc(\*s++); }  while (length-- > 0) { \_\_putc(' '); }  break;  case '%':  \_\_putc('%');  break;  default:  while (\*fmt != '%') fmt--;  break;  }  fmt++;  }  return 0;  } |

このプログラムはHello worldという文字列を表示するためだけに127行ものプログラムを書いています。これが一度だけというならまだしも、なにか文字列を表示するたびに、あなたはこれをコピーアンドペーストする必要があるのです！

しかし、関数を利用すると、この127行ものプログラムをまとめることができて、下記のように簡単に再利用することができます。

|  |
| --- |
| int main()  {  //HelloWorldと表示。  printf("Hello world\n");  //こんにちは世界と表示。  printf("こんにちは世界\n");  return 0;  } |

## **7.2 関数の定義と呼び出し**

**関数を定義する**

関数を定義する＝関数の処理を実装する＝関数の本体といった意味合いになります。後々関数宣言というものが出てきますが、それと混合しないように注意してください。

関数定義は関数の本体、プログラムの本体といった意味です。

関数は下記のように定義します。

戻り値　関数名( 引数リスト )

{

波カッコの間が関数の処理！！！

}

戻り値と引数リストは7.3と7.4で説明します。今は関数名だけ覚えてください。

**関数呼び出し**

新しいプロジェクト、Lesson7を作成してSample1.cppを入力してください。

関数を呼び出すには下記ように書く必要があることに注意してください。

関数名();

かっこが必要なことを忘れてしまう学生が多いです。()を忘れないようにしてください。

Sample1.cppが入力出来たら、VisualStudioのステップイン(F11)を使って、処理の流れを追いかけてみてください。

参考動画

<https://www.youtube.com/watch?v=vljzUcT3XiU&feature=youtu.be>

**関数を何度も呼び出す**

Sample2.cppを入力して下さい。

入力出来たら、ステップインを使って、処理の流れを追いかけてみてください。

## **7.2 Ex\_1関数化を行う理由**

**プログラムを綺麗に書くため**

　Lesson7では関数化を行う理由として、「プログラムの再利用性」を挙げています。これはもちろん正しいのですが、再利用を行うことができるプログラムというのは、それほど多くありません。そのため、それが最大の理由だと考えてしまうと、関数化を行うことが難しくなってきます。しかし、多くのプロジェクトでは、一か所でしか呼ばれない関数というものが沢山あります。つまり、**再利用されていない関数**が沢山あるのです。しかも、それらの関数は、再利用されている関数よりも多いのです！

では、なぜ関数化するのでしょうか？答えは、プログラムを見やすく、綺麗に描くためです。下記のプログラムは制作演習で行っている落ち物ゲームのゲームループです。

**関数化を行っていない落ち物ゲームのゲームループ**

|  |
| --- |
| //ここからゲームループ。  while (true) {  //1フレームの開始。  //フォントのサイズの設定。  const HANDLE hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE);  CONSOLE\_FONT\_INFOEX info = { sizeof info };  info.dwFontSize.X = 32;  info.dwFontSize.Y = 32;  SetCurrentConsoleFontEx(hConsole, 0, &info);  //コンソールの縦と横の行数を設定。  HANDLE hCons = GetStdHandle(STD\_OUTPUT\_HANDLE);  COORD dwSize = { 9999, 9999 };  SetConsoleScreenBufferSize(  hCons, // HANDLE hConsoleOutput  dwSize); // COORD dwSiz  //コンソールの高さを設定。  SMALL\_RECT consoleWindow = { 0, 0, 20, 20 };  BOOL ret = SetConsoleWindowInfo(  hCons, // HANDLE hConsoleOutput  TRUE, // BOOL bAbsolute  &consoleWindow);// CONST SMALL\_RECT \*lpConsoleWindow  //フレームバッファをクリア。  memset(sFrameBuffer, 0, sizeof(sFrameBuffer));  //背景を描画。  //二次元配列のmapを参照して、背景を描画していく。  for (int y = 0; y < 16; y++) {  for (int x = 0; x < 16; x++) {  if (map[y][x] == 0) {  //2次元配列のmap[y][x]に0が入っていたら、  //座標x, yの場所に空白文字を描画する。  kbcDrawMoji(x, y, ' ');  }  else if (map[y][x] == 1) {  //2次元配列のmap[y][x]に1が入っていたら、  //座標x, yの場所に@を描画する。  kbcDrawMoji(x, y, '@');  }  }  }  //プレイヤー１の移動処理。  for (int playerNo = 0; playerNo < 2; playerNo++) {  if (playerNo == 0) {  //1プレイヤー。  //プレイヤー１の移動処理。  if (GetAsyncKeyState(VK\_LEFT) != 0) {  //左のキーが押されたときの処理をここに記述する。  player[playerNo].posX--;  }  if (GetAsyncKeyState(VK\_RIGHT) != 0) {  player[playerNo].posX++;  }  }  else if (playerNo == 1) {  //2プレイヤー。  //続いてプレイヤー２の移動処理。  if (GetAsyncKeyState('A') != 0) {  //左のキーが押されたときの処理をここに記述する。  player[playerNo].posX--;  }  if (GetAsyncKeyState('D') != 0) {  player[playerNo].posX++;  }  }  //壁判定。  if (player[playerNo].posX < 1) {  player[playerNo].posX = 1;  }  if (player[playerNo].posX > 14) {  player[playerNo].posX = 14;  }  //落ち物との衝突判定。  for (int i = 0; i < otimonoArray.size(); i++) {  if (otimonoArray[i].posX == player[playerNo].posX  && otimonoArray[i].posY == player[playerNo].posY  ) {  otimonoArray[i].dead = 1;  //ゲット出来ていたらカウントアップ。  player[playerNo].score++;  }  }  //プレイヤー１を描画する。  if (player[playerNo].posX < FRAME\_BUFFER\_WIDTH && player[playerNo].posY < FRAME\_BUFFER\_HEIGHT) {  sFrameBuffer[player[playerNo].posY][player[playerNo].posX] = 'P';  }  }  //落ち物のプログラムを実行。  gameCount++;  if (gameCount == 10) {  //30フレーム経過したら新しい落ち物を発生させる。  Otimono otimono;  //落ち物のX座標をランダムで決定。  otimono.posX = (rand() % 14) + 1;  otimono.posY = 0;  //ゲームカウントを0に戻す。  gameCount = 0;  otimonoArray.push\_back(otimono);  }  //全ての落ち物を落下させる。  for (auto& otimono : otimonoArray) {  if ((gameCount % 5) == 0) {  //5フレームに一度落下する。  otimono.posY += 1;  }  }  //リストから削除。  otimonoArray.erase(std::remove\_if(otimonoArray.begin(), otimonoArray.end(), [&](auto& otimono)->bool {  return otimono.posY == 18 || otimono.dead == 1;  }), otimonoArray.end());  //落ち物を描画。  for (auto& otimono : otimonoArray) {  if (otimono.posX < FRAME\_BUFFER\_WIDTH && otimono.posX < FRAME\_BUFFER\_HEIGHT) {  sFrameBuffer[otimono.posY][otimono.posX] = '\*';  }  }  //スコアを表示する。  char scoreText[256];  sprintf\_s(scoreText, "Player1　スコア %d", player[0].score);  for (int i = 0; scoreText[i] != '\0'; i++) {  kbcDrawMoji(0 + i, 18, scoreText[i]);  }  sprintf\_s(scoreText, "Player2　スコア %d", player[1].score);  for (int i = 0; scoreText[i] != '\0'; i++) {  kbcDrawMoji(0 + i, 19, scoreText[i]);  }  //1フレームの終了。  //フレームバッファの内容を画面に表示する。  for (int i = 0; i < FRAME\_BUFFER\_HEIGHT; i++) {  for (int j = 0; j < FRAME\_BUFFER\_WIDTH; j++) {  std::cout << sFrameBuffer[i][j];  }  std::cout << "\n";  }  //32ミリ秒眠る。  Sleep(16);  //標準出力コンソールのハンドルを取得。  hCons = GetStdHandle(STD\_OUTPUT\_HANDLE);  COORD pos;  pos.X = 0;  pos.Y = 0;  //標準コンソールのカーソルを0行目、0列に戻す。  SetConsoleCursorPosition(hCons, pos);  CONSOLE\_CURSOR\_INFO cci;  // CONSOLE\_CURSOR\_INFO構造体の現在の状態を取得する  GetConsoleCursorInfo(hCons, &cci);  // メンバ変数であるbVisibleがカーソルの表示・非表示を制御する変数なので、これをFALSEにする事でカーソルを非表示にできる  cci.bVisible = FALSE;  // 変更した構造体情報をコンソールWindowにセットする  SetConsoleCursorInfo(hCons, &cci);  } |

いかがでしょうか。これが関数化を行っていない落ち物のゲームループのプログラムです。150行ほどのプログラムとなっております。では、続いて、関数化を行っている落ち物のゲームループを見てみましょう。

**関数化を行っている落ち物ゲームのゲームループ**

|  |
| --- |
| //ここからゲームループ。  while (true) {  //1フレームの開始。  **kbcBeginFrame();**  //背景を描画。  **DrawBackground()**;  //プレイヤーの処理。  for (int i = 0; i < 2; i++) {  **MovePlayer(player[i], i)**;  //落ち物との衝突判定。  **CheckHitPlayerAndOtimono(player[i])**;  //プレイヤー１を描画する。  **DrawPlayer(player[i])**;  }  //落ち物の描画処理。  **DrawOtimono()**;  //落ち物の移動処理。  **MoveOtimono();**  //スコアを表示する。  **DrawScore(player);**  //1フレームの終了。  **kbcEndFrame();**  } |

いかがでしょうか？関数化を行っている方だと、１フレームに何をしているのか？という処理の流れが一目霊山ではないでしょうか？この二つのプログラムは処理的には同じことをしています。ですが、１フレームの処理の流れをどちらが把握しやすいか？と問われると後者だと思います。このプログラムで網掛けになっている関数は再利用性を考慮して関数化を行っているわけではありません。一か所にしか記述されていませんね。そして、驚くべきことにすべての関数がそうです。

　関数化の最も大きなメリットは、「意味のある処理に名前を付けて、分割することでコードが読みやすくなる」ということだということが分かります。関数の再利用性は棚上げにして、まずは処理を分割するということを行っていきましょう。

この二つのプログラムは下記のURLにアップされています。両方とも動作するので、興味がある方はどうぞ。

<https://1drv.ms/u/s!AnpsoGtakIkVi41lbigVIJ8xjmOKHw>

## **7.2 Ex\_2関数は分かりやすい名前をつける。**

　関数化を行う最も大きな理由として、**読みやすい、理解しやすい**プログラムを書くというものがあるということを勉強しました。さて、関数の名前というのは自由に決めていいということを学びましたが、ではどんな名前でもつけてもいいのでしょうか？（まぁ不幸なことに、それでもコンパイルは通るのですが）下記のプログラムを見てみてください。

|  |
| --- |
| //ここからゲームループ。  while (true) {  //1フレームの開始。  A();  //背景を描画。  B();  //プレイヤーの処理。  for (int i = 0; i < 2; i++) {  C(player[i], i);  //落ち物との衝突判定。  D(player[i]);  //プレイヤー１を描画する。  E(player[i]);  }  //落ち物の描画処理。  F();  //落ち物の移動処理。  G();  //スコアを表示する。  H(player);  //1フレームの終了。  I();  } |

さて、いかがでしょうか？このプログラムは理解しやすいでしょうか？コメントがあるから、まだ何とかなる気がしますか？では、コメントをあまり書かない、ずぼらなプログラマーが書いたプログラムを見てみましょう。

|  |
| --- |
| //ここからゲームループ。  while (true) {  A();  B();  for (int i = 0; i < 2; i++) {  C(player[i], i);  D(player[i]);  E(player[i]);  }  F();  G();  H(player);  I();  } |

どうでしょうか？もはや、このプログラムは暗号文のようなものです。A();という関数呼び出しが、どんな処理をしているかさっぱり分かりません。では、しっかりと分かりやすい関数名を付けたプログラムを見てみましょう。

|  |
| --- |
| //ここからゲームループ。  while (true) {  kbcBeginFrame();  DrawBackground();  for (int i = 0; i < 2; i++) {  MovePlayer(player[i], i);  CheckHitPlayerAndOtimono(player[i]);  DrawPlayer(player[i]);  }  DrawOtimono();  MoveOtimono();  DrawScore(player);  kbcEndFrame();  } |

こちらもコメントはありませんが、関数名を読めば、何となく何をしているのかが分かります。これが分かりやすい、綺麗なプログラムです。プログラマーは理系っぽいというイメージがあるかもしれませんが、実は多彩な能力が求められます。このように分かりやすい名前を考える力は、文系の人の方が得意だと思われます。

## **Lesson\_07 ハンズオン 1**

サンプルプログラムのLesson\_07\_01/Questionを使用して、ハンズオン1～3を行いなさい。

## **Lesson\_07 実習課題\_1**

　サンプルプログラムのLesson\_07\_01/Questionを使用して、実習1～3を行いなさい。

サンプルプログラムのURL

<https://1drv.ms/u/s!AnpsoGtakIkVi41mnfcGk3d6lgyIsw>

実習課題の解説動画

<https://www.youtube.com/watch?v=Tr0u_uXMYsk&feature=youtu.be>

## **Lesson\_07 中間テスト1**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSfLoJBcmo9nAZqAPd6QBH6pfn6muV94rzxCbIjAgGnY9cZloA/viewform?usp=sf_link>

## **7.3 引数**

**引数を渡して関数を呼び出す(p.181)**

Sample3.cppを入力してください。

**キーボードから入力する(p.183)**

Sample4.cppを入力してください。

**複数の引数をもつ関数を使う(p.185)**

Sample5.cppを入力して下さい。

## **7.3 Ex\_3 引数はローカル変数**

　関数の引数はローカル変数と呼ばれるものとなります。ローカル変数ってなに？って思うと思いますが、みなさんがこれまで学んできた変数がローカル変数です。ローカル変数は関数の中で定義されて、その関数の中でだけ有効な変数です(ちょっと嘘です。詳しくは10.1のスコープで勉強します。

下記のプログラムを入力してください。コンパイルエラーが出るはずです。

|  |
| --- |
| #include <iostream>  using namespace std;  void buy()  {  //carPriceはローカル変数なのでアクセスできないよ！！！  cout << carPrice << "万円の車を買いました。\n";  }  int main()  {  int carPrice = 100; //車の値段。  //buyという関数を呼び出す。  buy();  return 0;  } |

下記のようなコンパイルエラーが発生したのではないでしょか？



carPriceという変数はmain関数の中で定義されていて、main関数の中でだけ有効なローカル変数と呼ばれるものです。そのため、buy関数では使用することができないため、　　　carPriceなんて変数はありませんよ？といった意味のエラーが発生します。そこで、buy関数に車の値段を渡すことができる引数を追加しましょう。

下記のプログラムを入力して、コンパイルエラーを修正して下さい。

|  |
| --- |
| #include <iostream>  using namespace std;  void buy( **int price** )  {  cout << price << "万円の車を買いました。\n";  }  int main()  {  int carPrice = 100; //車の値段。  //buyという関数を呼び出す。  buy( **carPrice** );  return 0;  } |

網掛けになっている箇所が修正箇所です。

関数の引数は、その関数のローカル変数になります。つまり、priceという変数はbuy関数の中でだけ有効な変数ということです。

## **7.3 Ex\_2 引数の値渡し**

　7.3 Ex\_1の引数渡しは値渡しと呼ばれる渡し方で、変数のコピーを渡しています。下記のコードを入力して動作を確認してください。

|  |
| --- |
| #include <iostream>  using namespace std;  void Add( int v )  {  v += 10;  }  int main()  {  int x = 10;  Add(x);  cout << "xの値は" << x << "です。\n";  return 0;  } |

Add関数の中でvに10加算していますが、xの値は20ではなく、10と表示されたと思います。これはAdd関数の変数vが、main関数の変数xのコピーだからです。このプログラムは下記のようなコードが実行されていると考えてください。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int x = 10;  //Add関数はこんな処理が実行されていることと同じ。  int v = x; //vにxの値をコピー！  v += 10; //vを10加算。  cout << "xの値は" << x << "です。\n"; //vが加算されているだけなので、xは当然10!  return 0;  } |

では、次の節では変数の値渡しではなく、参照渡しについて見ていきましょう。

## **7.3 Ex\_2 引数の参照渡し**

　さて、引数を値渡しで関数に渡すと、呼び出し元の変数の値は変更されないことを学びました。しかし、関数の中で値を変えたい場合は結構あります。そこで、引数の参照渡しについて学びましょう。

## **7.3 Ex\_2.1 参照って何？？？**

　では、引数の参照渡しを学ぶ前に、そもそも参照ってなに？というところから学びましょう。参照とは、変数の別名定義となります。では、下記のコードを入力して動作を確認してください。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int x = 10; //xは本名。  int& v = x; //vはxの別名！！！(芸名やペンネームみたいなもの)  v += 10; //vを10加算。  //vはxの別名なだけで正体は同じなので、当然xの値も変わっている。  cout << "xの値は" << x << "です。\n";  return 0;  } |

下記のように記述すると、vは参照となります。

int& v = x;

この場合、vはxの別名となります。芸能人の芸名みたいなものですね。名前は違うけど、正体は同じとなります。これを使うと、関数の中で変更した値を、呼び出し元に反映させることができます。下記のコードを入力して動作を確認してください。

|  |
| --- |
| #include <iostream>  using namespace std;  //vは参照！！！  void Add(int& v)  {  v += 10;  }  int main()  {  int x = 10;  //Add関数の引数vは参照なので、xの別名となる。  Add(x);  //Add関数の中で10加算されているので20と表示される！！！  cout << "xの値は" << x << "です。\n";  return 0;  } |

## **Lesson\_07 ハンズオン 2**

サンプルプログラムのLesson\_07\_02/Questionを使用して、ハンズオン1～3を行いなさい。

## **Lesson\_07 実習課題\_2**

　サンプルプログラムのLesson\_07\_02/Questionを使用して、実習1～3を行いなさい。

サンプルプログラムのURL

<https://1drv.ms/u/s!AnpsoGtakIkVi41n5ETsy70uYECQtg>

ハンズオン ２の動画

<https://www.youtube.com/watch?v=-NVyUeAXYtg&feature=youtu.be>

実習課題 2の解説動画

<https://www.youtube.com/watch?v=LgpJbw54_eU&feature=youtu.be>

## **Lesson\_07 中間テスト2**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSeAMKOEI_7SVdpkahmPo5DGLh8luvXFVAXn-cy1j7yejyZhag/viewform?usp=sf_link>

## **7.4 戻り値**

**戻り値の仕組みを知る**

　「ブロックの最後に行き着かなくても、return文が処理されたところで関数の処理が終了します。」VisualStudioのプロジェクト、Lesson7を立ち上げて、下記のサンプルプログラムを入力してください。

## **7.4.1 サンプルプログラム**

|  |
| --- |
| #include <iostream>  using namespace std;  //  // 概要  // クラスの最高点数を求める関数です。  // 引数  // score[] 点数の配列。  // numStudent 生徒数  // 戻り値  // 最高点数が返ります。  //  int GetMaxScore(int score[], int numStudent)  {  //最高点を線形探索で探す。  int maxScore = 0;  for (int i = 0; i < numStudent; i++) {  if (score[i] == 100) {  //100点が見つかったので、  //これ以上の点数はないので、探索を打ち切る。  return 100;  }  if (score[i] > maxScore) {  //この点数の方が高い。  maxScore = score[i];  }  }  return maxScore;  }  int main()  {  //Aクラスのスコア。  int scoreA[4] = { 60, 20, 80, 90 };  int saikouTokuten = 0;  //Aクラスの最高点を求める。  saikouTokuten = GetMaxScore(scoreA, 4);  cout << "Aクラスの最高点は" << saikouTokuten << "です。\n";  //続いて、Bクラスの最高得点を求める。  int scoreB[4] = { 20, 100, 80, 30 };  //F11でステップインして、21行目のreturnで関数が終了しているのを確認してみよう。  saikouTokuten = GetMaxScore(scoreB, 4);  cout << "Bクラスの最高点は" << saikouTokuten << "です。\n";  return 0;  } |

戻り値の型はintとは限らない。浮動小数点を返す場合はfloat型。

下記のサンプルプログラムを入力してください。

## **7.4.2 サンプルプログラム(浮動小数点を返す関数)**

|  |
| --- |
| #include <iostream>  using namespace std;  //大きい方の値を返す関数。  //浮動小数点の値を返す関数。  float Max(float x, float y)  {  if (x > y) {  //xの方が大きい。  return x;  }  //yの方が大きい。  return y;  }  int main()  {  float v1 = Max(10.5f, 2.5f);  float v2 = Max(25.53f, 20.12f);  cout << "v1の値は" << v1 << "です。\n";  cout << "v2の値は" << v2 << "です。\n";  //引数に変数を渡すこともできる。  float v3 = Max(v1, v2);  cout << "v3の値は" << v3 << "です。\n";  return 0;  } |

true、falseを返す場合はbool型となる。

下記のサンプルプログラムを入力してください。

## **7.4.3 サンプルプログラム(bool型を返す関数)**

|  |
| --- |
| #include <iostream>  #include <Windows.h> //WindowsAPIを使うので、windows.hをインクルードする。  using namespace std;  //キーボードのAが押されているかどうかをbool型の  //値で返す関数。  bool IsPressA()  {  if (GetAsyncKeyState('A') != 0) {  //キーボードのAが押されている。  return true;  }  return false;  }  int main()  {  //ゲームループ。  while (true) {  bool result = IsPressA();  if (result == true) {  //キーボードのAが押された。  MessageBox(NULL, "Aが押された", "通知", MB\_OK);  }  }  return 0;  } |

7.4.3のプログラムは下記のようにも書ける。

## **7.4.4　サンプルプログラム(if文の中で関数呼び出し)**

|  |
| --- |
| #include <iostream>  #include <Windows.h> //WindowsAPIを使うので、windows.hをインクルードする。  using namespace std;  //キーボードのAが押されているかどうかをbool型の  //値で返す関数。  bool IsPressA()  {  if (GetAsyncKeyState('A') != 0) {  //キーボードのAが押されている。  return true;  }  return false;  }  int main()  {  //ゲームループ。  while (true) {  if (IsPressA()== true) { //if文の中で関数を呼び出す。IsPressAの戻り値がtrueなら  //キーボードのAが押された。  MessageBox(NULL, "Aが押された", "通知", MB\_OK);  }  }  return 0;  } |

7.4.4のプログラムは下記のようにも書けます。

## **7.4.5 サンプルプログラム(if文の中で関数の戻り値をそのまま使う)**

|  |
| --- |
| #include <iostream>  #include <Windows.h> //WindowsAPIを使うので、windows.hをインクルードする。  using namespace std;  //キーボードのAが押されているかどうかをbool型の  //値で返す関数。  bool IsPressA()  {  if (GetAsyncKeyState('A') != 0) {  //キーボードのAが押されている。  return true;  }  return false;  }  int main()  {  //ゲームループ。  while (true) {  if (IsPressA()) { //IsPressAの戻り値がtrueなら条件成立。  //キーボードのAが押された。  MessageBox(NULL, "Aが押された", "通知", MB\_OK);  }  }  return 0;  } |

if文の判定はどの書き方でも構いません。自分が分かりやすい書き方をしてください。ただし、他人が書いたコードを読むことはたくさんあるので、色々な書き方に慣れておきましょう。

## **7.4 Ex 関数名は分かりやすい名前を考えよう！**

　分かりやすいプログラムを書くことは非常に重要です。そして、分かりやすいプログラムを書くためには、分かりやすい名前を考えることが重要です。

分かりやすい名前を考えるときに、大きな指針となるのが、プロジェクトで統一された規則を守ることでしょう。ここでは、多くのプロジェクトで採用されているいくつかの命名規則を紹介します。

**動詞＋目的語とする**

例えばプレイヤーを検索する関数を作る場合、

FindPlayerといった関数名にします。Find(動詞)+Player(目的語)

敵を攻撃する関数を作る場合は、

AttackEnemyといった関数名にします。

**Upper camel case(アッパーキャメルケース)**

単語の先頭を大文字にするという命名規則です。FindPlayerやAttackEnemyなどです。

　また、Upper camel caseにはパスカルケースと呼ばれるものと、ローワーキャメルケースと呼ばれるものがあります。

　パスカルケース

　　→単語の先頭はすべて大文字。FindPlayer、AttackEnemyなど。

　　　→関数名、構造体名、クラス名などでよく使われる。

　ローワーキャメルケース

　　→最初の単語だけ小文字。findPlayer、attackEnemyなど。

　　　→変数名でよく使われる。

**判定を行う関数にはIs~を使う**

　これは日本語圏のプログラマーが良く使うもので、英語的には正しくないのですが、関数名の先頭にIsがついていると大抵bool値を返す関数です。

　Is this a pen?といった感じで、疑問文で使われるIsからきています。例えば、下記のようなプログラム。

|  |
| --- |
| if (IsPressA()) {//キーボードのAが押された?  //キーボードのAが押された。  MessageBox(NULL, "Aが押された", "通知", MB\_OK);  } |

## **Lesson\_07 ハンズオン 3**

サンプルプログラムのLesson\_07\_03/Questionを使用して、ハンズオン1～2を行いなさい。

## **Lesson\_07 実習課題\_3**

　サンプルプログラムのLesson\_07\_03/Questionを使用して、実習1～5を行いなさい。

サンプルプログラムのURL

<https://1drv.ms/u/s!AnpsoGtakIkVi-0viStE3apdAzUxXw>

## **Lesson\_07 中間テスト3**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSeAMKOEI_7SVdpkahmPo5DGLh8luvXFVAXn-cy1j7yejyZhag/viewform?usp=sf_link>

## **7.6 関数宣言**

関数宣言は「こんな関数がどこかにありますよ」、とコンパイラに教えるもの。

これまで作ってきたものは関数定義。関数定義は関数の本体。下記のコードを入力してください。max関数が定義されていませんというエラーが発生するはずです。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int a = 10;  int b = 20;  //max識別子が見つかりませんでした、というエラーが出る。  int c = max(a, b);  cout << "大きい値は" << c << "です。\n";  return 0;  }  //大きいほうの値を返す関数。  int max(int x, int y)  {  if (x > y) {  //xのほうが大きい。  return x;  }  //yのほうが大きい。  return y;  } |

コンパイラはコードを上から下に向かってコンパイルしていくため、下に何が書かれているか分かりません。そのため、このようなエラーが発生してしまいます。このようなときに関数宣言を使いましょう。

では、プログラムを下記のように書き換えてください。

|  |
| --- |
| #include <iostream>  using namespace std;  //これが関数宣言  int max( int x, int y );  int main()  {  int a = 10;  int b = 20;  //max識別子が見つかりませんでした、というエラーが出る。  int c = max(a, b);  cout << "大きい値は" << c << "です。\n";  return 0;  }  //大きいほうの値を返す関数。  int max(int x, int y)  {  if (x > y) {  //xのほうが大きい。  return x;  }  //yのほうが大きい。  return y;  } |

# **Lesson 11 いろいろな型**

## **11.3 構造体**

ユーザー定義型

　ユーザー(プログラマー)が自由に作成することができる型。C++Ⅱで勉強するenumやclassもユーザー定義型となる。

組み込み型

　int、float、char、boolなどは組み込み型。最初から言語に組み込まれているので組み込み型と呼ばれます。

**構造体の仕組みを知る**

構造体は異なる型の値をまとめるための型となっています(p339)

→なぜまとめるのか？

　　　→そのほうが綺麗でわかりやすいプログラムが書けるから。

　　　　→これの発展形がオブジェクト指向。

Sample2.cppを入力してください。

**構造体を初期化する**

波かっこの初期化は初期化リストと呼ばれるもので、C++のほぼすべての型の初期化で使える方法です。

int型の初期化

|  |
| --- |
| int hoge = {0}; //もちろんOK |

intの配列型の初期化。

|  |
| --- |
| int hoge[4] = { 4, 5, 6, 7 }; // |

構造体の配列の初期化。

|  |
| --- |
| //Car型の配列。  Car car[4] = {  1111, 10.0, 2222, 16.5, 3333, 8.4, 4444, 10.5  }; |

下記のようにも書ける。こちらのほうが理解しやすいコードになっていると思います。

|  |
| --- |
| //Car型の配列。  Car car[4] = {  { 1111, 10.0 }, //0番目  { 2222, 16.5 }, //1番目  { 3333, 8.4 }, //2番目  { 4444, 10.5} //3番目  }; |

下記のサンプルプログラムを入力してください。

|  |
| --- |
| #include <iostream>  using namespace std;  //車の構造体。  struct Car {  int number;  double gas;  };  int main()  {  //Car型の配列。  Car car[4] = {  { 1111, 10.0 }, //0番目  { 2222, 16.5 }, //1番目  { 3333, 8.4 }, //2番目  { 4444, 10.5} //3番目  };  //車の情報を表示する。  for (int i = 0; i < 4; i++) {  cout << i << "番目の車のナンバーは : " << car[i].number << "です。\n";  cout << i << "番目の車のガソリン量は : " << car[i].gas << "です。\n";  }  return 0;  } |

**構造体に代入する**

　Sample3.cppを入力してください。

構造体の値のコピーのルールは組み込み型(int、floatなど)の変数と同じです。

つまり、下記のようなコードを描いた場合でもcar1.gasの値は変わりません。

|  |
| --- |
| Car car1 = { 1234, 25.5 };  Car car2 = { 4567, 52.2 };  car2 = car1; //car2にcar1をコピー。  car2.gas = 10.0; //car2とcar1は別物なので、car1.gasの値は変わらない。 |

下記のプログラムを入力して、動作を確認してください。

|  |
| --- |
| #include <iostream>  using namespace std;  //車の構造体。  struct Car {  int number;  double gas;  };  int main()  {  Car car1 = { 1234, 25.5 };  Car car2 = { 4567, 52.2 };  car2 = car1; //car2にcar1をコピー。  car2.gas = 10.0; //car2とcar1は別物なので、car1.gasの値は変わらない。  cout << "car1.gasの値は" << car1.gas << "です。\n";  return 0;  } |

## **11.4 構造体の応用**

**引数として構造体を使う**

Sample4.cppを入力してください。

Sample4.cppの引数渡しは値渡しなので、下記のようなコードを書いた場合、おそらくプログラマーが意図した動作はしません。

|  |
| --- |
| #include <iostream>  using namespace std;  //車の構造体。  struct Car {  int mileage; //走行距離。マイレージ。)  int number; //ナンバー  double gas; //ガソリン量。  };  //車を動かす関数。  void MoveCar(Car car)  {  car.mileage += 20; //走行距離を20km増やす。  car.gas -= 1.5; //ガソリン量を減らす。  }  int main()  {  //走行距離0、ナンバー1234、ガソリン量25.5で初期化する。  Car car = { 0, 1234, 25.5 };  //MoveCar関数を使って車を動かす。  MoveCar(car);  //さて結果は？  cout << "走行距離は" << car.mileage << "です。\n";  cout << "ガソリン量は" << car.gas << "です。\n";  return 0;  } |

関数の中で変更した値を、呼び出し元の変数に反映させたい場合は参照(変数の別名)を使えばよかったのを思い出してください。先ほどのコードを下記のように書き換えてください。

書き換える箇所は網掛けになっている箇所だけです。

|  |
| --- |
| #include <iostream>  using namespace std;  //車の構造体。  ここだけ変更する！  struct Car {  int mileage; //走行距離。マイレージ。)  int number; //ナンバー  double gas; //ガソリン量。  };  //車を動かす関数。  void MoveCar(Car**&** car)  {  car.mileage += 20; //走行距離を20km増やす。  car.gas -= 1.5; //ガソリン量を減らす。  }  int main()  {  //走行距離0、ナンバー1234、ガソリン量25.5で初期化する。  Car car = { 0, 1234, 25.5 };  //MoveCar関数を使って車を動かす。  MoveCar(car);  //さて結果は？  cout << "走行距離は" << car.mileage << "です。\n";  cout << "ガソリン量は" << car.gas << "です。\n";  return 0;  } |

型名の後ろに&をつけると参照になります。

## **11.Ex 1 構造体の発展型としてのクラス**

　さて、ついにC++のオブジェクト指向の核となるクラスの勉強を行うための準備が整いました。C++はC言語のスーパーセットの言語となっています。ここまで勉強してきた内容は、どちらかというとC言語の機能です。C++は雑な説明をすると、C言語にオブジェクト指向の機能を追加した言語といえます。

　これも雑な説明になるのですが、クラスは構造体と関数をひとまとめにしたものと言えます。

では、11.4のサンプルプログラムをクラスを利用するコードに書き換えてみましょう。

|  |
| --- |
| #include <iostream>  using namespace std;  //Carクラスの宣言。  class Car {  public:  //これらをメンバ変数という。  int mileage; //走行距離。マイレージ。)  int number; //ナンバー  double gas; //ガソリン量。  //そしてこれをメンバ関数という。  void Move();  };  //CarクラスのMove関数の定義。  void Car::Move()  {  //クラスのメンバ関数では、この関数を呼び出した  //インスタンスのメンバ変数にアクセスできる！！！！  mileage += 20; //走行距離を20km増やす。  gas -= 1.5; //ガソリン量を減らす。  }  int main()  {  //走行距離0、ナンバー1234、ガソリン量25.5で初期化する。  Car car = { 0, 1234, 25.5 };  //Move関数を使って車を動かす。  car.Move();  //さて結果は？  cout << "走行距離は" << car.mileage << "です。\n";  cout << "ガソリン量は" << car.gas << "です。\n";  return 0;  } |

クラスの実体(変数などのこと)のことをインスタンスといいます。クラスのインスタンスは複数作成することができます。先ほどのプログラムを下記のように改造してみてください。

網掛けになっている箇所が追加されたコードです。

|  |
| --- |
| #include <iostream>  using namespace std;  //Carクラスの宣言。  class Car {  public:  //これらをメンバ変数という。  int mileage; //走行距離。マイレージ。)  int number; //ナンバー  double gas; //ガソリン量。  //そしてこれをメンバ関数という。  void Move();  };  //CarクラスのMove関数の定義。  void Car::Move()  {  //クラスのメンバ関数では、この関数を呼び出した  //インスタンスのメンバ変数にアクセスできる！！！！  mileage += 20; //走行距離を20km増やす。  gas -= 1.5; //ガソリン量を減らす。  }  int main()  {  //走行距離0、ナンバー1234、ガソリン量25.5で初期化する。  Car car = { 0, 1234, 25.5 };  //Move関数を使って車を動かす。  car.Move();  //さて結果は？  cout << "走行距離は" << car.mileage << "です。\n";  cout << "ガソリン量は" << car.gas << "です。\n";  **//2台目**  **Car car2 = { 0, 2222, 30.0 };**  **//Move関数を使って車を動かす。**  **car2.Move();**  **//さて結果は？**  **cout << "走行距離は" << car2.mileage << "です。\n";**  **cout << "ガソリン量は" << car2.gas << "です。\n";**  return 0;  } |

クラスは後期のC++Ⅱでさらに詳しく見ていきますので、ここでは紹介だけにとどめておきます。

サンプルプログラムLesson\_11\_01は下記のURLからダウンロードして、下記のハンズオンと実習を行いなさい。

<https://1drv.ms/u/s!AnpsoGtakIkVi_MYnn___HfLkZI4cQ>

## **Lesson 11 ハンズオン\_1**

サンプルプログラムLesson\_11\_01/Questionを使用して、ハンズオン1～4を行いなさい。

## **Lesson 11 実習課題\_1**

Lesson\_11\_01/Questionを使って、下記の動画のようなプレイヤーを追いかける敵キャラクターを実装しなさいmain.cppの実習ヒント①～⑤を参考に実装しなさい。ただし、ヒントに従わずに自由な方法で実装するのもOKです。

<https://youtu.be/e53lS03xo-A>

## **Lesson 11 実習課題\_2**

Lesson\_11\_01/Questionを使って、下記の動画のように４体敵キャラクターを表示して、プレイヤーを追いかけるようにしなさい。

<https://youtu.be/6EVFBGOi0ck>

## **Lesson 11 実習課題\_Ex\_1**

　　実習課題\_2のプログラムをPlayerクラスとEnemyクラスを利用するようにリファクタリングを行いなさい。

(ちょっと難しいという人は、実習課題\_ex\_1\_Answerを見てもOK、構造体を勉強した今だからこそ、クラスを使ったコードも見てこう！)

## **Lesson 11 中間テスト 1**

下記のURLのテストを行いなさい。

<https://goo.gl/forms/tDxlbEJstAF7ziG52>

# **Lesson 12 クラスの基本**

## **12.1 クラスの宣言**

**クラスの仕組みを知る**

クラスの機能を利用すると、複雑なプログラムを効率よく作成することができます(p.360)

**→**クラスは、オブジェクト単位で思考することをサポートすることができる概念です(オブジェクトに関する説明は後述します)。これは、人間はオブジェクトに着目することで、考えをまとめやすくなるという考えから生まれてきたもので、この考え方は**オブジェクト指向**と呼ばれます。

クラスを扱うときには、現実世界に存在する、「モノ」などの概念に着目していきます(p.360)

　→C++の入門書によく書かれている説明ですが、この説明のせいでオブジェクト指向の理解が進まないといわれているものです。(初学者に分かりやすい説明ではあると思いますが・・・)。例えば、下記は数学のベクトルをクラスにしたものです。

|  |
| --- |
| class Vector {  public:  //メンバ変数。  float x;  float y;  float z;  //ベクトルの長さを求めるメンバ関数。  float Length();  }; |

ベクトルは現実世界に存在する「モノ」ではありません。なので、p.360の説明は適切ではありません。しかし、ベクトルはまだギリギリ、「現実世界に存在するモノ」といえそうな気もします(かなりギリギリですが)。

では、さらに見てみましょう下記のクラスは、ゲームプログラミングの時間によく使っているCSkinModelRenderクラスです。このクラスは「**３Dモデルを描画する**」という機能をクラス化しています。これは現実世界に存在しませんし、モノですらありません。

|  |
| --- |
| class CSkinModelRender {  public:  CSkinModel 　m\_skinModel; //!<スキンモデル。  CSkinModelData m\_skinModelData; //!<スキンモデルデータ。  //3Dモデルを描画するメンバ関数。  //Render関数を呼び出すと絵が描ける。  void Render(CRenderContext& rc);  }; |

このように、**「クラスを扱うときには、プログラムで実装したい処理、機能に着目して考えるべきです。」**

　まだ、ピンとは来ないでしょうし、元々の教科書の説明の方が分かりやすいかもしれません。ですが、現実世界のモノに着目すると考えてしまうと、何をクラス化したらいいのか、分からなくなると思います。なぜなら、そもそもゲームというのは非現実なものなのですから。

**クラスを宣言する**

実は、クラスの書き方は、第11章で学んだ構造体型とほとんど同じです。(p362)

構造体を下記のように書いていたことを思い出してください。これとほとんど同じです。

|  |
| --- |
| //車の構造体の宣言。  struct Car {  int mileage; //走行距離。マイレージ。)  int number; //ナンバー  double gas; //ガソリン量。  };  int main()  {  //走行距離0、ナンバー1234、ガソリン量25.5で初期化する。  Car car = { 0, 1234, 25.5 };  //車の走行距離とガソリン量を表示する。  cout << "走行距離は" << car.mileage << "です。\n";  cout << "ガソリン量は" << car.gas << "です。\n";  return 0;  } |

ユーザー定義型

intやfloat、boolなどは組み込み型といわれるもので、最初から用意されている型です。

一方、構造体やクラスなどは、ユーザー(プログラマ－)が自由に新しく作成できる(定義できる)型なので、ユーザー定義型と呼ばれます。**構造体もクラスもただの型**だということをしっかりと押さえておいてください。

クラスを宣言するコード

クラスは構造体と異なって、関数を記述することができます。[[1]](#footnote-1)この関数のことをメンバ関数といいます。メンバ関数はそのクラスを操作するためのものです。

## **12.1.Ex\_1 ベクトルクラスを作る**

では、実際にクラスを作って、それを活用してみましょう。まずは、下記の手順に従って、Vectorクラスを作成してください。

①　VisualStudioを立ち上げて、新しいプロジェクトのLesson\_12を作成する。

② ソリューションエクスプローラーから新しいソースファイル、Source.cppを追加する。

③　ソリューションエクスプローラーから新しいクラスのVectorを追加する。

①と②は前期の復習です。③はゲームプログラミングの授業で前期からやってきています。下記の動画を見ながら思い出しつつ、実習してください。



## **12.1.Ex\_2 ベクトルクラスにx、y、zの値を記録するメンバ変数を追加する。**

ベクトルクラスの宣言にx、y、ｚのメンバ変数を追加しましょう。メンバ変数は構造体で勉強したメンバ変数と全く同じです。構造体を思い出しながら、コードを書いてください。

網掛けの部分が追加されたコードです。コンストラクタとデストラクタと呼ばれるものをVisualStudioが勝手に作ってくれていますが、今は気にしないでください。

Vector.h

|  |
| --- |
| class Vector  {  public:  //これはコンストラクタ。  //VisualStudioの機能を使うと  //勝手に作られているけど、今は気にしない。  Vector();  //これはデストラクタ。  //VisualStudioの機能を使うと  //勝手に作られているけど、今は気にしない。  ~Vector();  **float x; //ベクトルのxを表すメンバ変数。**  **float y; //ベクトルのyを表すメンバ変数。**  **float z; //ベクトルのzを表すメンバ変数。**  }; |

## **12.1.Ex\_3 ベクトルクラスにx、y、zの値を表示するDisplayメンバ関数を追加する。**

続いて、ベクトルクラスにDisplayメンバ関数を追加します。まずはクラス宣言にDisplayメンバ関数の宣言を追加しましょう。網掛けになっている箇所が追加されたコードです。

Vector.h

|  |
| --- |
| #pragma once  class Vector  {  public:  //これはコンストラクタ。  //VisualStudioの機能を使うと  //勝手に作られているけど、今は気にしない。  Vector();  //これはデストラクタ。  //VisualStudioの機能を使うと  //勝手に作られているけど、今は気にしない。  ~Vector();  **//x,y,zの値を表示するDisplay関数。**  **void Display();**  float x; //ベクトルのxを表すメンバ変数。  float y; //ベクトルのyを表すメンバ変数。  float z; //ベクトルのzを表すメンバ変数。  }; |

Vector.hに追加されたのは**関数宣言**です。関数の項目で勉強したことを思い出してください。関数には**宣言**と**定義**がありました。[#7.6 関数宣言](#_7.6_関数宣言)

関数は本体となる、関数定義を書く必要がありましたね。では、Vector.cppに関数定義を追加しましょう。網掛けになっている箇所が追加されたコードです。Diysplayメンバ関数の中でcout関数を使用しているため、に関係ないものも少し追加されているので、忘れないようにしてください。

Displayメンバ関数の中でcout関数を使っているので、これが必要。

Vector.cpp

|  |
| --- |
| **#include <iostream>**  #include "Vector.h"  **using namespace std;**  //これはコンストラクタ。  //VisualStudioの機能を使うと  //勝手に作られているけど、今は気にしない。  Vector::Vector()  {  }  //これはデストラクタ。  //VisualStudioの機能を使うと  //勝手に作られているけど、今は気にしない。  Vector::~Vector()  {  }  **//これがDisplayメンバ関数の定義！**  これがDisplayメンバ関数の定義  **void Vector::Display()**  **{**  **cout << "xの値は" << x << "です。\n";**  **cout << "yの値は" << y << "です。\n";**  **cout << "zの値は" << z << "です。\n";**  **}** |

コードを入力できたでしょうか？では、Displayメンバ関数について詳しく見ていきましょう。普通の関数とはいくつか違う点があります。

Vectorクラスのメンバ関数であることを表している。

|  |
| --- |
| //これがDisplayメンバ関数の定義！  void Vector::Display()  {  cout << "xの値は" << x << "です。\n";  cout << "yの値は" << y << "です。\n";  cout << "zの値は" << z << "です。\n";  }  メンバ変数のx、y、zにアクセスできる |

## **12.1.Ex\_4 ベクトルクラスを利用する。**

では、作成したベクトルクラスを利用して、プログラムを書いていきましょう。クラスは設計図のようなもので、そのクラスの実態の事を**インスタンス**または**オブジェクト**といいます。では、下記のコードを書いて、実行してみてください。

Source.cpp

メンバ関数の呼び出し方もメンバ変数と同じく、ドット(.)演算子を使う。関数呼び出しは()をつけるのを忘れない。

|  |
| --- |
| #include <iostream>  **#include "Vector.h"**  using namespace std;  メンバ変数のアクセスの仕方は構造体と同じ。ドット(.)演算子を使ってアクセス。  int main()  {  **Vector va;**  **//vaの各要素に値を設定していく。**  **va.x = 100.0f;**  **va.y = 20.0f;**  **va.z = 30.0f;**  **//Displayメンバ関数を呼び出して、vaの要素の値を表示する。**  **va.Display();**  return 0;  } |

このコードでは、vaはVectorクラスのインスタンスになります。つまり、ただの変数もインスタンスです。

## **12.1.Ex\_5 インスタンスを複数作る**

クラスのインスタンスは複数作ることができます。下記のコードを書いて、複数のベクトルを用意してください。

|  |
| --- |
| #include <iostream>  #include "Vector.h"  using namespace std;  int main()  {  Vector va;  //vaの各要素に値を設定していく。  va.x = 100.0f;  va.y = 20.0f;  va.z = 30.0f;  //Displayメンバ関数を呼び出して、vaの要素の値を表示する。  va.Display();  **//ベクトルvbを用意。**  **Vector vb;**  **vb.x = 50.0f;**  **vb.y = 30.0f;**  **vb.z = 20.0f;**  **//Displayメンバ関数を呼び出して、vbの要素の値を表示する。**  **vb.Display();**  return 0;  } |

## **12.1.Ex\_6 ベクトルの長さを求めてみる。**

　ベクトルvの長さは下記のように3平方の定理を使うことで求めることができます。

また、平方根(√)はc言語の標準関数のsqrt( float x )を使うことで求めることができます。

下記はベクトルvの長さを求めるサンプルコードです。

|  |
| --- |
| Vector v;  v.x = 10.0f;  v.y = 20.0f;  v.z = 30.0f;  //v.x^2 + v.y^2 + v.z^2の計算結果を  //変数ｔに代入。  float t = v.x \* v.x + v.y \* v.y + v.z \* v.z;  //上で求めた t をsqrt関数の引数として渡して、ベクトルの長さを求める。  float length = sqrt( t ); |

では、Lesson12を立ち上げて、次のサンプルコードを入力してください。

|  |
| --- |
| #include <iostream>  #include "Vector.h"  using namespace std;  int main()  {  Vector v;  v.x = 3.0f;  v.y = 4.0f;  v.z = 5.0f;  //ベクトルｖの長さを求める。  //① ベクトルの各要素の２乗の総和を求める。  float t = v.x \* v.x + v.y \* v.y + v.z \* v.z;  //②　①で求めた値の平方根を求める。  float nagasa = sqrt(t);  cout << "ベクトルｖの長さは" << nagasa << "です。\n";  return 0;  } |

## **12.1.Ex\_7 ベクトルの長さを求めるメンバ関数を作成する。**

　12.1.Ex\_6でベクトルの長さを求めるプログラムを書いてみました。この処理はゲームにおいて非常によく使うプログラムなのですが、距離を求めるために毎回、12.1.Ex\_6のようなコードを書いていると、とても面倒になってきます。そこで、ベクトルの長さを求めるメンバ関数を作成して、プログラムを再利用できるようにしてみましょう。

　まず、Vectorクラスにメンバ関数の宣言を追加します。Vector.hを開いて、網掛けになっているコードを追加してください。

Vector.h

|  |
| --- |
| #pragma once  class Vector  {  public:  Vector();  ~Vector();    //ベクトルの各要素を表示する関数。  void Display();  **//ベクトルの長さを求める関数。**  **//戻り値　ベクトルの長さ。**  **float Length();**  float x;  float y;  float z;  }; |

VectorクラスにLengthメンバ関数の宣言を追加することができたら、次は、関数定義を作成しましょう。Vector.cppを開いてください。

Vector.cpp

|  |
| --- |
| float Vector::Length()  {  //①　各要素の２乗の総和を求める。  float t = x \* x + y \* y + z \* z;  //②　①で求めた値の平方根を求める。  float length = sqrt(t);  //③　長さが求まったので、長さを返す。  return length;  } |

では、作成したLengthメンバ関数を使用してみましょう。main.cppに下記のコードを入力してください。

main.cpp

|  |
| --- |
| #include <iostream>  #include "Vector.h"  using namespace std;  int main()  {  //ベクトルvaを定義。  Vector va;  //vaの各要素に値を代入する。  va.x = 2.2f;  va.y = 4.2f;  va.z = 5.6f;  //ベクトルvaの長さを計算して、tに代入する。  float t = va.Length();  cout << "vaの長さは" << t << "です。\n";  //ベクトルvbを定義。  Vector vb;  //vbの各要素に値を代入する。  vb.x = -3.5f;  vb.y = -10.0f;  vb.z = 2.0f;  //ベクトルvbの長さを計算してtに代入する。  t = vb.Length();  cout << "vbの長さは" << t << "です。\n";  return 0;  } |

## **12.1.Ex\_8 ベクトルとスカラーの掛け算**

　ベクトルはスカラーを乗算することができます。そして、例えばベクトルに２を乗算すると、そのベクトルの長さは２倍になります。

　では、Lesson12を立ち上げて、main.cppに次のコードを入力してください。

|  |
| --- |
| #include <iostream>  #include "Vector.h"  using namespace std;  int main()  {  Vector v;  v.x = 1.0f;  v.y = 2.0f;  v.z = 4.0f;  float len = v.Length();  cout << "vの長さは" << len << "です。\n";  //ベクトルｖを3倍にする。  v.x \*= 3.0f;  v.y \*= 3.0f;  v.z \*= 3.0f;  //ベクトルvの長さを求める。  len = v.Length();  cout << "vの長さは" << len << "です。\n";  return 0;  } |

入力ができた人は、プログラムを実行してみて、ベクトルの長さが３倍になっていることを確認してください。

## **12.1\_Ex\_9 ベクトルとスカラーの掛け算を行うメンバ関数を追加。**

ベクトルとスカラーの掛け算も、ゲームでは頻繁に行われる処理です。なので、長さを求める処理と同様に、この処理もメンバ関数にしてみましょう。まず、Vector.hを開いて、関数宣言を追加しましょう。下記の網掛けになっているコードを追加してください。

Vector.h

|  |
| --- |
| #pragma once  class Vector  {  public:  Vector();  ~Vector();  　　　　︙  　　　　省略  　　　　︙  **//ベクトルとスカラーの掛け算を行う。**  **//引数　s ベクトルに乗算するスカラー。**  **void Multiply( float s );**  float x;  float y;  float z;  }; |

Multiplyは掛け算という意味をもつ英語です。

では、続いて、関数定義を追加しましょう。Vector.cppに下記の網掛けになっているコードを追加してください。

Vector.cpp

|  |
| --- |
| #include "Vector.h"  #include <iostream>  using namespace std;  　　　　︙  　　　　省略  　　　　︙  **void Vector::Multiply(float s)**  **{**  **x \*= s;**  **y \*= s;**  **z \*= s;**  **}** |

　では、追加したMultiplyメンバ関数を使ってみましょう。main.cppに次のコードを入力してください。

main.cpp

|  |
| --- |
| #include <iostream>  #include "Vector.h"  using namespace std;  int main()  {  Vector v;  v.x = 3.2f;  v.y = -5.0f;  v.z = -2.3f;  //vの長さを確認する。  cout << "vの長さは" << v.Length() << "です。\n";  ここでvに10をかけている。    //vに10を乗算する。  v.Multiply( 10.0f );  //vの長さが10倍になっているか確認する。  cout << "vの長さは" << v.Length() << "です。\n";  return 0;  } |

## **12.1\_Ex\_10 実習課題　ベクトルをスカラーで除算(割り算)するメンバ関数を追加**

**ステップ１**　Vectorクラス宣言に、割り算を行うメンバ関数の宣言を追加する。

関数宣言は、戻り値なし、関数名Div、引数にスカラーを渡せるように、下記のように宣言してください。

**void Div( float s );**

**ステップ２**　ステップ１で追加した、Div関数の定義を追加してください。Div関数では、引数で渡

されたスカラーを使ってベクトルの各要素を、除算してください。

**ステップ３**　下記のコードをmain.cppに入力して、追加した除算を行うメンバ関数が正しく動いていることを確認してください。

main.cpp

|  |
| --- |
| #include <iostream>  #include "Vector.h"  using namespace std;  int main()  {  Vector v;  v.x = -2.2f;  v.y = 10.0f;  v.z = 10.3f;  //vの長さを確認する。  cout << "vの長さは" << v.Length() << "です。\n";    //vを10で割る。  v.Div( 10.0f );  //vの長さが1/10になっているか確認する。  cout << "vの長さは" << v.Length() << "です。\n";  return 0;  } |

下記のように表示されたら、できています。



## **12.Ex\_1 クラスと配列**

　クラスはユーザー定義型であり、そのintやfloatなどの組み込み型と同じ型であると繰り返し説明をしています。例えば、下記のようなコードを記述した場合、「vはVector型の変数である」と言えます。

|  |
| --- |
| int main()  {  **Vector v;**  　　　　︙  　　　　省略  　　　　︙  return 0;  } |

vはユーザー定義型の変数ですが、intやfloatなどの組み込み型の変数と同様に扱うことができるので、下記のように、クラスの配列変数を定義することができます。

|  |
| --- |
| #include <iostream>  #include "Vector.h"  using namespace std;  int main()  {  **//10個のベクトル変数が定義されていることと同じ！！！**  **Vector v[10];**  **︙**  **省略**  **︙**  return 0;  } |

では、下記のコードをmain.cppに入力して、動作を確認してみてください。このお題はfor文と配列の復習も兼ねています。しっかり考えながらコードを入力しましょう。また、今回はC++11で追加された、非決定的な乱数生成機のrandom\_deviceクラスを使っています。難しそうな言葉を使っていますが、要はランダムな数を作ることができる便利な奴です。

main.cpp

random\_deviceの機能を使いたいので、randomをインクルードしていることに注意！！！

|  |
| --- |
| #include <iostream>  #include "Vector.h"  #include <random>  using namespace std;  int main()  {  //非決定的な乱数生成機。  //ベクトルの各要素をランダムに決定するときに使うのさ。  random\_device rd;  //要素数3のベクトル型の配列を定義する。  Vector v[3];  for ( int i = 0; i < 3; i++ ) {  //ベクトルの各要素を乱数生成機を使ってランダムに決定する。  //warningでるけど、気にしない。  //rd()はランダムな整数を返してきます。  //そして、ランダムに生成した整数を50で割った余りをベクトルの各要素に  //代入しているので、ベクトルの各要素は0～49までの値となります。  v[i].x = rd() % 50;  v[i].y = rd() % 50;  ここで、random\_deviceのインスタンスのrdを使っている。実はこれはメンバ関数呼び出し。  v[i].z = rd() % 50;  //vの各要素と大きさを表示する。  cout << "v[" << i << "]" << "の値は？\n";  v[i].Display();  　　　　　 cout << "大きさは、" << v[i].Length() << "です\n";  cout << "\n";  }  return 0;  } |

　入力が出来たら実行してみてください。下記のように表示されたら完成です。表示されるベクトルの各要素(x,y,z)の値は、乱数を使っているので、毎回変わるので注意してください。



## **12.Ex\_1.1　何番目のベクトルが最も大きいか調べる**

　では、先ほどのプログラムを改造して、何番目のベクトルが最も大きいか調べるプログラムを書いてみましょう。下記の網掛けになっているコードをmain.cppに追加してください。　これはif文の復習も兼ねています。しっかりと考えながらコードを入力してください。

|  |
| --- |
| #include <iostream>  #include "Vector.h"  #include <random>  using namespace std;  int main()  {  //非決定的な乱数生成機。  //ベクトルの各要素をランダムに決定するときに使うのさ。  random\_device rd;  //要素数3のベクトル型の配列を定義する。  Vector v[3];    for ( int i = 0; i < 3; i++ ) {  //ベクトルの各要素を乱数生成機を使ってランダムに決定する。  //warningでるけど、気にしない。  //rd()はランダムな整数を返してきます。  //そして、ランダムに生成した整数を50で割った余りをベクトルの各要素に  //代入しているので、ベクトルの各要素は0～49までの値となります。  v[i].x = rd() % 50;  v[i].y = rd() % 50;  v[i].z = rd() % 50;    //vの各要素と大きさを表示する。  cout << "v[" << i << "]" << "の値は？\n";  v[i].Display();  cout << "大きさは、" << v[i].Length() << "です\n";  cout << "\n";  }  **float maxLen = 0.0f; //最も小さい距離を記録するための変数。**  **float maxVectorNo = 0; //最も大きいベクトルの番号を記録するための変数。**  **//一番大きいベクトルを調べるぞい。**  **for ( int i = 0; i < 3; i++ ) {**  **//i番目のベクトルの長さを計算する。**  **float len = v[i].Length();**  **if ( maxLen < len ) {**  **//このベクトルの方が大きいので、**  **//maxLenとmaxVectorNoを上書きする。**  **maxLen = len;**  **maxVectorNo = i;**  **}**  **}**  **cout << "最も大きいベクトルは" << maxVectorNo << "番目です。\n";**  return 0;  } |

## **12.Ex\_1.2 実習課題　何番目のベクトルが最も小さいか調べる**

　下記のプログラムの空欄aに入るプログラムを考えて、最も小さいベクトルの番号も表示できるようにしなさい。なお、このプログラムは12.Ex\_1.1のプログラムを改造したものである。

main.cpp

|  |
| --- |
| #include <iostream>  #include "Vector.h"  この3行は12.Ex\_1.1のプログラムに追加されたものです。自分で追加してください。  #include <random>  using namespace std;  int main()  {  **︙**  **省略**  **︙**  cout << "最も大きいベクトルは" << maxVectorNo << "番目です。\n";  **float minLen = FLT\_MAX; //最も小さい距離を記録するための変数。**  **float minVectorNo = 0; //最も小さいベクトルの番号を記録するための変数。**  **a**  //一番小さいベクトルを調べるぞい。  for (int i = 0; i < 3; i++) {  //i番目のベクトルの長さを計算する。  float len = v[i].Length();  if (minLen > len) {  //このベクトルの方が小さいので、  //minLenとminVectorNoを上書きする。  minLen = len;  minVectorNo = i;  }  }  **cout << "最も小さいベクトルは" << minVectorNo << "番目です。\n";**  return 0;  } |

下記のように表示されたら完成です。



## **12.Ex\_2 クラスと関数**

　クラスはユーザー定義型であり、そのintやfloatなどの組み込み型と同じ型であると繰り返し説明をしています。例えば、下記のようなコードを記述した場合、「vはVector型の変数である」と言えます。

|  |
| --- |
| int main()  {  **Vector v;**  　　　　︙  　　　　省略  　　　　︙  return 0;  } |

vはユーザー定義型の変数ですが、intやfloatなどの組み込み型の変数と同様に扱うことができます。なので、組み込みがたの変数と同様に関数の引数として渡すことができます。

　では、Lesson12を立ち上げて、ベクトル同士の足し算を行う関数を作成してみましょう。下記のコードを入力してみてください。

main.cpp

|  |
| --- |
| #include "stdafx.h"  #include <iostream>  #include "Vector.h"  using namespace std;  // 概要 二つのベクトルを加算する。  // 引数 lhs 左辺値  // 引数 rhs 右辺値  // 戻り値 lhs + rhsの結果。  Vector Add(Vector lhs, Vector rhs)  {  Vector kekka;  //ベクトルの各要素を足し算。  これが足し算を行う関数。  kekka.x = lhs.x + rhs.x;  kekka.y = lhs.y + rhs.y;  kekka.z = lhs.z + rhs.z;    //足し算をした結果を戻す。  return kekka;  }  int main()  {  Vector v1, v2;  v1.x = 10.0f;  メイン関数でAdd関数を使っている。  v1.y = 2.3f;  v1.z = 5.0f;  v2.x = 3.0f;  v2.y = 12.1f;  v2.z = -5.4f;  Vector v3;  v3 = Add(v1, v2); //v1とv2を足し算  v3.Display(); //足し算した結果を表示。  return 0;  } |

## **12.Ex\_2.1　演算子のオーバーロード**

　ベクトルのような、数学系のクラスだと、足し算を行うときに下記のように記述できるとコードが読みやすくなります。

|  |
| --- |
| int main()  {  Vector v1, v2;  v1.x = 10.0f;  ＋を使って足し算をする。  v1.y = 2.3f;  v1.z = 5.0f;  v2.x = 3.0f;  v2.y = 12.1f;  v2.z = -5.4f;  Vector v3;  **v3 = v1 + v2;** //v1とv2を足し算  v3.Display(); //足し算した結果を表示。  return 0;  } |

　数学の書籍に乗っている計算式なども、このように四則演算を使って表記されているため、こちらの方が、書籍の計算式をそのまま写せば良くなるので、簡単です。

　C++には、このようなことを実現するために、**演算子のオーバーロード**という機能があります。では、12.Ex\_2で作成したプログラムを下記のように書き換えてください。

|  |
| --- |
| #include "stdafx.h"  #include <iostream>  #include "Vector.h"  using namespace std;  これが演算子のオーバーロード。左辺と右辺にVectorクラスの変数を受け取って+演算子を使う場合は、この処理が呼ばれるようになる。  // 概要 二つのベクトルを加算する。  // 引数 lhs 左辺値  // 引数 rhs 右辺値  // 戻り値 lhs + rhsの結果。  **Vector operator+(Vector lhs, Vector rhs)**  {  Vector kekka;  //ベクトルの各要素を足し算。  kekka.x = lhs.x + rhs.x;  kekka.y = lhs.y + rhs.y;  kekka.z = lhs.z + rhs.z;  //足し算をした結果を戻す。  return kekka;  }  int main()  {  こんな感じで使える。  Vector v1, v2;  v1.x = 10.0f;  v1.y = 2.3f;  v1.z = 5.0f;  v2.x = 3.0f;  v2.y = 12.1f;  v2.z = -5.4f;  Vector v3;  **v3 = v1 + v2;** //v1とv2を足し算  v3.Display(); //足し算した結果を表示。    return 0;  } |

## **12.Ex\_2.2 実習課題**

**問１　ベクトルの引き算関数、Subtractを実装する。**

**ステップ１**　下記のプログラムを入力してください。

|  |
| --- |
| #include "stdafx.h"  #include <iostream>  #include "Vector.h"  using namespace std;  int main()  {  Vector v1, v2;  ここでエラーが出るけど気にしない。  v1.x = 10.0f;  v1.y = 2.3f;  v1.z = 5.0f;  v2.x = 3.0f;  v2.y = 12.1f;  v2.z = -5.4f;  Vector v3;  **v3 = Subtract( v1, v2 );** //v1-v2を計算する。  v3.Display(); 　　　　　//引き算した結果を表示。    return 0;  } |

**ステップ２**　ステップ１で作成したプログラムにベクトルの引き算を行う、Subtract関数

を追加してください。Subtract関数は下記のような関数です。

Vector Subtract(Vector lhs, Vector rhs)

　　　　　　　ベクトルの引き算の公式は下記のようになります。

　　　　　　　 [ x1 y1 z1 ] – [ x2 y2 z2 ] = [ x1－x2 y1－y2 z1－z2 ]

　　　　　　　計算例

　　　　　　　 [ 2 3 4 ]－[ 1 5 6 ] = [ 1 －２ －２ ]

　　　　　　　 完成したら実行してください。下記のように表示されたら正解です。



**問２　問１で作成した、Subtract関数を－演算子のオーバーロードに変更してください。**

## **12.2 実習課題　ベクトルクラスに色々なメンバ関数を追加してみよう。**

問１　Vectorクラスにベクトルを正規化するNormalizeメンバ関数を追加する。

**ステップ１**　下記のプログラムを入力してください。

|  |
| --- |
| #include "stdafx.h"  #include <iostream>  #include "Vector.h"  using namespace std;  int main()  {  Vector v;  ここでエラーが出るけど気にしない。  v.x = 10.0f;  v.y = 2.3f;  v.z = 5.0f;    //vを正規化する。  　　v.Normalize();  v.Display();      return 0;  } |

**ステップ2** Vector.hのVectorクラス宣言にNormalize関数の宣言を追加する。

　　　　　　 Normalize関数は下記のように宣言してください。

　　　　　　　void Normalize();

**ステップ3** Vector.cppにNormalize関数の定義を作成する。ベクトルの正規化は

　　　　　　　ベクトルの長さを求めて、その長さで各要素を除算することで行えます。

下記のように表示されたら完成です。



問2 Vectorクラスに2点間の距離を計算する、DistanceFrom関数を追加する。

**ステップ１**　下記のプログラムを入力してください。

|  |
| --- |
| #include "stdafx.h"  #include <iostream>  #include "Vector.h"  using namespace std;  int main()  {  Vector p1;  p1.x = 10.0f;  p1.y = 2.3f;  p1.z = 5.0f;  Vector p2;  p2.x = 20.0f;  p2.y = 6.5f;  p2.z = -3.5f;  //DistanceFromメンバ関数を使って、  //p1とp2の距離を調べる。  float dist = p1.DistanceFrom(p2);  cout << "p1とp2の距離は" << dist << "です。\n";  return 0;  } |

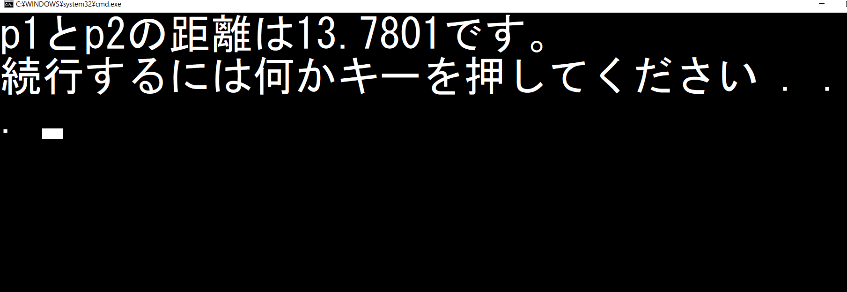
ステップ２　Vector.hのVectorクラス宣言にDistanceFrom関数の宣言を追加する。

　　　　　　DistanceFrom関数は下記のように宣言してください。

　　　　　　float DistanceFrom( Vector v );

ステップ３　Vector.cppにDistanceFrom関数の定義を作成する。

　　　　　　下記のように表示されたら完成です。



問3 Vectorクラスにゼロベクトルかどうかの判定を行うIsZero関数を追加しなさい。

ステップ 1 下記のプログラムを入力しなさい。

|  |
| --- |
| #include "stdafx.h"  #include <iostream>  #include "Vector.h"  using namespace std;  int main()  {  Vector v;  v.x = 0.0f;  v.y = 0.0f;  v.z = 0.0f;  if (v.IsZero() == true) {  std::cout << "vはゼロベクトルです。\n";  }  v.x = 0.1f;  if (v.IsZero() == false) {  std::cout << "vはゼロベクトルではありません。\n";  }  return 0;  } |

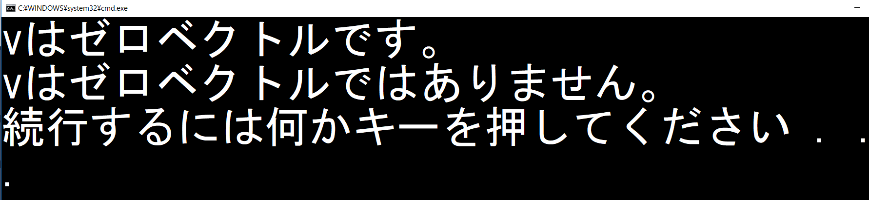
ステップ２　Vectorクラス宣言にIsZero関数の宣言を追加する。

　　　　　　IsZero関数は下記のように宣言してください。

　　　　　　bool IsZero();

ステップ3 Vector.cppにIsZero関数の定義を作成する。

　　　　　　下記のように表示されたら完成です。



問4 Vectorクラスにほぼゼロベクトルかどうかの判定を行うIsFuzzyZero関数を追加しなさい。

ステップ1 下記のプログラムを入力してください。

|  |
| --- |
| #include "stdafx.h"  #include <iostream>  #include "Vector.h"  using namespace std;  int main()  {  Vector v;  v.x = 1.0f;  v.y = 0.0f;  v.z = 0.0f;  for (int i = 0; i < 10; i++) {  v.x -= 0.1f;  }  //v.xは1.0から0.1を10回減算しているので、0になっているはず？？？  if (v.IsZero() == true) {  std::cout << "vはゼロベクトルです。\n";  }  else {  std::cout << "vはゼロベクトルではない。\n";  }  //浮動小数点の計算誤差で、0にはなっていない。  //浮動小数点は近似で判定を行うべきである。  if (v.IsFuzzyZero() == true) {  std::cout << "vはほぼゼロベクトルです。\n";  }  return 0;  } |

ステップ 2 Vectorクラス宣言にIsFuzzyZero関数の宣言を追加する。

　　　　　　IsFuzzyZero関数は下記のように宣言してください。

　　　　　　bool IsFuzzyZero();

ステップ3 Vector.cppにIsFuzzyZero関数の定義を作成する。

　　　　　　下記のように表示されたら完成です。



問5 Vectorクラスに引数で渡されたベクトルと値が等しいかを調べるEqual関数を追加しなさい。

ステップ1 下記のプログラムを入力してください。

|  |
| --- |
| #include "stdafx.h"  #include <iostream>  #include "Vector.h"  using namespace std;  int main()  {  Vector p1;  p1.x = 10.0f;  p1.y = 2.3f;  p1.z = 5.0f;  Vector p2;  p2.x = 10.0f;  p2.y = 2.3f;  p2.z = 5.0f;  //p1とp2が等しいか判定。  if (p1.Equal(p2) == true) {  cout << "p1とp2は同じ座標です。\n";  }  else {  cout << "p1とp2は違う座標です。\n";  }    //p2の値をp1と違う値に変更するとどうなる？  p2.x = 9.0f;  if (p1.Equal(p2) == true) {  cout << "p1とp2は同じ座標です。\n";  }  else {  cout << "p1とp2は違う座標です。\n";  }  return 0;  } |

ステップ２　Vectorクラス宣言にEqual関数の宣言を追加する。

　　　　　　Equal関数は下記のように宣言してください。

　　　　　　bool Equal( Vector v );

ステップ3　Vector.cppにEqual関数の定義を作成する。

　　　　　　下記のように表示されたら完成です。



問6 Vectorクラスに引数で渡されたベクトルと値がほぼ同じかを調べるNearlyEqual関数を追加する。

ステップ1 下記のプログラムを入力してください。

|  |
| --- |
| using namespace std;  int main()  {  Vector p1;  p1.x = 10.0f;  p1.y = 2.3f;  p1.z = 5.0f;  Vector p2;  p2.x = 0.0f;  p2.y = 2.3f; //yはp1.yと同じ値を代入。  p2.z = 5.0f; //zはp1.zと同じ値を代入。    //p2.xに10回、1.0を加算する。  for (int i = 0; i < 10; i++) {  p2.x += 1.0f;  }  //p2.xは1.0を10回加算しているから、10.0fになっているはずなのに  //同じ座標にならない！！！！  //p1とp1が等しいか判定する。  if (p1.Equal(p2) == true) {  cout << "p1とp2は同じ座標です。\n";  }  else {  cout << "p1とp2は違う座標です。\n";  }  //浮動小数点の計算は誤差が生まれるので、  //10.0fにはなっていない！！！！  //浮動小数点の比較は近似で行うのがセオリーです。  if (p1.NearlyEqual(p2) == true) {  cout << "p1とp2はほぼ同じ座標です。\n";  }  else {  cout << "p1とp2は違う座標です。\n";  }  return 0;  } |

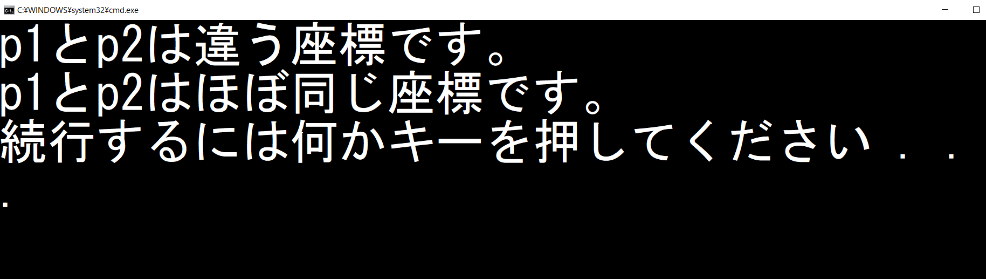
ステップ２　Vectorクラス宣言にNearlyEqual関数の宣言を追加する。

　　　　　　NearlyEqual Equal関数は下記のように宣言してください。

　　　　　　bool NearlyEqual( Vector v );

ステップ3　Vector.cppにNearlyEqual関数の定義を作成する。

　　　　　　下記のように表示されたら完成です。



ヒント１　2点間の距離を調べて、その距離が一定値以下(これは勝手に決めてください。0.001fくらいでいいと思います。)なら、ほぼ同じだと判定すればよい。

# **Lesson 8 ポインタ**

## **8.1　アドレス**

**アドレスの仕組みを知る(p.222)**

数字や文字列などを記録できる変数は、すべてメモリ上に領域が確保されていて、すべての変数に必ず、メモリ上の住所となるアドレスが存在します。

**変数のアドレスをみる(p223)**

VisualStudioで新しいプロジェクトLesson8を作成して、Sample1.cppを入力して、動作を確認してください。

## **8.2 ポインタ**

**ポインタの仕組みを知る(p225)**

型名の後ろに＊を書くと、その型はポインタ型となる。

**int** hoge; //hogeはint型の変数。Int方は整数値を記録できる。

**int\*** pHoge; //pHogeはintのポインタ型の変数。Intのポインタ型はint型の変数のアド//レスを記録できる。

**変数を定義するときは、必ずその変数の型を指定する！**

Lesson8にSample2.cppを記入して、動作を確認してください。

pAは変数aをさす

　ポインタはメモ用紙のようなもので、変数のアドレスを記録することができます。

メモ帳に電話番号を記録するようなイメージでもＯＫです。なので、そのメモ用紙を使って、そのアドレスに対してアクセスすることができます。

**ポインタから変数の値を知る(p228)**

ポインタの前に＊をつけることで、そのポインタに記録されているアドレスにアクセスすることができます。Lesson8にSample3.cppを記述して、動作を確認してください。

**ポインタについて整理する(p230)**

**ポインタに別のアドレスを代入する(p232)**

ポインタはアドレスを記録できるメモ帳のようなものなので、消しゴムでもとのアドレスを消して、新しいアドレスを上書きすることができます。

　Lesson8にSample4.cppを入力して、動作を確認してください。

**ポインタに値を代入しないと？(p234)**

ポインタはアドレス(数字)が入っているだけの変数です。本質的にはint型の変数と変わりはありません。ですので、値を代入してやらないと、どんなアドレス(数字)が入っているか分かりません。

VisualStudioのプロジェクトのLesson8に下記のコードを入力してみてください。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int a = 10; //変数aを定義。メモリ上に10が記録される。  int\* pa; //ポインタpaを定義。  cout << "paに記録されているアドレスは？" << pa << "\n";  cout << "paが指している場所の値は?" << \*pa << "\n";  } |

下記のようなコンパイルエラーが起きて実行できなかったのではないでしょうか？



**ポインタを使って変数を変更する(p.235)**

VisualStudioのプロジェクトのLesson8にSample5.cppを入力してください。

## **8.2.1 実習　いろいろな型を使って、ポインタを使ってみよう。**

　下記のコードはint型のポインタを使っています。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int a = 10; //変数aを定義。メモリ上に10が記録される。  int\* pa = &a; //ポインタpaを定義。  \*pa += 5;  cout << "paに記録されているアドレスは？" << pa << "\n";  cout << "paが指している場所の値は?" << \*pa << "\n";  } |

int 型のポインタは、int型の変数のアドレスを覚えることができます。では、下記の実習を行ってください。

①　変数aの型をfloat型(32ビット浮動小数点型)に変更して、動作するようにする。

②　変数aの型をdouble型(64ビット浮動小数点型)に変更して、動作するようにする。

③　変数aの型をshort型(16bit整数型)に変更して、動作するようにする。

## **8.ex\_1 構造体型の変数とポインタ**

　構造体はユーザー定義型と呼ばれるもので、intやfloatなどとと同じく型です。なので、当然構造体の変数にもアドレスがあります。

# **Lesson 9 配列　応用**

## **9.4 配列の応用**

キーボードから入力する(p.265)

**修飾子**

修飾子とは関数、変数、クラスなどの性質を決めるキーワードです。const修飾子は関数、変数などに定数性(変化しない)という性質を付加します。ゲームの属性、エンチャントみたいなものだと考えると分かりやすいかものではないでしょうか。

　例えば、変数に**初期化以降は値を変更できない**という属性を付加したければ、下記のように変数定義のところにconstを付与します。

|  |
| --- |
| const int hoge = 5; |

constが付与された変数は、**定数**となります。VisualStudioのプロジェクトのLesson\_9を立ち上げて、下記のプログラム\_1を入力して、コンパイルエラーが出るのを確認してください。

**プログラム\_1**

|  |
| --- |
| #include <iostream>  using namespace std;  int main(void)  {  const int hoge = 10;  //値を変えることはできないので、コンパイルエラーが発生する。  hoge = 5;  return 0;  } |

**どのような場面で使うのか？**

constは値を変更されたくない変数などに付与すべきです。例えば、数学の円周率を表す3.14159265359などの数値は不変のものです。このような場合に、下記のような定数を用意すると便利です。

|  |
| --- |
| const float PI = 3.14159265359f; |

定数を用意していると、円周率を扱う次のようなプログラム\_2をプログラム\_3のようにすっきりと書くことができます。

**プログラム\_2(定数を未使用)**

|  |
| --- |
| //円の面積と長さを計算するプログラム  #include <stdio.h>  #include <math.h>  int main(void)  {  //円の半径  定数を使用しない場合は、円周率の数値をすべての箇所で間違えないように注意する必要がある。**ヒューマンエラーが起きやすい！**  float radius;  //円の面積  float area;  //円周の長さ  float length;  //球の半径を入力。  printf("円の半径を入力: r = ");  scanf\_s("%f", &radius);  //円の面積を計算・出力  area = 3.14159265359f \* radius \* radius;  printf("円の面積: S = %.3f\n", area);  //円周の長さを計算・出力  length = 2 \* 3.14159265359f \* radius;  printf("円周の長さ: L = %.3f\n", length);  return 0;  } |

**プログラム3(定数を使用)**

|  |
| --- |
| //円の面積と長さを計算するプログラム  #include <stdio.h>  #include <math.h>  int main(void)  {  定数を使用する場合は、円周率の数値は一か所でだけ指定すればよいので、**ヒューマンエラーが起きにくい！**  //円周率  **const float PI = 3.14159265359f;**  //円の半径  float radius;  //円の面積  float area;  円周率を使う場所では、PIを使えばよい。  //円周の長さ  float length;  //球の半径を入力。  printf("円の半径を入力: r = ");  scanf\_s("%f", &radius);  //円の面積を計算・出力  area = **PI** \* radius \* radius;  printf("円の面積: S = %.3f\n", area);  //円周の長さを計算・出力  length = 2 \* **PI** \* radius;  printf("円周の長さ: L = %.3f\n", length);  return 0;  } |

このように、ソースコードに数値を書き込んでいるプログラムよりも、不変な数値は定数として定義して、それを利用するようにすると、プログラムの修正コストや間違いを大幅に減らすことができます。

**定数を配列の要素数として使う**

定数は配列の要素数として使うことができます。Lesson9にP265のSample2.cppを入力してください。

Sample2.cppは配列の要素数の指定に定数を使っているので、配列のサイズを簡単に変更することができます。もしもSample2.cppがプログラム\_4のような、直接値を書き込んでいた場合に配列の要素数を変更しようとするとどうなるでしょうか？

**プログラム\_4**

|  |
| --- |
| #include <iostream>  要素数を変更したら４か所修正する必要がある！！！  using namespace std;  int main(void)  {  int test[5];  cout << 5 << "人の点数を入力してください。\n";  for (int i = 0; i < 5; i++) {  cin >> test[i];  }  for (int j = 0; j < 5; j++) {  cout << j + 1 << "番目の人の点数は" << test[j] << "です。\n";  }  return 0;  } |

このように、プログラム\_4では、配列の要素数を変更すると４か所の修正が必要になります。保守性、拡張性の高いプログラムを書くためには、**定数を使える箇所は積極的に定数を使用する**ようにしましょう。

|  |
| --- |
| Tips　マジックナンバー  　先ほどの例のように、プログラムに直接書き込まれた数字のことをマジックナンバーといいます。マジックナンバーとは「この数字の意味はわからないが、とにかくプログラムは正しく動く。まるで魔法の数字だ」という皮肉を含んだ言葉です。  例えば下記のループを見てください。  for (int i = 0; i < 4; i++) {  ・  ・  省略  ・  ・  }  この４が何を意味しているか、分かりますか？では、このコードを下記のように書き換えてみましょう。  for (int i = 0; i < PLAYER\_NO\_KAZU; i++) {  ・  ・  省略  ・  ・  }  このコードだとプレイヤーの人数分回しているループだ、ということが読み取れます。このように、定数を使うことでマジックナンバーではわからなかった、プログラマーの意図をすぐに読み取れるようになります。 |

**配列の要素をソートする(P.266)**

前期でチラっと学習したバブルソートの説明が載っています。Sample3.cppを入力して、動作を確認してください。swap(交換)のアルゴリズムなどを思い出しながら入力してください。

**多次元配列の仕組みを知る(p.269)**

多次元配列を頻繁に使うことはありませんが、多次元配列を使う方が、プログラムを分か

りやすく書くことができるケースがあります。

## **Lesson\_09 実習課題\_2**

Lesson\_09\_実習課題２のプロジェクトを改造して、if文を一切使わないじゃんけんゲームを完成させなさい。

## **Lesson 9 中間テスト 2**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSdZE7J8H2sGVDmhSjqMoqiPmTriS6lQ2VyUKS9dIeymJbNd6A/viewform?usp=sf_link>

## **Lesson\_09 実習課題\_3**

Lesson\_09\_実習課題3のプロジェクトを立ち上げて、ハンズオン１、２を行った後で、課題１、２を行いなさい。

**ハンズオン\_１**

① main.cppのmain関数にBitmapクラス型の変数bmを定義する。

　② bmのメンバ関数のLoad関数を呼び出す。Load関数の引数には文字列で、”test.bmp”

を渡す。

③ bmのメンバ関数のSave関数を呼び出す。Save関数の引数には文字列で”test.2bmp”を渡す。

**ハンズオン\_2**

① Bitmap.hに記載されているBitmapクラスにRemoveRedColorメンバ関数の宣言を追加する。RemoveRedColorメンバ関数の仕様は下記のとおりです。

　　　　　戻り値　　　　なし

　　　　　引数　　　　　なし

　　　　　関数名　　　　RemoveRedColor

② Bitmap.cppにRemoveRedColorメンバ関数の定義を追加する。RemoveRedColorメンバ関数にはロードしている画像データからR成分を除去する処理が記述されています。

|  |
| --- |
| void Bitmap::RemoveRColor()  {  for (int y = 0; y < IMAGE\_H; y++) {  for (int x = 0; x < IMAGE\_W; x++) {  m\_image[y][x].r = 0; //R成分を除去する。  }  }  } |

　③ mai.cppでRemoveRedColorメンバ関数を呼び出して、ロードした画像を加工して保存する。

④ test2.bmpを確認して、下記のような画像になっていたら成功。

****

**課題１**

　下記に指定されている仕様通りにプログラムを改造して、ロードした画像を白黒にして、

保存できるようにしなさい。

仕様

　① Bitmap.hに記載されているBitmapクラスにConvertMonochromeメンバ関数の宣言を追加する。ConvertMonochromeメンバ関数の仕様は下記のとおりです。

　　　　　戻り値　　　　なし

　　　　　引数　　　　　なし

　　　　　関数名　　　　ConvertMonochrome

　② Bitmap.cppにConvertMonochromeメンバ関数の定義を追加する。

　 ConvertMonochrome関数は、Bitmap:: m\_imageに記録されているカラー情報をモノクロ化する。画像の白黒化は下記のような計算で行えます。

|  |
| --- |
| //colorが1ピクセルの情報が入っている。  float Y = 0.29900 \* color.r + 0.58700 \* color.b + 0.11400 \* color.b;  color.r = Y;  color.g = Y;  color.b = Y; |

　③ main.cppで、ConvertMonochrome関数を呼び出して、ロードした画像を白黒に変換しなさい。

　④ test2.bmpを確認して、下記のような画像になっているか確認しなさい。



**課題２**

下記に指定されている仕様通りにプログラムを改造して、ロードした画像をぼか

すことができるようにしなさい。

仕様

　① Bitmap.hに記載されているBitmapクラスにConveretBokeメンバ関数の宣言を追加

する。ConveretBokeメンバ関数の仕様は下記のとおりです。

　　　　　戻り値　　　　　なし

　　　　　引数　　　　　　なし

　　　　　関数名　　　　　ConvertBoke

② Bitmap.cppにConvertBokeメンバ関数の定義を追加する。

　　 ConvertBoke関数はBitmap::m\_imageに記録されているカラー情報を、近傍9ピクセ

ルをフェッチして、平均化していきます。

③ main.cppで、ConvertBoke関数を呼び出して、ロードした画像をぼかしなさい。

　④ test.bmpとtest2.bmpの比較を行って、下記のような画像になっているか確認しなさ

い。

**課題3**

下記に指定されている仕様通りにプログラムを改造して、ロードした画像のコントラストを強めなさい。(**コントラストを強める**ということは、「明るいところはさらに明るく、暗いところはさらに暗くしなさい」ということです。

仕様

　① Bitmap.hに記載されているBitmapクラスにUpContrastメンバ関数宣言を追加する。

　　 UpContrastメンバ関数の仕様は下記のとおりです。

　　　　戻り値　　　　　なし

　　　　引数　　　　　　なし

　　　　関数名　　　　　UpContrast

　② Bitmap.cppにUpContrastメンバ関数の定義を追加する。

　　 UpContrastメンバ関数は、Bitmap::m\_imageに記録されているカラー情報のコントラストを強めます。コントラストの強化は下記のようなコードで行えます。

明るいピクセルは1.5倍、暗いピクセルは0.5倍にしている。

|  |
| --- |
| //左下の1ピクセルだけコントラストを強めるサンプルコード。  //まず、ピクセルの明度を求める。  unsigned char akarusa = max(m\_image[0][0].r, m\_image[0][0].g);  akarusa = max(akarusa, m\_image[0][0].b);  if (akarusa > 127) {  //明るさが127以上なら、さらに明るくする。  m\_image[0][0].r = min(255.0f, m\_image[0][0].r \* 1.5f);  m\_image[0][0].g = min(255.0f, m\_image[0][0].g \* 1.5f);  m\_image[0][0].b = min(255.0f, m\_image[0][0].b \* 1.5f);  }  else {  //明るさが127未満なら、さらに暗くする。  m\_image[0][0].r \*= 0.5f;  m\_image[0][0].g \*= 0.5f;  m\_image[0][0].b \*= 0.5f;  } |

　③ main.cppで、UpContrastメンバ関数を呼び出して、ロードした画像のコントラストを強める。

　④ test.2bmpを確認して、次のページのような画像になっていることを確認しなさい。



**課題4**

下記に指定されている仕様通りにプログラムを改造して、関数呼び出しの時にコントラストの強度を指定できるようにしなさい。

仕様

　① Bitmap.hに記載されているBitmapクラスにUpContrastメンバ関数宣言にfloat型の引数を追加する。引数の名前はpowerとする。

② Bitmap.cppのUpConstractメンバ関数の定義もfloat型の引数、powerを受け取るように変更する。

③ UpConstractメンバ関数の定義の中身を、引数のpowerを使うように改造する。下記は1ピクセルだけコントラストを強化するサンプルコードです。

|  |
| --- |
| //左下の1ピクセルだけコントラストを強めるサンプルコード。  //まず、ピクセルの明度を求める。  unsigned char akarusa = max(m\_image[0][0].r, m\_image[0][0].g);  akarusa = max(akarusa, m\_image[0][0].b);  if (akarusa > 127) {  //明るさが127以上なら、さらに明るくする。  m\_image[0][0].r = min(255.0f, m\_image[0][0].r \* **( 1.0f + power )** );  m\_image[0][0].g = min(255.0f, m\_image[0][0].g \* **( 1.0f + power )** );  m\_image[0][0].b = min(255.0f, m\_image[0][0].b \* **( 1.0f + power )** );  }  else {  //明るさが127未満なら、さらに暗くする。  m\_image[0][0].r \*= **1.0f - power**;  m\_image[0][0].g \*= **1.0f - power**;  m\_image[0][0].b \*= **1.0f - power**;  } |

④ main.cppのUpConstractメンバ関数の呼び出しの際に、引数として0.9を渡すようにする。

⑤ test2.bmpを確認して、次のページのような画像になっていることを確認しなさい。



**課題5**

下記に指定されている仕様通りにプログラムを改造して、ロードした画像を上下逆さまにしなさい。

仕様

①BitmapクラスにFlipUpDownメンバ関数の宣言を追加する。

　　 FlipUpDownメンバ関数の仕様は下記のとおりです。

　　　　戻り値　　　　　　なし

　　　　引数　　　　　　　なし

　　　　関数名 FlipUpDown

② FlipUpDownメンバ関数の定義を追加する。FlipUpDownメンバ関数にBitmap::m\_imageに記録しているカラー情報を上下反転する処理が記述されています。

③ main.cppでFlipUpDown関数を呼び出して、ロードした画像を上下反転する。

　④ test2.bmpを確認して下記の画像のようになっていることを確認しなさい。



**課題6**

下記に指定されている仕様通りにプログラムを改造して、ロードした画像を左右反転しなさい。

仕様

①BitmapクラスにFlipLeftRightメンバ関数の宣言を追加する。

　　 FlipLeftRightメンバ関数の仕様は下記のとおりです。

　　　　戻り値　　　　　　なし

　　　　引数　　　　　　　なし

　　　　関数名 FlipLeftRight

② FlipLeftRightメンバ関数の定義を追加する。FlipLeftRightメンバ関数にBitmap::m\_imageに記録しているカラー情報を左右反転する処理が記述されています。

③ main.cppでFlipLeftRight関数を呼び出して、ロードした画像を上下反転する。

　④ test2.bmpを確認して下記の画像のようになっていることを確認しなさい。



**課題7**

これまでに追加したメンバ関数を利用して、test.bmpを下記の順番通りに加工してtest2.bmpという名前で保存しなさい。

**赤成分を除去→モノクロ化→コントラストを強める(強度0.4)→上下反転→左右反転**

加工ができたらtest2.bmpを確認して、下記の画像のようになっているか確認しなさい。



## **9.5 配列とポインタの関係**

配列名の仕組みを知る

重要！！！

**配列名だけを記述した場合、配列の先頭アドレスを表す。**

下記のコードを入力して動作を確認してください。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  //int型の要素数5の配列testを定義する。  int test[5];  //int型のポインタ変数のpに配列testの先頭アドレスを代入する。  int\* p = test;  //pの指している場所に10を代入する。  \*p = 10;  //test[0]に10が入っている。  std::cout << test[0] << "\n";  return 0;  } |

ポインタ演算の仕組みを知る

重要！！！

**ポインタ変数に+1を実行すると、pが指している次の要素のアドレスを得る。**

ポインタ変数に+1を実行すると、次の要素のアドレスを得るということは、次のようなコードを書けることになります。下記のコードを入力して動作を確認してください。

|  |
| --- |
| #include <iostream>  int main()  {  //配列のサイズを定数で定義する。  const int NUM\_ARRAY = 5;  //要素数が5の配列testを定義して、全要素を0で初期化する。  int test[NUM\_ARRAY] = { 0 };  //ポインタ型の変数pに配列testの先頭アドレスを代入する。  int\* p = test;  for (int i = 0; i < NUM\_ARRAY; i++) {  //pが指している箇所にi\*10の値を代入。  \*p = i \* 10;  //pをインクリメントして、次の要素のアドレスにする。  p++;  }  //配列の各要素の値が変更されていることを確認する。  for (int i = 0; i < NUM\_ARRAY; i++) {  std::cout << i << "番目の要素の値は" << test[i] << "です。\n";  }  return 0;  } |

## **9.5.Ex\_1　多次元配列もメモリ上並んで配置されている。**

　配列はメモリ上並んで配置されています。例えば、int test[6];といった配列を定義した場合、メモリ上は下記のように配置されています。int型は4バイトなのでtest[0](アドレス100に格納されている)の次のtest[1]のアドレスは104になります。



これは、多次元配列も同じです。例えば、int test[2][3];といった多次元配列を定義した場合、メモリ上は下記のように配置されています。



さて、気づいたでしょうか？実はint test[6]とint test[2][3]はメモリ上では、同様に連続して配置されています。つまり、多次元配列の場合も下記のようなコードを書くことができます。

|  |
| --- |
| #include <iostream>  int main()  {  //配列のサイズを定数で定義する。  const int NUM\_ARRAY\_W = 2;  const int NUM\_ARRAY\_H = 3;  //要多次元配列testを定義して、全要素を0で初期化する。  int test[NUM\_ARRAY\_H][NUM\_ARRAY\_W] = { 0 };  //ポインタ型の変数pに配列test[0]の先頭アドレスを代入する。  int\* p = test[0];  for (int i = 0; i < NUM\_ARRAY\_W \* NUM\_ARRAY\_H; i++) {  //pが指している箇所にi\*10の値を代入。  \*p = i \* 10;  //pをインクリメントして、次の要素のアドレスにする。  p++;  }  //配列の各要素の値が変更されていることを確認する。  for (int y = 0; y < NUM\_ARRAY\_H; y++) {  for (int x = 0; x < NUM\_ARRAY\_W; x++) {  std::cout << y << "," << x << "番目の要素の値は" << test[y][x] << "です。\n";  }  }  return 0;  } |

**重要　多次元配列もメモリ上連続して並んでいる！！！**

## **Lesson\_09 実習課題\_4**

Lesson\_09\_実習課題４を立ち上げて、下記の課題とハンズオンを行いなさい。

**課題1**

Lesson\_09\_実習課題４はビットマップ画像のイメージデータを２次元配列ではなく、1次元配列(普通の配列のこと)で保持するように変更されています。

BitmapクラスのConvertMonochromeメンバ関数の中身を実装して、1次元配列でも白黒画像に変換できるようにしなさい。

**ハンズオン**

　課題で作成したConvertMonochromeメンバ関数の実装をポインタを用いて、画像を変換するようにしてみよう。

**課題2**

Lesson\_09\_実習課題３で実装したRemoveRedColor 、ConvertBoke、UpContrast、FlipUpDown、FlipLeftRight関数をLesson\_09\_実習課題4でも実装しなさい。

## **9.5.Ex\_1 ポインタ演算の仕組みをさらに詳しく見ていく。**

9.5節で、「ポインタ変数に+1を実行するとpが指している次の要素のアドレスを得る。」と教えましたが、実はこれは正しい説明ではありません。正確には、「pが指している型のサイズ分だけpが記録しているアドレスに加算される。」となります。

　では、下記のコードを入力してみてください。

|  |
| --- |
| #include <iostream>  using namespace std;  //プレイヤーのデータを表す構造体。  struct PlayerData {  int hp; /ヒットポイント。 hpのサイズは① byte  int mp; //マジックポイント。mpのサイズは② byte  short atk; //攻撃力。 atkのサイズは③ byte  short def; //防御力。 defのサイズは④ byte  char level; //レベル。 levelのサイズは⑤ byte  char age; //年齢。 ageのサイズは⑥ byte  }; PlayerDataのサイズは⑦ byte  /\*!  \*@brief メイン関数。  \*/  int main()  {  //PlayerData型の変数を定義。  PlayerData pl\_data;  //プレイヤーのデータを設定していく。  pl\_data.hp = 2000; //ここではpl\_data.hpには2000が代入されている。  pl\_data.mp = 1000; //ここではpl\_data.mpには1000が代入されている。  pl\_data.atk = 512;  pl\_data.def = 512;  pl\_data.level = 10;  //pl\_dataのアドレスをpに代入。  //キャストを使うことで異なる型に代入することが可能。  int\* p = (int\*)&pl\_data;  //この時pはpl\_data.hpを指している。  \*p = 4000;  //pに1加算する。pはint型(4バイト)のポインタ変数なので、  //アドレスは4加算される。  p+=1;  //この時pはpl\_data.mpを指している。  \*p = 3000;  //pl\_data.hpとpl\_data.mpの値を確認してみる。  std::cout << "pl\_data.hpは" << pl\_data.hp << "\n";  std::cout << "pl\_data.mpは" << pl\_data.mp << "\n";  return 0;  } |

いかがでしょうか、int型のポインタ変数のpにPlayerData型のpl\_dataのアドレスを代入していますが、pの指している先の値を変えることによって、pl\_data.hpやpl\_data.mpの値を変えることができたと思います。要は、**「ポインタ変数はアドレスを記録しているだけに過ぎない」**ということです。

では、先ほどのサンプルプログラムをもう少し詳しく見ていきましょう。

PlayerData型のpl\_dataはメモリ上、下記のように領域を確保されます。



pl\_data.age

pl\_data.level

pl\_data.def

pl\_data.atk

pl\_data.mp

pl\_data.hp

では、プログラムを追いかけてみましょう。

① ポインタ変数pにpl\_dataのアドレスを代入。



p(100)

pl\_data.age

pl\_data.level

pl\_data.def

pl\_data.atk

pl\_data.mp

pl\_data.hp

②pの指している先に4000を代入。



p(100)

pl\_data.age

pl\_data.level

pl\_data.def

pl\_data.atk

pl\_data.mp

pl\_data.hp

③　p+=1;を実行して、pが記録しているアドレスに4を加算。



p(104)

pl\_data.age

pl\_data.level

pl\_data.def

pl\_data.atk

pl\_data.mp

pl\_data.hp

③ pの指している先に3000を代入。



p(104)

pl\_data.age

pl\_data.level

pl\_data.def

pl\_data.atk

pl\_data.mp

pl\_data.hp

キモとなるのは、**ポインタ変数が記録しているのはアドレス(数値)に過ぎない**という点です。

では、先ほどのプログラムに、下記の網掛けになっているコードを追加で入力してみてください。

|  |
| --- |
| int main()  {  　　　…  　　　省略  　　　…    //pl\_data.hpとpl\_data.mpの値を確認してみる。  std::cout << "pl\_data.hpは" << pl\_data.hp << "\n";  std::cout << "pl\_data.mpは" << pl\_data.mp << "\n";  **//pが記憶しているアドレスに4加算する。**  **p += 1;**  **//shortのポインタ型のspにpが記憶しているアドレスを代入。**  **short\* sp = (short\*)p;**  **//この時spはpl\_data.atkを指している。**  **\*sp = 1000;**  **//spに1を加算する。spはshort型(2バイト)のポインタ変数なので**  **//アドレスは2加算される。**  **sp += 1;**  **//この時spはpl\_data.defを指している。**  **\*sp = 2000;**  **//pl\_data.atkとpl\_data.defの値を確認してみる。**  **std::cout << "pl\_data.atkは" << pl\_data.atk << "\n";**  **std::cout << "pl\_data.defは" << pl\_data.def << "\n";**  return 0;  } |

　では、これも処理を追いかけてみましょう

① p+=1;を実行して、pが記録しているアドレスに4を加算。



p(108)

pl\_data.age

pl\_data.level

pl\_data.def

pl\_data.atk

pl\_data.mp

pl\_data.hp

②　shortのポインタ型のspにpのアドレスを代入。



sp(108)

p(108)

pl\_data.age

pl\_data.level

pl\_data.def

pl\_data.atk

pl\_data.mp

pl\_data.hp

③　spが指している先に1000を代入



sp(108)

p(108)

pl\_data.age

pl\_data.level

pl\_data.def

pl\_data.atk

pl\_data.mp

pl\_data.hp

④ sp+=1を実行して、spが記録しているアドレスに2を加算。spはshort型のポインタ変数なので、加算されるのは2になります。



sp(110)

p(108)

pl\_data.age

pl\_data.level

pl\_data.def

pl\_data.atk

pl\_data.mp

pl\_data.hp

⑤　spが指している先に2000を代入。



sp(110)

p(108)

pl\_data.age

pl\_data.level

pl\_data.def

pl\_data.atk

pl\_data.mp

pl\_data.hp

重要！！！

**ポインタ変数のアドレスに加算される数値は、型によって変わる**

## **9.5.Ex\_2 ハンズオン**

**ハンズオン\_1**

TSXBINというツールを使って、Lesson\_09実習課題５/Bitmap/test.bmpファイルの中身を確認してみる。

**ハンズオン\_2**

　Lesson\_09実習課題5を使って、ビットマップファイルを読み込む処理を実装してみましょう。

## **Lesson 9 中間テスト3**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/1ImyI0Kcq-CXJ6AA5mEzf9oOawpu4Zvu3PtFA5rPLSGk/edit>

## **9.6 配列と引数**

**配列を引数として使う(p.279)**

VisualStudioのプロジェクトのLesson\_9を立ち上げて、Sample7.cppを入力してください。

配列名は先頭要素のアドレスをあらわすので、関数には配列の先頭要素のアドレスだけが渡されていることになります。

　→つまり、ポインタでもいける？

**ポインタを引数として使う(p.280)**

先ほど入力したプログラムの引数の部分だけ、下記のように書き換えてみてください。

|  |
| --- |
| …  　　省略  　　…  //avg関数の宣言  double avg(**int\*** t);  　　…  　　省略  　　…  //avg関数の定義  double avg(**int\*** t)  {  double sum = 0;  for (int i = 0; i < 5; i++) {  sum += t[i];  }  return sum / 5;  } |

avg関数の引数をint t[]からint\* tに変更しています。**配列とポインタは密接な関係があって、同じように扱うことができる**という性質を思い出してくださいavg関数に渡されていたのは、配列の先頭アドレスだったので、引数をポインタ型にすることが可能です。

　このように配列を関数に渡すときには、ポインタ型の引数がよく使われるので、しっかりと覚えておきましょう。

**重要！！！**

**配列を関数に渡すときはポインタ型の引数を使える！**

## **9.7 文字列と配列**

**文字列と配列の関係を知る(p.284)**

char型の変数は文字を覚えることができる。

|  |
| --- |
| char moji = 'A'; |

文字列=文字の集合

→**文字列=char型の配列**

必ず最後に’\0’という値をつける

NULL文字を忘れると、コンピュータが文字列の終わりが分からなくなってしまいます。NULL文字がない文字列をコンピュータに渡した場合の動作は不定(決まっていない)となります。では、下記のプログラムを入力して、NULL文字を忘れてしまった場合の挙動を確認してみてください。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  char str[6];  str[0] = 'H';  str[1] = 'E';  str[2] = 'L';  str[3] = 'L';  str[4] = 'O';  //終端にNULL文字を入れるのを忘れた！！！  //str[5] = '\0';  cout << str;  return 0;  } |

さて、いかがでしょうか？私の環境では下記のような文字列が表示されました。



恐らく、みなさんの環境では異なる結果になったのではないでしょうか？このように、NULL文字がない文字列を、コンピュータに渡した場合の動作は不定の結果となりますので、注意しましょう。

**文字列配列を初期化する(p.286)**

“”を使って文字列配列を初期化した場合は、自動的にNULL文字(\0)が付け加えらえることになっています

　この仕様のため、下記のようなコードはコンパイルエラーとなります。



HELLOという文字列を配列に記録するためには、HELLO(5文字)+NULL文字(1文字)が必要になります。そのため、要素数5の配列のstrでは記録することができないので、このようなエラーが発生します。

**文字列配列を出力する(p.286)**

VisualStudioのプロジェクトのLesson\_9を立ち上げて、Sample8.cppを入力してください。

**文字列をポインタで扱う(p.287)**

教科書を読み進める前に、教科書のコードには間違いがあります。ポインタで文字列リテラル(後述)を扱う場合は、下記のようにconst修飾子をつける必要があります。

|  |
| --- |
| const char\* str = "HELLO"; |

constがついていない場合はコンパイルエラーになるので注意して下さい。

ポインタで扱う場合には、””を使うことで、メモリ上のどこか別の場所に文字列が格納され、その場所を指し示すようになっています。

→このような文字列は**文字列リテラル(文字列定数)**となります。文字列**定数**なので、文字を変更することはできません。

プログラムに割り当てられるメモリは下記の図のように、いくつかの領域に分かれています。



メモリ領域の役割は下記のようになります。



　これまでのC++の授業で扱ってきたメモリは、ほとんどがスタック領域の**ローカル変数**と呼ばれるものです。(学内のtkEngineのNewGOで確保されるメモリはヒープ領域です。)

文字列リテラルはメモリ上の静的領域の書き換え禁止の場所に記憶されており、配列で記録できる文字列のように変更することはできません。

では、下記のコードを入力してみてください。配列の文字列は文字を変更することができますが、文字列リテラルは変更することができないため、コンパイルエラーが起きるはずです。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  //"HELLO"は書き換え禁止の領域に記録されている。  const char\* str\_1 = "HELLO";  //この"HELLO"はスタック領域に記録されている。  char str\_2[] = "HELLO";  str\_1[0] = 'h'; //書き換えることができない！！！  str\_2[0] = 'h'; //書き換え可能。  return 0;  } |

**配列とポインタの違いを知る(p.289)**

文字列リテラルを指しているポインタは、文字列リテラルの文字を変更することはできませんが、指している文字列リテラルを変更することはできます。

下記のコードを入力して動作を確認してください。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  //"HELLO"は書き換え禁止の領域に格納されている。  //pは"HELLO"が格納されているアドレスを覚えているだけ！  const char\* p = "HELLO";  //HELLOと表示される。  cout << p << "\n";  //p[0] = 'a'; //これはできない！  //指しているアドレスを変更することはできる。  p = "WORLD";  //WORLDと表示される。  cout << p << "\n";  return 0;  } |

## **Lesson 9 中間テスト4**

<https://docs.google.com/forms/d/e/1FAIpQLSdPxUDagLmt4EdXsZI6ZiijnjsqiWjWUIjlkQXT8H2E6fbeSg/viewform?usp=sf_link>

## **文字列を操作する**

VisualStudioのプロジェクトのLesson\_9を立ち上げて、Sample10.cppを入力してください。

下記のSample10.cppのfor文は文字列を操作する処理の典型的なループなので、しっかりと理解してください。

|  |
| --- |
| for ( int i = 0; **str[i] != '\0'**; i++ ) {  } |

　strのi番目が終端文字になるまでループします。

## **文字列を扱う標準ライブラリ**

　標準ライブラリとは、言語に用意されている機能の事です。下記のURLにジャンプしてみてください。

<https://ja.cppreference.com/w/cpp/header>

　このように、標準ライブラリとは膨大な量となっています。これらをすべて覚えているプログラマーはいないでしょうし、これらを覚えようとする人はプログラマーに向いていないと思います。プログラマ―にとって重要なスキルは**調べる力**です。例えば、C++言語で文字列をコピーする処理を調べるのであれば、googleなどの検索エンジンを使って、「**C++ 文字列　コピー**」などと検索します。文字列の連結がしたい場合は「**C++ 文字列　連結**」といった具合です。最初は「どんなワードで検索すればいいのか分からない＞＜」となると思いますが、回数をこなせば分かってくるので、どんどん調べ物をしていきましょう。

## **9.7.Ex\_1　ハンズオン**

Lesson\_09 実習課題\_5のStringTest.cppのMyPrint関数を自作してみよう。

## **Lesson\_09 実習課題\_5**

Lesson\_09 実習課題\_5のStringTest.cppの4つの関数を実装しなさい。

　・MyStrlen

・MyStrcpy

・MyStrcmp

・MyStrcat

また、関数の仕様はコメントから推測しなさい。

## **Lesson 9 中間テスト5**

<https://docs.google.com/forms/d/e/1FAIpQLSd0TXmFLNZo7LXdI_kewSdR43sxpcgv1dze15MFP1aWV2MazA/viewform?usp=sf_link>

## **9.8.Ex 文字列クラスを作ろう**

## **9.8.Ex.1 C言語は文字列をまともに扱えない**

まず、最初にC言語には文字列を扱う型は存在しません。

「(。´・ω・)ん?、文字列はcharの配列なんじゃないの？」

と思いましたか？違います。**charの配列は文字の配列です**。決しても文字列ではありません。文字の配列を文字列として無理矢理扱っているだけです。

(。´・ω・)ん? (。´・ω・)ん? (。´・ω・)ん? (。´・ω・)ん? (。´・ω・)ん?

なんのこっちゃい？何が違うの？という感じだと思いますが、繰り返しますが、C言語は文字列をまともに扱えない言語です。文字列をまともに扱える言語には**文字列型**というものがちゃんと用意されています。

では、文字列をちゃんと扱える言語を少しだけ触ってみましょう。下記のexeファイルを実行して、pythonをインストールしてください。

・python-3.7.2.exe

　あと、テキストエディタもインストールしておきましょう。

　・sakura\_install2-2-0-1.exe

## **9.8.Ex.2 python超入門！！！！**

Pythonはスクリプト言語で、現在大変人気のあるプログラミング言語です。Pythonと機械学習できます！！！と嘯いておけば、とりあえず仕事が見つかるのではないでしょうか。（後でどうなるかは知りませんが）ネタのようなホントの話です。

では、pythonでプログラムを書いて実行してみましょう。まずは、サクラエディタを立ち上げて、下記のコードを入力してみてください。

|  |
| --- |
| print("Hello Python!!!"); |

入力出来たら、テキストを保存しましょう。デスクトップにtest.pyというファイル名で保存してください。この時に文字コードのセットをUTF-8に変更しておくのを忘れないでください。

参考動画URL

<https://youtu.be/cJHzmqAoRIc>

テキストを保存出来たら、下記の動画を参考にして、プログラムを実行してみましょう。今回のプログラムの実行はコマンドプロンプトを使います。ちょっとできるプログラマっぽくてかっこいいです。

参考動画URL

<https://youtu.be/rMPpEJm56sM>

どうでしょう、Hello Pythonと表示できたでしょうか。

では、続いて、C言語とPythonを比較して、文字列をまともに扱えるということの意味を見ていきましょう。

## **9.8.Ex.3 C言語は文字列のコピーができない！！！**

C言語は文字列のコピーができません。・・・もちろん下記のようにstrcpy系の関数を使えばできます。

Visual StudioのLesson\_09を立ち上げて、下記のプログラムを入力して実行してください。

|  |
| --- |
| //画面に文字を出力するコード  #include <iostream>  using namespace std;  int main()  {  char hoge[256] = "Hello World\n";    //Hello Worldと表示される。  printf(hoge);  //hogeに"こんにちは世界"がコピーされる  strcpy\_s(hoge, "こんにちは　世界\n");  //こんにちは世界と表示される。  printf(hoge);  //今度はhogeに顔文字(しょぼーん)をコピーする  strcpy\_s(hoge, "(´・ω・｀)\n");  //(´・ω・｀)が表示される。  printf(hoge);  return 0;  } |

どうでしょう、確かにhogeに文字列をコピーすることができました。「ほらC言語だって文字列をちゃんと扱えるじゃないか！」と言いたくなると思います。しかし、これはまともに文字列を扱えていません。

　文字列をコピーするのであれば、下記のようなコードを書きたくなりませんか？

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  char hoge[256] = "Hello World\n";    //Hello Worldと表示される。  printf(hoge);  //hogeに"こんにちは世界"がコピーされる  **hoge = "こんにちは世界\n";**  //こんにちは世界と表示される。  printf(hoge);  //今度はhogeに顔文字(しょぼーん)をコピーする  **hoge = "(´・ω・｀)\n";**  //(´・ω・｀)が表示される。  printf(hoge);  return 0;  } |

数字のコピーと同じように、代入演算子を使いたいですよね？しかし、C言語ではこれはできません。

　では、先ほどデスクトップに保存したtest.pyを開いて、下記のようなコードを入力してみてください。

|  |
| --- |
| #変数hogeにHello Worldをコピーする。  hoge = "Hello World\n";  #Hello Worldと表示される。  print( hoge );  #変数hogeにこんにちは世界をコピーする。  hoge = "こんにちは世界\n";  print( hoge );  #変数hogeに(´・ω・｀)をコピーする。  hoge = "(´・ω・｀)\n";  print( hoge ); |

いかがでしょうか？C言語とは異なり、文字列のコピーが直観的だと思います。これが文字列をまともに扱えるということです。

## **9.8.Ex.4 C言語は文字列の連結ができない！！！**

　C言語は文字列の連結ができません。・・・もちろん、ええ、もちろん下記のようにstrcat系の関数を使えばできます。

Visual StudioのLesson\_09を立ち上げて、下記のプログラムを入力して実行してください。

|  |
| --- |
| //画面に文字を出力するコード  #include <iostream>  using namespace std;  int main()  {  char hoge[256] = "Hello";    //Helloと表示される。  printf(hoge);  //改行する。  printf("\n");  //hogeに"World"を連結する  strcat\_s(hoge, "World");  //Hello Worldと表示される。  printf(hoge);  //改行する。  printf("\n");  //hogeに好きな顔文字を連結してみよう。  strcat\_s(hoge, "(\*ﾉωﾉ)");  printf(hoge);  //改行する。  printf("\n");  return 0;  } |

　どうでしょう、C言語でも文字列を連結することができました。しかし・・・・・・、+演算子使いたくないですか？下記のようなプログラムの方が分かりやすくないですか？

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  char hoge[256] = "Hello";    //Helloと表示される。  printf(hoge);  //改行する。  printf("\n");  //hogeに"World"を連結する  **hoge += "World";**  //Hello Worldと表示される。  printf(hoge);  //改行する。  printf("\n");  //hogeに好きな顔文字を連結してみよう。  **hoge += "(\*ﾉωﾉ)";**  printf(hoge);  //改行する。  printf("\n");  return 0;  } |

　しかし、例のごとくC言語ではこのようなコードを書くことはできません。では、Phythonのtest.pyを開いて、下記のコードを入力して実行してみてください。

|  |
| --- |
| #変数hogeにHelloをコピーする。  hoge = "Hello";  #Hello Worldと表示される。  print( hoge );  #変数hogeにWorldを連結する。  hoge += "World";  print( hoge );  #変数hogeに好きな顔文字を連結。  hoge += "(\*- -)(\*\_ \_)ペコリ";  print( hoge ); |

　どうでしょうか。そろそろC言語を投げ捨てて、pythonでプログラムを書きたくなってきたのではないでしょうか？これでもまだC言語の方がいいというのであれば、相当な変人だと思います。

## **9.8.Ex.5 C言語は文字列が同じかどうか判定できない！！！**

ええ、確かにstrcmpを使えば文字列が同じかどうか判定できます。Lesson\_9に下記のコードを入力してみてください。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  char hoge[256] = "Hello";    //strcmpは文字列が同じ場合は0を返してくる    if (strcmp(hoge, "Hello") == 0 ) {  printf("hogeにはHelloという文字列が設定されています。\n");  }  return 0;  } |

・・・、==演算子使いたくないですか？下記のようなコードの方が分かりやすくないですか？

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  char hoge[256] = "Hello";    //strcmpは文字列が同じ場合は0を返してくる    if ( **hoge == "Hello"** ) {  printf("hogeにはHelloという文字列が設定されています。\n");  }  return 0;  } |

ええ、もちろんC言語でこんな書き方はできません。ですが、安心してください。我々にはpythonがあります。

test.pyに下記のようなコードを入力して、実行してみてください。

|  |
| --- |
| #変数hogeにHelloをコピーする。  hoge = "Hello";  if hoge == "Hello":  print( "hogeにはHelloという文字列が設定されています。\n"); |

(※Pythonはインデント、空白が意味を文法的な意味を持っています。printの前にインデントを入れるのを忘れないようにしてください。少しだけPythonが嫌いになりますね。)

## **9.8.Ex.6 C++には文字列型がある！！！**

　文字列をまともに扱えるということは、整数型(int型)と同じように代入演算子、加算演算子、比較演算子などが扱えるということです。なので、**C言語は文字列をまともに扱えない！**ということになります。

　さて、ここまで意図的にC++とC言語を区別して説明してきました。C++はC言語のスーパーセット言語です。そうです、安心してください。C++にはちゃんとstd::stringという文字列型があります！良かったですね！では、std::stringを使ったコードをいくつか書いていきましょう。

## **9.8.Ex.6.1 std::stringを使った文字列のコピー**

　Lesson\_9に下記のコード入力してみてください。

|  |
| --- |
| #include <iostream>  #include <string> //これを追加するのを忘れないでね。  using namespace std;  int main()  {  string hoge = "Hello World\n";  //Hello Worldと表示される。  cout << hoge;  //hogeに"こんにちは世界"がコピーされる  hoge = "こんにちは世界\n";  //こんにちは世界と表示される。  cout << hoge;  //今度はhogeに顔文字(しょぼーん)をコピーする  hoge = "(´・ω・｀)\n";  //(´・ω・｀)が表示される。  cout << hoge;  return 0;  } |

いかがでしょうか？C言語では使えなかった、代入演算子が使えたはずです。

## **9.8.Ex.6.2 std::stringを使った文字列の連結**

では、続いて下記のコードを入力してみてください。

|  |
| --- |
| #include <iostream>  #include <string> //これを追加するのを忘れないでね。  using namespace std;  int main()  {  string hoge = "Hello";  //Helloと表示される。  cout << hoge << "\n";  //hogeに"World"を連結する  hoge += "World";  //Hello Worldと表示される。  cout << hoge << "\n";  //hogeに好きな顔文字を連結してみよう。  hoge += "(\*ﾉωﾉ)";  cout << hoge << "\n";  return 0;  } |

+演算子も使えましたね。

## **9.8.Ex.6.3 std::stringを使った文字列の比較**

では、std::stringの勉強の最後に下記のコードを入力してみてください。

|  |
| --- |
| #include <iostream>  #include <string> //これを追加するのを忘れないでね。  using namespace std;  int main()  {  string hoge = "Hello";  //比較演算子を使える！  if (hoge == "Hello") {  cout << "hogeにはHelloという文字列が設定されています。\n";  }  return 0;  } |

このように、C++にはstd::stringという文字列型があります。なので、C++は文字列をまともに扱える言語だと言えます。

さて、ここまで見てきた、PythonとC++の文字列に対する操作ですが、実はこの二つの言語に限らず、文字列というのは根本的にはC言語のchar型の配列です。Pythonの文字列型もC++の文字列型も、C言語のchar型の配列を用いて作られていると考えて構いません。なので、C言語での文字列の扱い方を理解するということは、**他言語の文字列型に対する深い理解を得る**ということになります。(そして、残念なことに深い理解を得た結果、C言語最高！となってしまったのがゲームプログラマです。)

では、次の節では文字列型への深い理解を得るために、自作のMyStringクラスを作っていきましょう。

## **9.8.Ex.7 MyStringクラス**

実はstd::stringはC++が用意している文字列クラスです。C++では文字列クラスを作成することで、文字列をまともに扱えるようにしています。では、ここからは、std::stringクラスを模して、自作のMyStringクラスを作成していきましょう。

　MyStringクラスは下記の3点の機能をハンズオンで実装していきます。

　　① 文字列のコピーができる。

　　② 文字列の連結ができる。

　　③ 文字列の比較ができる。

　続いて、下記の3点の機能を実習課題として、実装していきます。

　　①　文字列の長さの取得機能。

　　②　文字の位置の検索機能。

　　③　文字の挿入。

　では、まずVisualStudioのLesson\_09を立ち上げて、MyStringクラスをプロジェクトに追加してください。クラスの追加の仕方は覚えていますか？忘れてしまったという人は、ここで思い出しましょう。

　MyStringクラスの追加ができたら、下記のようにMyStringクラスにcharのポインタ型のm\_strignというメンバ変数を追加してください。

　MyString.h

|  |
| --- |
| class MyString  {  public:  MyString();  ~MyString();  **//文字列の先頭アドレスを記録するためのメンバ変数を追加する。**  **//一応nullptrで初期化しておく。**  **char\* m\_string = nullptr;**  }; |

続いて、C言語の標準関数を利用したいので、MyString.cppにstringのインクルードの追加と、非セキュア関数の使用の警告を無視したいので、\_CRT\_SECURE\_NO\_WARNINGSの#defineを追加してください。

|  |
| --- |
| **#define \_CRT\_SECURE\_NO\_WARNINGS //マイクロソフトの警告がうっとおしいので・・・。**  #include "MyString.h"  **#include <string>**  MyString::MyString()  {  }  MyString::~MyString()  {  } |

**Tips**

|  |
| --- |
| stringは日本語に訳すと、「一連のもの」、「一続きのもの」といった意味になります。character(文字) string(続きもの)で文字列という意味になります。  　ただし、コンピュータの世界ではstringというとほとんどの場合で文字列を指します。  なので、strcpy関数はstr(string)cpy(copy)の略称なわけですね。strcmp関数はstr(string)cmp(compare)の略称です。string=文字列ということを覚えておけば、今後プログラムを読むのが少しだけ簡単になると思います。  　また、余談ですが、関数名に略称が用いられている理由は、メモリが今より貴重だった時代は、文字数すら節約したかったからです。PS2のファイルシステムにはファイル名は5文字、拡張子は3文字までという制約や、一つのフォルダに含まれるファイルは20個まで、といった制約があったりしました。 |

## **9.8.Ex.7.1 HandsOn-1 MyStringクラス(文字列のコピー)**

では、文字列コピーを行う関数を作成してみましょう。ここまで見てきたように、代入演算子を使って、コピーできるようにしたいのですが、まずは、Copyというメンバ関数でコピーできるようにしてみましょう。

　まずは、Copyメンバ関数の宣言をMyStringクラスに追加します。

MyString.h

|  |
| --- |
| class MyString  {  public:  MyString();  ~MyString();  //文字列のコピーを行うメンバ関数。  void Copy(const char\* str);  //文字列の先頭アドレスを記録するためのメンバ変数を追加する。  //一応nullptrで初期化しておく。  char\* m\_string = nullptr;  }; |

続いて、Copyメンバ関数の定義を作成しましょう。意味が分からないコードがいくつかあると思いますが、それは後で解説します。

MyString.cpp

|  |
| --- |
| void MyString::Copy( const char\* str )  {  //コピー元の文字列の長さを調べる。  int len = strlen(str);  //コピーすることが可能なメモリを動的に確保する。  //malloc関数は確保するメモリのサイズを引数に受け取ります。  //戻り値は、確保したメモリのアドレスを返してきます。  m\_string = (char\*)malloc( len + 1 );  //文字列を確保したメモリにコピーする。  strcpy(  m\_string, //コピー先のメモリの先頭アドレス。  str //コピー元の文字列の先頭アドレス。  );  } |

　これで、ひとまずCopy関数は完成です(不具合がありますが)。では、MyStringクラスをインスタンス化して、使ってみましょう。Source.cppに下記のコードを入力してみてください。

Source.cpp

|  |
| --- |
| #include <iostream>  #include "MyString.h"  using namespace std;  int main()  {  char hoge [] = "Hello";  //hogeをstrにコピーする。  MyString str;  str.Copy( hoge );  //ちゃんとコピーできているか確認する。  cout << str.m\_string << "\n";  return 0;  } |

## **9.8.Ex.7.1.1 メモリの動的確保**

　MyString::Copy関数では、C言語のmalloc関数を利用して、メモリの動的確保が行われています。

　では、メモリ確保の前に、文字列のコピーについて再考してみましょう。例えば、下記のようなコードの場合。

|  |
| --- |
| //"Hello"の5文字とヌル文字'\0'を格納する必要があるので、  //配列strの要素数は6になる。  char srcStr[] = "Hello";  //Helloをコピーするためには、ヌル文字を含む6文字コピーできるだけの  //メモリが必要になる。  char dstStr[6];  //srcStrに格納されている文字列を、dstStrにコピー。  strcpy\_s(dstStr, srcStr); |

　srcStrの文字列をdstStrにコピーしているのですが、この場合srcStrとdstStrは6バイト分のメモリが確保されており、メモリ上では下記の図のようになります。



dstStr

srcStr

そして、strcpyを行うと下記のようになります。



これがstrcpy関数で行われている処理。

dstStr

srcStr

では、下記のようなコードを書くとどうなるでしょうか？網掛けの部分のコードが追加されています。

|  |
| --- |
| //"Hello"の5文字とヌル文字'\0'を格納する必要があるので、  //配列strの要素数は6になる。  char srcStr[] = "Hello";  //Helloをコピーするためには、ヌル文字を含む6文字コピーできるだけの  //メモリが必要になる。  char dstStr[6];  //srcStrに格納されている文字列を、dstStrにコピー。  strcpy(dstStr, srcStr);  **//dstStrは6文字しかコピーできないのに、12文字コピーしようとしている！**  **strcpy(dstStr, "Hello World");** |

結論から言いますと、これでもプログラムは動作しますが、メモリ破壊という致命的な不具合が発生しています。

この場合、メモリ上は次の図のような操作が行われています。



文字列リテラル

dstStr

**配列の範囲外にコピーされてしまっている！！！**

これがメモリ破壊と呼ばれもので、致命的な不具合を発生させてしまいます。では、この不具合を回避するためには、下記のようなコードにすればいいのでしょうか？

|  |
| --- |
| //"Hello"の5文字とヌル文字'\0'を格納する必要があるので、  //配列strの要素数は6になる。  char srcStr[] = "Hello";  **//12文字コピーできるように、dstStrの要素数を12にする。**  **char dstStr[12];**  //srcStrに格納されている文字列を、dstStrにコピー。  strcpy(dstStr, srcStr);  //これでメモリ破壊は起きない。  strcpy(dstStr, "Hello World"); |

　確かに、これで”Hello World”という文字列はコピーできるようになりました。

では、次のようなコードではどうなるでしょうか？

|  |
| --- |
| //"Hello"の5文字とヌル文字'\0'を格納する必要があるので、  //配列strの要素数は6になる。  char srcStr[] = "Hello";  //12文字コピーできるように、dstStrの要素数を12にする。  char dstStr[12];  //srcStrに格納されている文字列を、dstStrにコピー。  strcpy(dstStr, srcStr);  //これでメモリ破壊は起きない。  strcpy(dstStr, "Hello World");  //また、メモリ破壊が起きる・・・。  strcpy(dstStr, "Hello World こんにちは\n"); |

　さらに、最近のゲームでは、ユーザー名やチャットなど、ユーザーが自由に入力できる文字列を扱うことがよくあります。このような処理でも、内部では文字列のコピーが行われることがあります。

ここで理解してほしいことは、**「文字列のコピーで何文字のコピーが行われるかは、コンパイル時点では分からない」**ということです。このように、必要なメモリのサイズが、プログラム実行時まで分からないといった時に行われるものが、メモリの動的確保です。

では、MyString::Copy関数の定義を再度見ていきましょう。

|  |
| --- |
| void MyString::Copy( const char\* str )  {  //① コピー元の文字列の長さを調べる。  int len = strlen(str);  //②　コピーすることが可能なメモリを動的に確保する。  //malloc関数は確保するメモリのサイズを引数に受け取ります。  //戻り値は、確保したメモリのアドレスを返してきます。  **m\_string = (char\*)malloc( len + 1 );**  //③　文字列を確保したメモリにコピーする。  strcpy(  m\_string, //コピー先のメモリの先頭アドレス。  str //コピー元の文字列。  );  } |

　mallocはC言語の標準関数で、下記のような関数宣言となっています。

void＊ malloc( size\_t size );

malloc関数は引数に割り当てるサイズ(バイト単位)を受け取って、確保したメモリの先頭アドレスを戻り値として返してきます。このコードを実行すると、メモリ上は次の図のようになります。

strはHogeという文字列を指しているものとする。



① strlenはヌル文字を含まない、文字数を返すので、この場合は4を返す

str

② malloc(len+1)で確保されたメモリ。+1はヌル文字の分。

m\_string

③ strcpy\_sで文字列をコピー

動的確保を行うことで、必要なメモリを実行時に割り当てることができるようになります。これにより、文字列の長さに縛られることなく、コピーを行うことができるようになります。

　では、のコードをSource.cppに記入して、文字列のコピーが行えていることを確認してみましょう。

Source.cpp

|  |
| --- |
| #include <iostream>  #include "MyString.h"  using namespace std;  int main()  {    char hoge [] = "Hello";  //hogeをstrにコピーする。  MyString str;  str.Copy( hoge );  //"Helloと表示される"。  cout << str.m\_string << "\n";  //"こんにちは 世界"をコピー。  str.Copy("こんにちは　世界");  //"こんにちは　世界"と表示される。  cout << str.m\_string << "\n";  return 0;  } |

さて、このプログラムは**メモリリーク**という致命的な不具合が発生しています。これを次のチャプターで修正してみましょう。

## **9.8.Ex.7.1.3　動的に確保したメモリの解放**

動的に確保したメモリは、不要になったら明示的に開放する必要があります。malloc関数を使って確保したメモリは、不要になったらfree関数を使用して、解放してやりましょう。MyStringクラスのデストラクタを下記のように改造してください。

MyString.cpp

|  |
| --- |
| MyString::~MyString()  {  //確保したメモリはちゃんと開放する！  free(m\_string);  } |

これで、MyStringクラスのインスタンスが破棄されるときに、きちんとメモリが解放されるようになりました。しかし、まだMyStringクラスは改造が必要です。前節で作成したCopy関数は下記の網掛けになっている箇所でメモリリーク(メモリの開放忘れ)という不具合が発生しています。

|  |
| --- |
| #include <iostream>  #include "MyString.h"  using namespace std;  int main()  {  char hoge [] = "Hello";  //hogeをstrにコピーする。  MyString str;  str.Copy( hoge );  //"Helloと表示される"。  cout << str.m\_string << "\n";  ひとつ前のCopy関数で確保したメモリがリークしている。  //"こんにちは 世界"をコピー。  **str.Copy("こんにちは　世界");**  //"こんにちは　世界"と表示される。  cout << str.m\_string << "\n";  return 0;  } |

　このプログラムを実行すると、メモリは下記のようになっています。



str.m\_string

2回目のCopyで確保した領域。

最初のCopyで確保した領域が解放されていない！

これを修正するために、MyString::Copy関数に下記の網掛けのコードを追加してください。

|  |
| --- |
| void MyString::Copy(const char\* str)  {  **if (m\_string != nullptr) {**  **//m\_stringにアドレスが入っていたら開放する。**  **free(m\_string);**  **}**  //コピー元の文字列の長さを調べる。  int len = strlen(str);  //コピーすることが可能なメモリを動的に確保する。  //malloc関数は確保するメモリのサイズを引数に受け取ります。  //戻り値は、確保したメモリのアドレスを返してきます。  m\_string = (char\*)malloc( len + 1 );  //文字列を確保したメモリにコピーする。  strcpy(  m\_string, //コピー先のメモリの先頭アドレス。  str //コピー元の文字列。  );  } |

これで、Copy関数で発生していたメモリリークは次の図のように解決しました。

最初のCopyで確保した領域が正しく解放されている！

str.m\_string



2回目のCopyで確保した領域。

メモリの動的確保は非常に難しい概念ですが、とても重要です。動的確保は10章の「大規模なプログラムの作成」でも見ていきますので、そこでまた復習しましょう。

## **9.8.Ex.7.1.4 文字列のコピーで代入演算子を使えるようにしてみよう。**

　ここまでの実習で文字列のコピーが行えるようになりましたが、文字列のコピーは代入演算子を使って行いたいですよね？そこで、C++の機能の演算子のオーバーロードを使って、代入演算子を使えるようにしてみましょう。

　では、次のようにMyStringクラスにoperator=演算子のメンバ関数宣言を追加しましょう。(代入演算子も関数なのです！)

MyString.h

|  |
| --- |
| class MyString  {  public:  MyString();  ~MyString();  //代入演算子をオーバーロード。  void operator=(const char\* str);  //文字列のコピーを行うメンバ関数。  void Copy(const char\* str);  //文字列の先頭アドレスを記録するためのメンバ変数を追加する。  //一応nullptrで初期化しておく。  char\* m\_string = nullptr;  }; |

　続いて、operator=のメンバ関数定義を追加しましょう。関数の中身はCopy関数と同じなので、コピー&ペーストでもOKです。

MyString.cpp

|  |
| --- |
| void MyString::operator=(const char\* str)  {  if (m\_string != nullptr) {  //m\_stringにアドレスが入っていたら開放する。  free(m\_string);  }  //コピー元の文字列の長さを調べる。  int len = strlen(str);  //コピーすることが可能なメモリを動的に確保する。  //malloc関数は確保するメモリのサイズを引数に受け取ります。  //戻り値は、確保したメモリのアドレスを返してきます。  m\_string = (char\*)malloc(len + 1);  //文字列を確保したメモリにコピーする。  strcpy(  m\_string, //コピー先のメモリの先頭アドレス。  str //コピー元の文字列。  );  } |

　これで、代入演算子が使えるようになりました。Source.cppを次のように改造してください。

|  |
| --- |
| #include <iostream>  #include "MyString.h"  using namespace std;  int main()  {  char hoge [] = "Hello";  //hogeをstrにコピーする。  MyString str;  //MyString::operator=が呼び出される。  **str = hoge ;**  //"Helloと表示される"。  cout << str.m\_string << "\n";  //"こんにちは 世界"をコピー。  **str = "こんにちは　世界";**  //"こんにちは　世界"と表示される。  cout << str.m\_string << "\n";  return 0;  } |

## **Lesson 9 中間テスト６**

下記のURLの中間テストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSf8Q-n1qp4shPD9aYL2Dm3sxn4qqfvZwuJ1nHnauk9ZmahhoA/viewform?usp=sf_link>

## **9.8.Ex.7.2 HandsOn-2 MyStringクラス(文字列の連結)**

　ここでは、文字列の連結を行う関数を作成してみましょう。文字列の連結は+=演算子を使って行いたいのですが、先ほどの文字列のコピーと同様に、まずはAddというメンバ関数でコピーできるようにしてみましょう。

　まずは、Addメンバ関数の宣言をMyStringクラスに追加します。網掛けになっている箇所を追加してください。

MyString.h

|  |
| --- |
| class MyString  {  public:  　　　　…  　　　　省略  　　　　…  **//文字列の連結を行うメンバ関数。**  **//引き数のstrがm\_stringの後ろに連結される。**  **void Add(const char\* str);**  //文字列の先頭アドレスを記録するためのメンバ変数を追加する。  //一応nullptrで初期化しておく。  char\* m\_string = nullptr;  }; |

続いて、Addメンバ関数の定義を作成しましょう。今度は分からないコードはないはずですが、少々難しいと思います。しっかりと処理を考えて、コードを入力してください。

MyString.cpp

|  |
| --- |
| void MyString::Add(const char\* str)  {  /////////////////////////////////////////////////////  //① 連結後の文字列の長さを計算する。  /////////////////////////////////////////////////////  int len = strlen(str);  //元の文字列の長さを加算する。  len += strlen(m\_string);  /////////////////////////////////////////////////////  //② 新しい文字列を記憶するための、メモリを確保する。  /////////////////////////////////////////////////////  char\* newStr = (char\*)malloc( len + 1 );  /////////////////////////////////////////////////////  //③ 文字列連結  /////////////////////////////////////////////////////  //まず、newStrにm\_stringが指している文字列をコピーする。  strcpy(newStr, m\_string);  //続いて文字列を連結させる。  strcat(newStr,str);  /////////////////////////////////////////////////////  //④ m\_stringが指している古い文字列を解放して、  // 新しい文字列のアドレスを代入する。  /////////////////////////////////////////////////////  　　 //古い文字列のアドレスを解放する。  free(m\_string);  //m\_string新しく確保した  //文字列のアドレスを代入する。  m\_string = newStr;  } |

　では、アルゴリズムの流れを見ていきましょう。下記のPowerPointのファイルを見てください。

ppt/9.8.Ex.7.2 HandsOn-2 MyStringクラス(文字列の連結).pptx

## **9.8.Ex.7.2.1 文字列の連結で+=演算子を使えるようにしてみよう。**

　では、文字列のコピーと同じように、連結も+=演算子を使って行えるように、演算子をオーバーロードしてみましょう。MyStringクラスにoperator+=演算子のメンバ関数宣言を追加しましょう。

MyString.h

|  |
| --- |
| class MyString  {  public:  　　　…  　　　省略  　　　…  //+=演算子をオーバーロード。  void operator+=(const char\* str);  //文字列の先頭アドレスを記録するためのメンバ変数を追加する。  //一応nullptrで初期化しておく。  char\* m\_string = nullptr;  }; |

　続いて、operator+=のメンバ関数定義を追加しましょう。関数の中身はAdd関数と同じなので、コピー&ペーストでもOKです。

MyString.cpp

|  |
| --- |
| void MyString::operator+=(const char\* str)  {  /////////////////////////////////////////////////////  //① 連結後の文字列の長さを計算する。  /////////////////////////////////////////////////////  int len = strlen(str);  //元の文字列の長さを加算する。  len += strlen(m\_string);  /////////////////////////////////////////////////////  //② 新しい文字列を記憶するための、メモリを確保する。  /////////////////////////////////////////////////////  char\* newStr = (char\*)malloc(len + 1);  /////////////////////////////////////////////////////  //③ 文字列連結  /////////////////////////////////////////////////////  //まず、newStrにm\_stringが指している文字列をコピーする。  strcpy(newStr, m\_string);  //続いて文字列を連結させる。  strcat(newStr, str);  /////////////////////////////////////////////////////  //④ m\_stringが指している古い文字列を解放して、  // 新しい文字列のアドレスを代入する。  /////////////////////////////////////////////////////  //古い文字列のアドレスを解放する。  free(m\_string);  //m\_string新しく確保した  //文字列のアドレスを代入する。  m\_string = newStr;  } |

続いて、Source.cppを次のように改造してください。

|  |
| --- |
| #include <iostream>  #include "MyString.h"  using namespace std;  int main()  {  MyString str;  //strに"こんにちは"を代入。  str = "こんにちは";  //strに"世界"を連結。  **str += "世界";**  //"こんにちは世界"と表示される。  cout << str.m\_string;  return 0;  } |

　これで、+=演算子で文字列の連結が行えるようになりました。

## **9.8.Ex.7.3　Hands-On3　MyStringクラス(文字列の比較)**

　では、最後に文字列の比較を行えるようにMyStringクラスを改造しましょう。今回は最初から==演算子をオーバーロードしていきましょう。MyStringクラスに==演算子のメンバ関数宣言を追加してください。

MyString.h

|  |
| --- |
| class MyString  {  public:  　　　…  　　　省略  　　　…  //==演算子をオーバーロード。  bool operator==(const char\* str);  //文字列の先頭アドレスを記録するためのメンバ変数を追加する。  //一応nullptrで初期化しておく。  char\* m\_string = nullptr;  }; |

==演算子は、等しい(true)か等しくないか(false)を返す必要があるので、戻り値をbool型にしています。

では、==演算子のメンバ関数定義を作成します。

MyString.cpp

|  |
| --- |
| bool MyString::operator==(const char\* str)  {  //文字列の比較はstrcmpを使う。  int result = strcmp(m\_string, str);  if (result == 0) {  //strcmpの戻り値が0なら文字列は等しいので、trueを返す。  return true;  }  //ここに来たということは、等しくないということなので  //falseを返す。  return false;  } |

これで、文字列の比較に==演算子が使えるようになりました。Source.cppのメイン関数に下記のコードを入力して、確認してみてください。

Source.cpp

|  |
| --- |
| int main()  {  MyString str;  //strに"こんにちは"を代入。  str = "こんにちは";  //strに"世界"を連結。  str += "世界";    if (str == "こんにちは世界") {  cout << "strは「こんにちは世界」という文字列です\n";  }  return 0;  } |

## **Lesson 9 実習課題\_6**

Lesson 9 実習課題\_6を立ち上げて、MyString.cppの下記の3つのメンバ関数定義を完成させなさい。

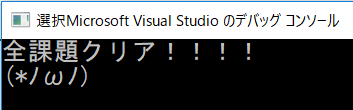
・MyString::Length関数

・MyString::Search関数

・MyString::Insert関数

　関数の仕様はコメントを読んで推測しなさい。

下記の画面が表示されたら課題クリアです。



# **Lesson 10 大規模なプログラムの作成**

## **10.1 変数とスコープ**

ローカル変数のスコープ　＝　変数を宣言した場所から関数の終わりまで(p305の表10)

これは正確には間違った説明です。ローカル変数のスコープは、変数を宣言した場所が属する、波かっこ”{ }”の間です。Lesson\_10を立ち上げて、下記のプログラムSample10.1をmain関数に入力して動作を確認してみてください。

### Sample10.1

|  |
| --- |
| int main()  {  int a = 10;  {  int b = 20;  printf("bの値は%dです。\n", b);  }  printf("aの値は%dです。\n", a);  //bが定義されていないというコンパイルエラーが発生します。  printf("bの値は%dです。\n", b);  return 0;  } |

各変数のスコープは下記のようになっています。

aのスコープ

|  |
| --- |
| int main()  {  int a = 10;  {  int b = 20;  printf("bの値は%dです。\n", b);  }  printf("aの値は%dです。\n", a);  //bが定義されていないというコンパイルエラーが発生します。  printf("bの値は%dです。\n", b);  return 0;  } |

bのスコープ

**ローカル変数の名前が重なると？(p.306)**

同じ関数内のローカル変数には同じ名前を付けることはできません。

これは正しい説明ではありません。正しくは、**「同じスコープ内の変数に同じ名前を付けることはできません」**です。下記のSample.10.2のプログラムを入力してみてください。

### Sample10.2

|  |
| --- |
| int main()  {  int hoge = 10;  {  int hoge = 5;  printf("どっちのhogeの値が表示される？？？？？ hoge = %d\n", hoge);  }  printf("どっちのhogeの値が表示される？？？？？ hoge = %d\n", hoge);  return 0;  } |

このように、同じスコープ内で変数名の重複は許されませんが、異なるスコープ通しの場合は、変数名の重複が許されています。上のコードでは、main関数にhogeという変数が二つ作成されていることになります。そして、このhogeは名前は同じだが、全く別の変数となります。

　異なるスコープに同じ名前の変数がある場合、どの変数が使われるのか？ですが、より近いスコープの変数が使われることとなります。

　下記のプログラムSample10.3を読んで実行結果がどうなるか考えてみてください。

### Sample10.3

|  |
| --- |
| int main()  {  int hoge = 10;  {  int hoge = 5;  printf("どのhogeの値が表示される？？？？？ hoge = %d\n", hoge);  {  int hoge = 20;  printf("どのhogeの値が表示される？？？？？ hoge = %d\n", hoge);  {  printf("どのhogeの値が表示される？？？？？ hoge = %d\n", hoge);  }  }  }  {  printf("どのhogeの値が表示される？？？？？ hoge = %d\n", hoge);  }  printf("どのhogeの値が表示される？？？？？ hoge = %d\n", hoge);  return 0;  } |

## **10.1.ex.1 for文のループ変数のスコープは？？？**

　下記のようなfor文のループ変数もローカル変数です。

|  |
| --- |
| for (int hoge = 0; hoge < 20; hoge++) {  printf(  "hogeの値を10進数で表記すると%d、16進数で表記すると%xになります。\n",  hoge,  hoge  );  } |

では、このプログラムの変数hogeのスコープはどうなっているのでしょうか。実は、このhogeのスコープはfor文の中のみになります。

下記のプログラムSample10.4を入力してみてください。

### Sample10.4

|  |
| --- |
| int main()  {  for (int hoge = 0; hoge < 20; hoge++) {  printf(  "hogeの値を10進数で表記すると%d、16進数で表記すると%xになります。\n",  hoge,  hoge  );  }  //hogeが定義されていないというエラーが出ます。  printf("hogeの値を8進数で表記すると%oです。\n", hoge);  return 0;  } |

　さて、for文の後でループ変数を使いたいというケースはそれなりにあります。そのようなときは下記のようにすればいいのです。Sample10.3を入力して、動作を確認してください。

### Sample10.5

|  |
| --- |
| int main()  {  //ループ変数をfor文の外で定義する。  int hoge = 0;  for (; hoge < 20; hoge++) {  printf(  "hogeの値を10進数で表記すると%d、16進数で表記すると%xになります。\n",  hoge,  hoge  );  }  //これでfor文の外でhogeが使える。  printf("hogeの値を8進数で表記すると%oです。\n", hoge);  return 0;  } |

## **10.2 記憶寿命**

　ローカル変数の記憶寿命は関数が終了するまで(表10-2)

これは正しい説明ではありません。もう分かるかもしれませんが、正確には、スコープを抜けるまでです。

**ローカル変数へのポインタと戻り値(p.314)**

これは上級者になるとあ、めったに侵さない不具合なのですが、少しプログラムが書けるようになってきた中級者はよく起こす不具合です。次のSample.10.6を実行して動作を確認してみてください。

### Sample.10.6

|  |
| --- |
| int\* Func()  {  int a = 10;  //ローカル変数のアドレスを返している！！！  return &a;  }  int main()  {  int\* p = Func();  //funcのローカル変数はすでに削除されているので不正なメモリにアクセスしていることになる。  //でも、期待通りに動いてしまう・・・ときもあるし、クラッシュすることもある。  //とても厄介なバグを生みます。  printf("%d\n", \*p);  return 0;  } |

## **10.2.ex.2 メンバ変数の記憶寿命は？？？**

　メンバ変数の記憶寿命は、インスタンスが破棄されるまでの間です。下記のプログラムをSample.10.7を入力してみてください。

### Sample.10.7

|  |
| --- |
| #define \_CRT\_SECURE\_NO\_WARNINGS //MSのエラーを消すために追加。  #include <iostream>  using namespace std;  class Hoge {  public:  char name[256];  //コンストラクタ。  Hoge()  {  printf("Hogeクラスのインスタンスが作られました。\n");  }  //デストラクタ。  ~Hoge()  {  printf("Hogeクラスのインスタンスの%sが破棄されました。\n", name);  }  void SetName(const char\* n)  {  strcpy(name, n);  }  };  int main()  {  Hoge hoge\_0;  hoge\_0.SetName("hoge\_0");  {  Hoge hoge\_2;  hoge\_2.SetName("hoge\_2");  Hoge hoge\_1;  hoge\_1.SetName("hoge\_1");  }  return 0;  } |

　Hogeクラスのインスタンスがmain関数で複数定義されています。このインスタンスの寿命は下記のようになります。

|  |
| --- |
| int main()  {  Hoge hoge\_0;  hoge\_0.SetName("hoge\_0");  {  Hoge hoge\_2;  hoge\_2.SetName("hoge\_2");  Hoge hoge\_1;  hoge\_1.SetName("hoge\_1");  }**//hoge\_2とhoge\_1はここで死ぬ。**  return 0;  }**//hoge\_0はここで死ぬ。** |

　クラスのインスタンスの記憶寿命は、組み込み型の変数と同じです。クラスのインスタンスが死ぬときに、そのインスタンスのメンバ変数も死にます。

## **Lesson 10 中間テスト 1**

下記のURLの中間テストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSeV4nIZL5CIHAS0K5j0FnbQlfoYomtvrD5b6-0t1nGV_6uvqg/viewform?usp=sf_link>

## **10.3.ex.1 new/deleteとmalloc/free**

　ここまでnew/deleteを利用した、メモリの動的確保について学んできました。さて、[9.8.Ex.7.1.1 メモリの動的確保](#_9.8.Ex.7.1.1_メモリの動的確保)でmalloc/freeを利用したメモリの動的確保を学びました。この２種類の処理は、メモリを確保するという観点からは全く同じ処理になります。Sample10.8とSample10.9を入力して、動作に違いがないことを確認してみてください。

### Sample10.8(new/delete版)

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  //intのポインタ型のpAというローカル変数を定義する。  int\* pA;  //int型の値を記憶できるメモリを確保する。  pA = new int;  //確保したメモリに10を書き込む。  \*pA = 10;  printf("pAが指しているメモリには、%dという値が設定されています。\n", \*pA);  //確保したメモリを開放する。  delete(pA);  return 0;  } |

### Sample10.9(malloc/free版)

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  //intのポインタ型のpAというローカル変数を定義する。  int\* pA;  //int型の値を記憶できるメモリ(4バイト)を確保する。  pA = (int\*)malloc( 4 );  //確保したメモリに10を書き込む。  \*pA = 10;  printf("pAが指しているメモリには、%dという値が設定されています。\n", \*pA);  //確保したメモリを開放する。  free( pA );  return 0;  } |

いかがでしょうか？全く同じ動作になったと思います。

では、new/deleteとmalloc/freeの違いについてみていこうと思います。

## **10.3.ex.2 C++ではnew/deleteを利用する**

new/deleteとmalloc/freeの違いを見る前に、**C++ではnew/deleteを利用する**と覚えておいてください。malloc/freeを使わなければいけない場面のあるのですが、それはとても稀なケースとなります。原則new/deleteを利用すると覚えていてください。

## 10.3.ex.3 new/deleteとmalloc/freeの違い

　new/deleteとmalloc/freeの違いは、**コンストラクトとデストラクタを実行できるかどうか**になります。

　では、次のSample10.10を入力して動作を確認してみてください。

### Sample10.10(new/delte版)

|  |
| --- |
| #define \_CRT\_SECURE\_NO\_WARNINGS //MSのエラーを消すために追加。  #include <iostream>  using namespace std;  class Hoge {  char\* myStr = nullptr;  public:  //コンストラクタ。  Hoge()  {  printf("コンストラクタが呼ばれた。\n");  //要素数20のchar型の配列を確保する。  myStr = new char[20];  //myStrにHelloWorldをコピーする。  strcpy(myStr, "Hello World\n");  }  //デストラクタ。  ~Hoge()  {  printf("デストラクタが呼ばれた。\n");  delete[] myStr;  }  void Print()  {  printf( "myStrには%sという文字列が設定されています。", myStr );  }  };  int main()  {  //Hogeクラスのインスタンスを作成。  Hoge\* pHoge = new Hoge;  pHoge->Print();  delete pHoge;  return 0;  } |

newを使用してインスタンスを作成する際には、インスタンスに必要なメモリを確保するという処理が行われています。では、続いてmalloc/free版のSample10.11を入力して動作を確認してみてください。

### Sample10.11(maloc/free版)

|  |
| --- |
| int main()  {  //Hogeクラスのインスタンスに必要なメモリを確保する。  Hoge\* pHoge = (Hoge\*)malloc( sizeof(Hoge) );  pHoge->Print();  free( pHoge );  return 0;  } |

実行結果はどうなったでしょうか？こちらも[Sample10.10(new/delte版)](#_Sample10.10(new/delte版))と同様に、インスタンスに必要なメモリは確保しています。しかし、コンストラクタが呼ばれていないため、pHoge->Print()の箇所でクラッシュしてしまう人がいたのではないでしょうか。もしくはでたらめな文字列が表示された人もいるかもしれません。

　ではSample10.12のように、インスタンスの作成だけnewにするとどうなるでしょうか？

### Sample10.12

|  |
| --- |
| int main()  {  //Hogeクラスのインスタンスに必要なメモリを確保する。  Hoge\* pHoge = new Hoge;  pHoge->Print();  free( pHoge );  return 0;  } |

クラッシュはしなくなったと思いますが、まだデストラクタが呼ばれていません。このままでは、Hogeクラスのコンストラクタで確保したメモリがリークしてしまいます。

　このように、malloc/freeではコンストラクタとデストラクタが呼び出されません。そのため、C++はメモリの動的確保は、ほとんどの場合でnew/deleteを利用します。

## **Lesson10 中間テスト２**

下記のURLの試験を行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSfpfdBWoXraznpo6NMtacNtd8qsVuQ2b9rCVm6-a8ZbmykrLA/viewform?usp=sf_link>

# **Lesson 11 いろいろな型**

## 11.3.ex.1 enumの典型的な使われ方

　enumは、配列の要素数や添え字(配列のインデックス)を表す定数としてよく活用されています。下記のSample11.1を入力してみてください。

### Sample11.1(enumを配列の要素数、添え字として活用するサンプル)

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  //都市の情報。  struct CityInfo {  string name; //都市の名前。  int population; //人口。  int populationRank; //人口ランキング。  };  //都市を表す列挙  enum {  TOKYO, //TYOKYOという定数には0が入っている。  OOSAKA, //OOSAKAという定数には1が入っている。  FUKUOKA, //FUKUOKAという定数には2が入っている。  EHIME, //EHIMEという定数には3が入っている。  NUM\_CITY, //NUM\_CITYという定数には4が入っている。  };  int main()  {  //NUM\_CITYは4という数値が入っているので、cityInfoは要素数４の配列になる。  CityInfo cityInfo[NUM\_CITY];  //各都市の情報を設定していく。  //まずは東京。  cityInfo[TOKYO].name = "東京";  cityInfo[TOKYO].population = 13843403;  cityInfo[TOKYO].populationRank = 1;  //大阪  cityInfo[OOSAKA].name = "大阪";  cityInfo[OOSAKA].population = 882566;  cityInfo[OOSAKA].populationRank = 3;  //福岡  cityInfo[FUKUOKA].name = "福岡";  cityInfo[FUKUOKA].population = 5111494;  cityInfo[FUKUOKA].populationRank = 9;  //愛媛  cityInfo[EHIME].name = "愛媛";  cityInfo[EHIME].population = 1351510;  cityInfo[EHIME].populationRank = 28;  //都市の情報を出力。  for (int i = 0; i < NUM\_CITY; i++) {  cout << "=====================================\n";  cout << "都市の名前 = " << cityInfo[i].name << "\n";  cout << "人口　　　　 = " << cityInfo[i].population << "\n";  cout << "人口ランキング = " << cityInfo[i].populationRank << "\n";  }  return 0;  } |

　enumは上から順番に1つづ数値が増えていくという特性から、配列の添え字(インデックス)と要素数として、しばしば利用されます。では、これにどのような利点があるのか見ていきましょう。

## **11.3.ex.2 enumを配列で利用するメリット**

　では、先ほどのプログラムでenumを使わずに書いてみましょう。。Sample11.2を入力してみてください。(Sample11.1のプログラムは後で利用するので、別プロジェクトを作って、Sample11.2を入力してみてください。)

### Sample11.2

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  //都市の情報。  struct CityInfo {  string name; //都市の名前。  int population; //人口。  int populationRank; //人口ランキング。  };  int main()  {  CityInfo cityInfo[4];  //各都市の情報を設定していく。  //まずは東京。  cityInfo[0].name = "東京";  cityInfo[0].population = 13843403;  cityInfo[0].populationRank = 1;  //大阪  cityInfo[1].name = "大阪";  cityInfo[1].population = 882566;  cityInfo[1].populationRank = 3;  //福岡  cityInfo[2].name = "福岡";  cityInfo[2].population = 5111494;  cityInfo[2].populationRank = 9;  //愛媛  cityInfo[3].name = "愛媛";  cityInfo[3].population = 1351510;  cityInfo[3].populationRank = 28;  //都市の情報を出力。  for (int i = 0; i < 4; i++) {  cout << "=====================================\n";  cout << "都市の名前 = " << cityInfo[i].name << "\n";  cout << "人口　　　　 = " << cityInfo[i].population << "\n";  cout << "人口ランキング = " << cityInfo[i].populationRank << "\n";  }  return 0;  } |

動作に違いはありません。しかし、保守性、可読性といった面で大きな違いが生まれています。

　では、Sample11.2のコードを改造して、次のような仕様を実装してみてください。

新しく京都の情報を表示できるようにする。京都の情報は大阪の後ろ、福岡の前に表示できるようにしてください。

京都の人口は2,591,779人、人口ランキングは13位です。

**次のページに解答例を載せていますが、まずは自分で考えてみてください。**

**では、先ほどの課題の解答例を下記に記載します。**

**解答例**

都市の数が増えたのでループの回数も+1

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  //都市の情報。  struct CityInfo {  都市の数が増えたので要素数を+1  string name; //都市の名前。  int population; //人口。  int populationRank; //人口ランキング。  };  int main()  {  CityInfo cityInfo[**5**];  //各都市の情報を設定していく。  //まずは東京。  cityInfo[0].name = "東京";  cityInfo[0].population = 13843403;  cityInfo[0].populationRank = 1;  京都の情報を追加  //大阪  cityInfo[1].name = "大阪";  cityInfo[1].population = 882566;  cityInfo[1].populationRank = 3;  **cityInfo[2].name = "京都";**  **cityInfo[2].population = 2591779;**  **cityInfo[2].populationRank = 13;**  //福岡  京都を間に入れたので、1ずつ後ろにずらす  cityInfo[**3**].name = "福岡";  cityInfo[**3**].population = 5111494;  cityInfo[**3**].populationRank = 9;  //愛媛  cityInfo[**4**].name = "愛媛";  cityInfo[**4**].population = 1351510;  cityInfo[**4**].populationRank = 28;  //都市の情報を出力。  for (int i = 0; i < **5**; i++) {  cout << "=====================================\n";  cout << "都市の名前 = " << cityInfo[i].name << "\n";  cout << "人口　　　　 = " << cityInfo[i].population << "\n";  cout << "人口ランキング = " << cityInfo[i].populationRank << "\n";  }  return 0;  } |

**仕様追加に対応するために、いくつかの点で修正が必要になっています。では、この仕様追加を**[Sample11.1(enumを配列の要素数、添え字として活用するサンプル)](#_Smaple11.1(enumを配列の要素数、添え字として活用するサン)**のプログラムに対して行うとどうなるでしょうか？**

**では、今度はSample11.1を改造して、京都の情報を追加してみてください。**

**では、先ほどの課題の解答例を下記に記載します。**

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  KYOTOという定数を追加  //都市の情報。  struct CityInfo {  string name; //都市の名前。  int population; //人口。  int populationRank; //人口ランキング。  };  //都市を表す列挙  enum {  TOKYO, //TYOKYOという定数には0が入っている。  OOSAKA, //OOSAKAという定数には1が入っている。  **KYOTO**, //KYOTOという定数には2が入っている。  FUKUOKA, //FUKUOKAという定数には3が入っている。(enumの特性で自動的に+1される)  EHIME, //EHIMEという定数には4が入っている。(enumの特性で自動的に+1される)  NUM\_CITY, //NUM\_CITYという定数には5が入っている。(enumの特性で自動的に+1される)  };  int main()  {  //NUM\_CITYは4という数値が入っているので、cityInfoは要素数４の配列になる。  CityInfo cityInfo[NUM\_CITY];  //各都市の情報を設定していく。  //まずは東京。  cityInfo[TOKYO].name = "東京";  cityInfo[TOKYO].population = 13843403;  cityInfo[TOKYO].populationRank = 1;  //大阪  cityInfo[OOSAKA].name = "大阪";  cityInfo[OOSAKA].population = 882566;  cityInfo[OOSAKA].populationRank = 3;  京都の情報を追加  **cityInfo[KYOTO].name = "京都";**  **cityInfo[KYOTO].population = 2591779;**  **cityInfo[KYOTO].populationRank = 13;**  //福岡  cityInfo[FUKUOKA].name = "福岡";  cityInfo[FUKUOKA].population = 5111494;  cityInfo[FUKUOKA].populationRank = 9;  //愛媛  cityInfo[EHIME].name = "愛媛";  cityInfo[EHIME].population = 1351510;  cityInfo[EHIME].populationRank = 28;  //都市の情報を出力。  for (int i = 0; i < NUM\_CITY; i++) {  cout << "=====================================\n";  cout << "都市の名前 = " << cityInfo[i].name << "\n";  cout << "人口　　　　 = " << cityInfo[i].population << "\n";  cout << "人口ランキング = " << cityInfo[i].populationRank << "\n";  }  return 0;  } |

**配列の添え字と要素数に列挙を使った場合、修正が必要な箇所が格段に減りました。コーディングを行う量が減るということは、ヒューマンエラーが減るということです。**

**今回は短いプログラムだったので、そこまで恩恵を感じていないかもしれませんが、例えばcityInfoの要素数が30ほどあり、10番目に新しい要素を追加する場合を考えてみて下さい。なかなかに苦痛ではないですか？**

## **11.3.ex.2.1 可読性の向上**

では、最後にenumを利用する場合の可読性の向上について見ていきましょう。

enumを使っていない下記のコードを見てみてください。

|  |
| --- |
| cout << "人口　　　　 = " << cityInfo[10].population << "\n"; |

さて、どこの都道府県の人口を表示しようとしているか分かりますか？

では、enumを使っているコードを見てみましょう。

|  |
| --- |
| cout << "人口　　　　 = " << cityInfo[KAGAWA].population << "\n"; |

さて、今度はどうでしょうか？プログラムを見ただけで、香川県の人口を表示していることが分かったのではないでしょうか？これが可読性の高いプログラムです。

可読性の低いプログラムを書いているプログラマは、数か月後に呪文のようなプログラムに苦しめられて、残業の毎日になることでしょう。

Tips

|  |
| --- |
| enumは下記のように、数値を設定することができます。  //都市を表す列挙  enum {  TOKYO,  OOSAKA,  KYOTO,  FUKUOKA,  EHIME = 10,  KAGAWA,　//KAGAWAは11  KOUTI //KOUTIは12になる。  };  このように記述すると、EHIMEから下は10から1ずつ増えていくことになります。 |

# **Lesson 12 クラス(発展)**

## **12.2 メンバへのアクセス制限**

　テキストを進める前にCarクラスを作成しましょう。VisualStudioで新規プロジェクトを作成して、下記のようにCarクラスを作成してください。

Car.h

|  |
| --- |
| #pragma once  class Car  {  public:  Car();  ~Car();  void show();  int num; //車のナンバー。  double gas; //ガソリンの残量。  }; |

Car.cpp

|  |
| --- |
| #include <iostream>  #include "Car.h"  using namespace std;  Car::Car()  {  }  Car::~Car()  {  }  void Car::show()  {  cout << "車のナンバーは" << num << "です。\n";  cout << "ガソリンの量は" << gas << "です。\n";  } |

では、Carクラスを利用するプログラムをmain関数に記述してみましょう。

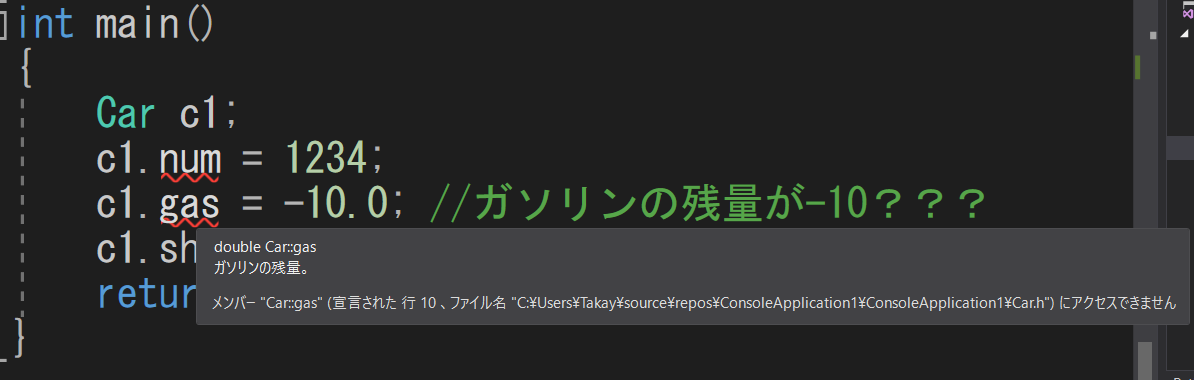
|  |
| --- |
| int main()  {  Car c1;  c1.num = 1234;  c1.gas = -10.0; //ガソリンの残量が-10？？？  c1.show();  return 0;  } |

**privateメンバを作る**

では、Carクラスのヘッダーファイルを下記のように書き換えてください。

|  |
| --- |
| #pragma once  class Car  {  public:  Car();  ~Car();  void show();  **private: //ここから下に記述されたメンバ変数、関数はprivateになる。**  int num; //車のナンバー。  double gas; //ガソリンの残量。  }; |

こうすることで、numとgasのアクセス指定子はprivateとなり、外部からアクセスできなくなります。main関数で下記のようなエラーが起きるようになっているはずです。



**publicメンバを作る**

では、下記のようにCarクラスを変更してください。

Car.h

|  |
| --- |
| #pragma once  class Car  {  public:  Car();  ~Car();  void show();  **//ナンバーを設定する関数を追加。**  **void setNum(int n);**  **//ガソリンの残量を設定する関数を追加。**  **void setGas(double g);**  private: //ここから下に記述されたメンバ変数、関数はprivateになる。  int num; //車のナンバー。  double gas; //ガソリンの残量。  }; |

Car.cpp

|  |
| --- |
| #include "stdafx.h"  #include <iostream>  #include "Car.h"  using namespace std;  Car::Car()  {  }  　　…  　　省略  　　…  **void Car::setNum(int n)**  **{**  **if (n >= 0 && n < 9999) {**  **//車のナンバーは0～9999まで**  **num = n;**  **}**  **else {**  **cout << "車のナンバーに不適切な値が渡されました。\n";**  **}**  **}**  **void Car::setGas(double g)**  **{**  **if (g >= 0 && g < 60) {**  **//ガソリン残量は0～60まで**  **gas = g;**  **}**  **else {**  **cout << "ガソリンの残量に不適切な値が渡されました。\n";**  **}**  **}**  　　…  　　省略  　　… |

では、最後にmain関数を下記のように書き換えてみてください。

|  |
| --- |
| int main()  {  Car c1;  c1.setNum(1234);  c1.setGas(40.0);    c1.show();  //正しくないガソリン量を設定してみる。  c1.setGas(-10.0); //ガソリンの残量が-10？？？  c1.show();  return 0;  } |

**カプセル化の仕組みを知る**

**重要！！！**

**メンバ変数は原則private！！！！**

**publicメンバ変数はダメ！絶対！！！**

## **12.2.ex\_1 privateなメンバ変数の値を参照する**

　前節で学んだように、privateなメンバ変数の値を変更したい場合は、publicなメンバ関数を追加しました。では、外部からprivateなメンバ変数の値を参照したくなった場合はどうすればいいのでしょうか？例えば下記のようなケースです。

**「プレイヤーのHPが100より小さくなったら、エネミーは総攻撃をする」**

このような仕様を実装する場合は下記のようなコードを記述すると思います。

|  |
| --- |
| void Enemy::Update()  {  //hpがprivateメンバ変数なのでアクセスできない！！！  if (g\_player.hp < 100) {  コンパイルエラーが起きてしまう！！！  Attack();  }  } |

このような場合はPlayerクラスに下記のようなpublicなメンバ関数を追加することで解決できます。

Player.h

|  |
| --- |
| class Player {  private:  int hp;  …  　　　省略  　　　…  public:  　　　…  　　　省略  　　　…  int GetHP();  }; |

Player.cpp

|  |
| --- |
| int Player::GetHP()  {  return hp;  } |

先ほどのEnemyのコードは下記のように書き換えることができます。

|  |
| --- |
| void Enemy::Update()  {  if (g\_player.GetHP() < 100) {  Attack();  }  } |

## **実習課題\_2**

　①　ハンズオン

　　　プレイヤークラスをカプセル化してみよう。

　②　実習

　　　Enemyクラスをカプセル化してみよう。

メンバ関数をインライン関数にする

　関数呼び出しには微々たるものですが**オーバーヘッド**(関数を呼び出すための準備時間のようなものだと考えてください)が発生します。このオーバーヘッドは本当に微々たるもので、多くのアプリケーションでは無視できるものです。しかし、ゲームは高いリアルタイム性と、高いパフォーマンスが求められるアプリケーションです。なので、オーバーヘッドを除去できるインライン関数は積極的に利用していってほしいのですが、実はインライン関数を使えば必ず速くなるというわけではなく、逆に遅くなるケースも存在します。

　そこで、インライン関数を利用すべきルールを教えます。下記を頭に叩き込んでインライン化を行うようにしてください。

　・**短い関数(3行以内)だけインライン化！！！！**

　今は理由は教えませんが、このルールを徹底するようにしてください。ちなみに３行以内というのは目安です。

## **実習課題\_3**

①　ハンズオン

　　　プレイヤークラスをカプセル化してみよう。(インライン関数を利用するように)

　②　実習

　　　Enemyクラスをカプセル化してみよう。(インライン関数を利用するように)

## **中間テスト\_1**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSc_oj8BedpFmoIH3Ap2r2x2x76uI2xbvWQCh7-Im5CqAELASA/viewform?usp=sf_link>

## **12.ex.1 関数のオーバーロード**

　関数は引数が異なっていれば、同じ名前を付けることができます。これを関数のオーバーロードといいます。

下記のSample12.1を入力してみてください。

### Sample12.1

|  |
| --- |
| #include <iostream>  using namespace std;  class Hoge {  public:  void Draw( int value )  {  cout << value << "\n";  }  void Draw(float value)  {  cout << value << "\n";  }  void Draw(int value\_0, int value\_1)  {  cout << value\_0 << ":" << value\_1 << "\n";  }  };  int main()  {  Hoge hoge;  //void Draw( int value )が呼び出される。  hoge.Draw(10);  //void Draw( float value )が呼び出される。  hoge.Draw(12.5f);  //void Draw( int value\_0, int value1 )が呼び出される。  hoge.Draw(30, 40);  return 0;  } |

このように、すべてDrawという関数名ですが、渡される引数に応じて適切な関数が呼び出されています。この仕組みを関数のオーバーロードといいます。

## **12.3 引数とオブジェクト**

**重要！**

　クラスのインスタンス(オブジェクト)のコピーは一部の例外を除いて行ってはいけないと考えてください。教科書ではパフォーマンスの問題が起きると説明されていますが、ほとんどすべてのクラスで、インスタンスのコピーは予期しない不具合を起こします。

　次のSample12.2を入力して実行してみてください。

### **Sample12.2**

|  |
| --- |
| #include <iostream>  using namespace std;  //3Dモデルを描画するクラスの疑似コード。  class SkinModelRender {  int indexBuffer[6]; //インデックスバッファ  public:  SkinModelRender()  {  //適当にインデックスバッファに値を埋める。  //特にこれに意味はない。  for (int i = 0; i < 6; i++) {  indexBuffer[i] = i;  }  }  //3Dモデルを描画する関数。  void Draw()  {  cout << "プレイヤーの描画すたーと\n";  for (int i = 0; i < 6; i++) {  cout << indexBuffer[i] << "\n";  }  cout << "プレイヤーの描画しゅうりょう\n";  }  };  //プレイヤークラスの疑似コード。  class Player {  SkinModelRender\* skinModelRender;  public:  Player()  {  //スキンモデルレンダーのインスタンスを作成。  skinModelRender = new SkinModelRender;  }  ~Player()  {  //ちゃんと後始末を行う。  delete skinModelRender;  }  void Draw()  {  //プレイヤーの絵を描画。  skinModelRender->Draw();  p2.skinModelRenderにp1.skinMoelRenderがコピーされる。  **両方とも同じアドレスを指している！！！**  }  };  //メイン関数。  int main()  {  Player p1; //プレイヤー１  //ここでははちゃんと実行できる。  p1.Draw();  {  p2の寿命が尽きるので、デストラクタが呼ばれて、delete p2.skinModelRenderが実行される。  //プレイヤー２にプレイヤー１をコピーする。  Player p2 = p1;  }  //プログラムがクラッシュする！！！！  p1.Draw();  return 0;  } |

このプログラムでは、ダンブリングポインタ(削除済みのアドレスにアクセスしてしまうポインタ)による問題が発生しています。

　クラスのインスタンスのコピーは、ほとんどの場合でダンブリングポインタの問題を引き起こすので、現時点ではインスタンスのコピーはNGだと覚えておきましょう。

|  |
| --- |
| Tips  クラスのインスタンスのコピーは慎重に実装する必要があります。(そしてそれは大変難しい)  ほとんどの場合でNGですが、君たちが使用しているtkEngineの数学系のクラス(CVector3、CQuaternion、CMatrixなど)はコピーOKです。  それ以外は、原則コピーしてはいけないと考えてください。 |

## **実習課題\_4**

Lesson\_12\_実習課題４のプログラムは2回目のDrawPlayer関数の呼び出しでクラッシュが発生してしまいます。このクラッシュを修正してください。

|  |
| --- |
| //メイン関数。  int main()  {  Player p1; //プレイヤー１  //ここでははちゃんと実行できる。  DrawPlayer (p1);  DrawPlayer (p1); //ここでクラッシュが発生する！  return 0;  } |

また、DrawPlayer関数は引数で渡された、PlayerクラスのインスタンスのDraw関数を呼び出すという機能を実装しています。この機能を変更することなく修正してください。

## **中間テスト\_2**

次のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSdjVIdDCUPUyGE6meOrvual3jRJ9Z2UtCtuoMR6E_WfCunZcQ/viewform?usp=sf_link>

# Lesson 13 クラスの機能

## 13.1 コンストラクタの基本

　C++では、初期化されていない変数に入っている値は未定義となっています。つまりどんな数値が入っているか分からないのです。

　変数の未初期化は修正困難な不具合を発生させます。「時々ゲームがクラッシュしてしまいます」とか「時々、ゲームの表示がおかしくなります」などの、たまーーーーに発生する不具合は8割がた未初期化変数によるものです。

初期化忘れを防ぐために、インスタンスを生成したら、自動的に呼ばれるコンストラクタで適切な初期化を行いましょう。

## 13.2 コンストラクタのオーバーロード

コンストラクタを省略すると

　下記のSample13.1を入力してください。

### Sample13.1

|  |
| --- |
| #include <iostream>  class Car {  int m\_gas;  int m\_number;  public:  void SetGasAndNumber(int gas, int number);  void Show();  };  void Car::SetGasAndNumber(int gas, int number)  {  m\_gas = gas;  m\_number = number;  }  void Car::Show()  {  std::cout << m\_gas << "\n";  std::cout << m\_number << "\n";  }  int main()  {  //Carクラスのインスタンスを生成する。  //空のデフォルトコンストラクタが呼ばれる。  Car car;  //ガソリンとナンバーを設定する。  car.SetGasAndNumber(10, 200);  //ガソリンとナンバーを表示する。  car.Show();  } |

　このCarクラスにはコンストラクタがないので、コンパイラが空のデフォルトコンストラクタを作成してくれています。コンパイラが生成するコードのイメージは下記のようなものです。

これが追加されている。

|  |
| --- |
| class Car {  int m\_gas;  int m\_number;  public:  **Car() {}**  void SetGasAndNumber(int gas, int number);  void Show();  }; |

　デフォルトコンストラクタは、コンストラクタを定義すると用意されなくなります。Sample13.1に引数付きのコンストラクタを追加してみてください。

### Sample13.2

|  |
| --- |
| #include <iostream>  class Car {  int m\_gas;  int m\_number;  public:  **Car(int gas, int number);**  void SetGasAndNumber(int gas, int number);  void Show();  };  **Car::Car(int gas, int number)**  **{**  **m\_gas = gas;**  **m\_number = number;**  **}**  void Car::SetGasAndNumber(int gas, int number)  {  m\_gas = gas;  m\_number = number;  }  void Car::Show()  {  std::cout << m\_gas << "\n";  std::cout << m\_number << "\n";  }  int main()  ここでコンパイルエラーが起きるようになる。  {  //Carクラスのインスタンスを生成する。  //空のデフォルトコンストラクタが呼ばれる。  Car car;  //ガソリンとナンバーを設定する。  car.SetGasAndNumber(10, 200);  //ガソリンとナンバーを表示する。  car.Show();  } |

　Carクラスのインスタンスを定義している箇所で、「**デフォルトコンストラクタがありません**」といったエラーが起きるようになりました。これは引数付きのコンストラクタを定義したため、デフォルトコンストラクタが用意されなくなったためです。

## **13.ex.1 デフォルトメンバ初期化子**

　C++11から、メンバ変数の初期化を宣言時に行うことができるようになりました。

下記のSample13.3を入力してください。

### Sample13.3

|  |
| --- |
| #include <iostream>  class Car {  int m\_gas = 40;  int m\_number = 1000;  public:  void Show();  };  void Car::Show()  {  std::cout << m\_gas << "\n";  std::cout << m\_number << "\n";  }  int main()  {  //Carクラスのインスタンスを生成する。  Car car;  //ガソリンとナンバーを表示する。  car.Show();  } |

使いやすい初期化の仕方を利用してみて下さい。

# **13.ex.2 Singletonパターン**

　静的メンバ変数の活用として、Singletonパターンという設計を勉強してみましょう。Singletonパターンが提供する機能は下記の二つになります。

**・インスタンスを一つしか作れなくする。**

**・グローバルなアクセスポイントを提供する。**

## インスタンスを一つしか作れなくする

　これは、例えばインスタンスを複数作られると問題が起きるクラスでは、インスタンスの生成を一つに制限できるようにすると、ヒューマンエラーを回避することができます。

　これまで君たちが作成してきたゲームであれば、GameクラスやBackgroundクラス、GameCameraクラスなどがそれにあたります。

## **グローバルなアクセスポイントを提供する**

　これは単純にグローバル変数を用意するというものと同じです。プログラムのどこからでもインスタンスにアクセスできるようにします。

## **典型的なSingletonパターンの実装方法**

　下記のSample13.4を入力して、典型的なSingletonパターンを実装してみましょう。

### Sample13.4

|  |
| --- |
| #include <iostream>  //Gameクラス。  class Game {  static Game\* m\_instance; //唯一のインスタンス  public:  Game();  ~Game();  void Show()  {  std::cout << "Game::Show\n";  }  static Game\* GetInstance();  };  //唯一のインスタンスのアドレスを記録するポインタ変数。  Game\* Game::m\_instance = nullptr;  //コンストラクタ。  Game::Game()  {  if (m\_instance != nullptr) {  //Gameクラスのインスタンスはすでに作られている。  //恐らくプログラムの間違いなので、クラッシュする。  std::abort();  }  //このインスタンスを唯一のインスタンスとして記録する。  m\_instance = this;  }  //デストラクタ。  Game::~Game()  {  //インスタンスが破棄されたので、nullptrを代入する。  m\_instance = nullptr;  }  //インスタンスの取得。  Game\* Game::GetInstance()  {  return m\_instance;  }  //メイン関数。  int main()  {  Game game; //インスタンスを作成する。  //Game game2; //二つ目のインスタンスは作れないのでクラッシュする！  //GameクラスのインスタンスはGetInstance関数を使えば  //どこからでもアクセスできる。  Game::GetInstance()->Show();  } |

　Singletonパターンの肝は静的メンバ変数のm\_instanceがインスタンスのアドレスを記録していることです。

## **Hands-On Singletonを実装してみる。**

では、Lesson\_13\_HandsOnを利用して、Singletonパターンの実装してみましょう。

今回のハンズオンでは、サンプルのGameクラスをリファクタリングして、Singletonパターンに変更します。

　まず、Gameクラスに下記のような静的メンバ変数を追加します。

Game.cpp

|  |
| --- |
| class Game : public IGameObject  {  　　　　…  　　　　省略  　　　　…  **static Game\* m\_instance;**  }; |

　続いて、インスタンスを取得するための静的メンバ関数を追加します。

Game.h

|  |
| --- |
| class Game : public IGameObject  {  public:  //メンバ関数。  Game();  ~Game();  bool Start() override;  void Update() override;    **static Game\* GetInstance()**  **{**  **return m\_instance;**  **}**  …  　　　省略  　　　…  }; |

続いて、**インスタンスの作成を一つに制限する**ためのコードを追加します。

Game.cppに下記の網掛けのコードを追加してください。

Game.cpp

|  |
| --- |
| //静的メンバ変数を定義する。  Game\* Game::m\_instance = nullptr;  Game::Game()  {  if (m\_instance != nullptr) {  //すでにインスタンスを作っている。  std::abort();  }  m\_instance = this;  }  Game::~Game()  {  m\_instance = nullptr;  } |

これでGameクラスはSingletonパターンにリファクタリングされました。

　では、Gameクラスのインスタンスにアクセスしている箇所を、Game::GetInstance()を利用するように変更してみましょう。

Enemy.cpp

|  |
| --- |
| bool Enemy::Start()  {  …  　　省略  　　 …  **//m\_game = FindGO<Game>("Game"); FindGOは使わないようにしてみる。**  **m\_game = Game::GetInstance();**  return true;  }  …  　　省略  　　 …  void Enemy::SearchPlayer()  {  //敵からプレイヤーに向かうベクトルを計算。  CVector3 toPlayer = m\_player->GetPosition() - m\_position;  //続いて、toPlayerを可視化してみる。  /\*dbg::DrawVector(  toPlayer,  m\_position);\*/  if (toPlayer.Length() < 400.0f) {  //視野角を判定。  toPlayer.Normalize();  float angle = acosf(toPlayer.Dot(m\_forward));  if (fabsf(angle) < CMath::PI \* 0.25f) {  //視野角45度以内。  //ゲームにゲームオーバーを通知。  **//Game\* game = FindGO<Game>("Game"); FindGOは使用しない**  **Game\* game = Game::GetInstance();**  game->NotifyGameOver();  m\_isFindPlayer = true;  }  }  } |

　FindGOを利用しなくても、インスタンスを取得でき、正しくゲームオーバーが実行されていることを確認してください。(敵兵に見つかるとゲームオーバーです。)

## **中間テスト**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSeVufF-EPc9DOfFOXw1v4fP0KtOzrnXZXOvm5bNKL1BeYjpvA/viewform?usp=sf_link>

## **Lesson 14 可変長配列**

大きさを変えることができる配列です。ここまでに君たちが勉強したのは固定長配列。大きさが固定の配列です。可変長配列はプログラム実行中に大きさを変更することができる配列です。

例) 固定長配列

|  |
| --- |
| int hoge[3] = { 0, 3, 4 }; //この配列の大きさは3。  //プログラム実行中に変更することはできない。 |

可変長配列は配列の大きさをプログラムの実行中に変更することができます。

例)可変長配列

|  |
| --- |
| std::vector< int > hoge; //可変長配列。  hoge.push\_back( 0 ) ; //これで大きさ１  hoge.push\_back( 1 ); //これで大きさ２  hoge.push\_back( 2 ) ; //これで大きさ３ |

変数hogeの型はstd::vector<int>です。std::vector<int>はクラスで、push\_backはメンバ変数です。push\_back関数を呼び出すことで配列の大きさが一つずつ大きくなっていきます。

## 14.1 どういうときに使うの？

　プログラムを書いているときには大きさが分からない要素をプログラミングしているとき。例えば、スポーツジムの会員登録システムなど。

**実習**

VisualStudioでプロジェクトを作成して、下記のプログラムを入力してみてください。

|  |
| --- |
| #include <iostream>  #include <vector> //std::vectorを使うためにはインクルードが必要。  #include <string>  using namespace std;  int main()  {  vector<string> menberNames; //会員の名前の可変長配列。  while (true) {  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  cout << "会員登録を行います。\n名前を入力してください。\n";  string name;  //ユーザーに名前を入力させる。  cin >> name;  //入力された名前を可変長配列に追加する。  menberNames.push\_back( name );  //会員の情報を出力する。  cout << "会員の人数は" << menberNames.size() << "です。\n";  for (int i = 0; i < menberNames.size(); i++) {  cout << " 会員No:" << i << "\n";  cout << " 名前は" << menberNames[i] << "\n";  }  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";  }  } |

size関数は可変長配列のサイズを取得できます。可変長配列は普通の配列のように使うことができ、添え字演算子を利用することで配列の要素にアクセスすることができます。

## 14.2 std::vectorで文字列以外の可変長配列を作るには？

　固定長配列は、文字列以外にも整数、浮動小数点、クラスのインスタンスなどいろいろな型で作ることができます。

|  |
| --- |
| int iHoge[3];  float fHoge[3];  string strHoge[3];  Player playerHoge[3]; |

　可変長配列も固定長配列と同様に、文字列以外の型でも作ることができます。

|  |
| --- |
| vector<int> iHoge;  vector<float> fHoge;  vector<string> strHoge[3];  vector<Player> playerHoge[3]; |

**実習**

会員の年齢も記録できるようにしてみましょう。黄色い網掛けの箇所が追加されたコードです。

|  |
| --- |
| #include <iostream>  #include <vector> //std::vectorを使うためにはインクルードが必要。  #include <string>  using namespace std;  int main()  {  vector<string> menberNames; //会員の名前の可変長配列。  vector< int > menberAges; //会員の年齢の可変長配列。  while (true) {  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  cout << "会員登録を行います。\n名前を入力してください。\n";  string name;  //ユーザーに名前を入力させる。  cin >> name;  //入力された名前を可変長配列に追加する。  menberNames.push\_back( name );  //年齢を入力させる。  int age;  cout << "年齢を入力してください。\n";  cin >> age;  menberAges.push\_back( age );  //会員の情報を出力する。  cout << "会員の人数は" << menberNames.size() << "です。\n";  for (int i = 0; i < menberNames.size(); i++) {  cout << " 会員No:" << i << "\n";  cout << " 名前は" << menberNames[i] << "\n";  cout << " 年齢は" << menberAges[i] << "\n";  }  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";  }  } |

## 14.3 イテレーター

　固定長配列は下記のようにポインタを使って各要素にアクセスすることができました。

|  |
| --- |
| int hoge[10] = { 10, 20, 30};  int\* p = hoge; //配列のhogeの先頭アドレスをpに代入  for( int i = 0; i < 10; i++ ){  cout << \*p << “\n”;  p++; //次の要素へ } |

このプログラムを実行すると下記のように表示されます。

*10*

*20*

*30*

このintのポインタ型に該当するのがイテレーターです。

**実習**

次のプログラムを入力してみてください。

|  |
| --- |
| #include <iostream>  #include <vector> //std::vectorを使うためにはインクルードが必要。  using namespace std;  int main()  {  vector<int> hoge;  hoge.push\_back( 10 );  hoge.push\_back( 20 );  hoge.push\_back( 30 );  vector<int>::iterator it = hoge.begin(); //hogeの先頭イテレータを取得。  for (int i = 0; i < 3; i++) {  cout << \*it << "\n";  it++; // 次の要素へ  }  } |

vector<int>::iteratorがint\* pに該当します。可変長配列でポインタと同様のことをしたい場合はイテレーターを使います。

## 1.4.4 イテレーターって何に使うの？？？

　イテレーターは色々なケースで利用するのですが、一番使われる頻度が高いのは、要素を削除するときだと思います。std::vectorには要素を削除するerase関数があるのですが、この関数が引数に削除したい要素のイテレーターを受け取るようになっています。

**実習**

下記のプログラムを入植してみてください

|  |
| --- |
| #include <iostream>  #include <vector> //std::vectorを使うためにはインクルードが必要。  using namespace std;  int main()  {  vector<int> hoge;  hoge.push\_back( 10 );  hoge.push\_back( 20 );  hoge.push\_back( 30 );  vector<int>::iterator it = hoge.begin(); //hogeの先頭イテレータを取得。  for (int i = 0; i < 3; i++) {  if ( \*it == 20 ) {  //20の要素を検索して削除  hoge.erase( it );  break;  }  it++; // 次の要素へ  }  } |

## 14.5 中間テスト

https://forms.office.com/Pages/ResponsePage.aspx?id=DQSIkWdsW0yxEjajBLZtrQAAAAAAAAAAAAMAANOMgpZURFZPWEc1T0FETDdUSjZDUlQ3VEw2T1hDNS4u

# **Lesson 14 いろいろなクラス**

# **14.ex.1 より良い設計を求めて**

### **14.ex.1.1　クラスの責任を考える**

より良いクラスの設計を考えるときに、クラスの責任について考えていくのは良い習慣です。例えば、下記のような責任を持っているPlayerクラスを作成していることを考えてみてください。

**① Playerクラスでは3Dモデルの表示を行う責任がある。**

**② Playerクラスでは3D空間で背景にめり込むことなく移動を行う責任がある。**

**③ Playerクラスでは状態(走り状態、歩き状態、攻撃状態、ダメージ状態など)に応じて**

**適切な行動を行う責任がある。**

これらの責任をすべてPlayerクラスに実装した場合、Playerクラスはとても大きなクラスとなってしまいます。このようなクラスは何でもできる神クラスを揶揄して、Godクラスと呼ばれたりもします。そして、一般的に責任の多すぎるクラスというのは、プログラムは読みにくく、不具合が起きやすく、拡張のしづらいクラスとなります。

　このようなたくさんの責任を実装する必要が出てきた場合は、新しいクラスを作成して、責任を委譲するのが良い選択となります。

　例えば、学内ゲームエンジンtkEngineであれば、①と②の責任はCSkinModelRenderクラスと、CCharacterControllerクラスに委譲することができます。

**① Playerクラスは3Dモデルの表示を行う責任がある。**

　　 →CSkinModelRenderに委譲する。

**② Playerクラスでは3D空間で背景にめり込むことなく移動を行う責任がある。**

→CCharacterControllerクラスに委譲する。

　責任が明確になっているプログラムは読みやすく、保守性の高いクラスとなっていきます。クラスが大きくなってきたと思ったら、新しくクラスを作成して、責任の委譲を検討してみましょう。

### **14.ex.1.2 凝集度と結合度**

**高い凝集度で低い結合度**となっているクラスは、良い設計であると言えます。そして、これは14.ex.1.1のクラスの責任を考えて設計を行っていくと、おのずと達成できるようになります。

**低い凝集度**

　機能が大きく異なる複数の分野での多くの作業の責任を1つのクラスが単独で行う。

　　→Godクラス

**高い凝集度**

クラスが1つの機能分野で適度な責任を持ち処理がクラス内で完結している。

**低い結合度**

他のクラスとの依存関係が低く、処理が自身のクラス内で完結している。

→他のクラスの修正による影響を受けにくい。

**高い結合度**

他のクラスとの依存関係が高く、処理が自身のクラス内で完結していない。

　　→他のクラスの修正による影響を受けやすい。

凝集度が高いクラスは、おのずと結合度が低くなっていく傾向にあります。

**中間テスト**

[**https://forms.office.com/Pages/ResponsePage.aspx?id=DQSIkWdsW0yxEjajBLZtrQAAAAAAAAAAAAMAANOMgpZUODhCMDFPNE5HS04wQkQ5RjQ2NkJCMFVKRC4u**](https://forms.office.com/Pages/ResponsePage.aspx?id=DQSIkWdsW0yxEjajBLZtrQAAAAAAAAAAAAMAANOMgpZUODhCMDFPNE5HS04wQkQ5RjQ2NkJCMFVKRC4u)

### **14.ex.1.3 インターフェースに対するプログラミング**

　インターフェースに対するプログラミングを行っていくと、凝集度が高く、結合度が低い設計になっていきます。インターフェースに対するプログラミングは「**実装の詳細を気にしないプログラミングである**」と言えます。

　では、いくつかのデザインパターンを例にこれを考えてみましょう。

### 14.ex.1.4 UpdateMethodパターン

#### 概要

　登録されたインスタンスの更新関数が自動的に呼び出される。

#### クラス図

GameObjectManager

+Update

Enemy

+Update

GameCamera

+Update

Player

+Update

IGameObject

+Update

GameObjectManagerクラスはIGameObjectのリストを保持しており、定期的に登録されているインスタンスのUpdate関数を呼び出している。

　このとき、GameObjectMangaerは実装の詳細、IGameObjectの正体がなんなのか、ということを一切気にせずにプログラミングされている。

#### 実装

　IGameObject.h(インタフェースクラスなので、cppファイルは不要)

|  |
| --- |
| #pragma once  class IGameObject  {  public:  virtual void Update() = 0;  }; |

GameObjectManager.h

|  |
| --- |
| #pragma once  #include <vector> //可変長配列を使うためにインクルード  #include "IGameObject.h"  class GameObjectManager  {  public:  /// <summary>  /// ゲームオブジェクトを登録する。  /// </summary>  /// <param name="go">登録するゲームオブジェクト</param>  void AddGameObject(IGameObject\* go)  {  m\_gameObjectList.push\_back(go);  }  /// <summary>  /// 登録されているゲームオブジェクトの更新処理を呼び出す。。  /// </summary>  void Update();  private:  std::vector<IGameObject\*> m\_gameObjectList;  }; |

GameObjectManager.cpp

|  |
| --- |
| #include "GameObjectManager.h"  void GameObjectManager::Update()  {  for (int i = 0; i < m\_gameObjectList.size(); i++) {  m\_gameObjectList[i]->Update();  }  } |

Player.h

|  |
| --- |
| #pragma once  #include "IGameObject.h"  class Player : public IGameObject  {  public:  void Update();  }; |

Player.cpp

|  |
| --- |
| #include <iostream>  #include "Player.h"  void Player::Update()  {  std::cout << "Player::Update()\n";  } |

main.cpp

|  |
| --- |
| #include "stdafx.h"  #include "GameObjectManager.h"  #include "Player.h"  int main()  {  //GameObjectManagerを使ってみる。  //4人対戦でプレイヤーが4人いるとする。  Player pl[4];  GameObjectManager goMgr;  //プレイヤーを登録していく。  goMgr.AddGameObject(&pl[0]);  goMgr.AddGameObject(&pl[1]);  goMgr.AddGameObject(&pl[2]);  goMgr.AddGameObject(&pl[3]);  //ゲームループ。  while (true) {  goMgr.Update();  }  return 0;  } |

### 14.3x.1.4 Stateパターン

#### 概要

オブジェクトの状態をクラスとして表現するパターン。ゲームでよく利用されるFSM(有限状態機械)で利用できる。

#### クラス図

　PlayerクラスのFSMにStateパターンを使用した場合

IState

+Move

DeathState

+Move

AttackState

+Move

MoveState

+Move

PlayerクラスはIStateの派生クラスのインタスタンスを保持しており、状態によって処理が変わる関数を呼び出している。例えば、状態に応じて移動処理が異なる場合、Playerクラスは処理の詳細を気にせずに、IState::Move関数を呼び出すようになる。

Player

#### if～else、switch～caseとの違い

Playerクラスに状態を表す変数を持たせる実装と、Stateパターンパターンを利用する場合の違いを見ていきましょう。

・状態に関する処理が、各状態を表すクラスに委譲され、Playerクラスの凝集度が向上する。

・PlayerクラスはIStateクラスのインターフェースに対するプログラミンを行うようになるため、実際の状態の正体に依存しない処理が記述できるようになる。そのため、結合度が低下する。

凝集度が上がり、結合度が下がっていくと、一般的に、クラス間の依存関係が下がっていきます。クラス間の依存関係が下がることで、状態に関する処理の修正で、Playerクラスに影響がでる確率が下がっていきます。

|  |
| --- |
|  |

#### 実装(new/deleteを利用する場合)

　では、new/deleteを使って、状態を切り替えていくコードを見ていきましょう。

Lesson\_14\_StatePattern\_01/Player.hを見てください。

|  |
| --- |
| class Player  {  public:  Player();  ~Player();  void Update();  private:  template<class T> void ChangeState();  private:  IPlayerState\* currentState; //現在のステート。  }; |

Playerクラスは現在のステートを表す、currentStateというメンバ変数を保持しています。

続いて、Player.cppを見ていきましょう。

Player.cpp(8行目)コンストラクタで初期ステートを作成

|  |
| --- |
| Player::Player()  {  //待機ステートに切り替える。  ChangeState<PlayerStateIdle>();  } |

Player.cpp(27行目) Stateクラスに処理を委譲

|  |
| --- |
| void Player::Update()  {  //現在の状態によって処理が変わる更新処理を実行。  currentState->Update();  if (GetAsyncKeyState('J')) {  //現在の状態にジャンプ可能か問い合わせを行う。  if (currentState->IsPossibleJump()) {  std::cout << "ジャンプしました！\n";  }  }  //現在の状態が無敵か問い合わせを行う。  if (currentState->IsInvincible()) {  std::cout << "無敵です！。\n";  } |

　状態に依存する処理は、すべてIStateの派生クラスに委譲しています。

Player.cpp(41行目) 状態切り替え

|  |
| --- |
| //ここから状態切り替えのコード。  //これらも状態に依存するなら、各状態クラスに委譲しても良い。  if (GetAsyncKeyState('A')) {  ChangeState<PlayerStateIdle>();  }  if (GetAsyncKeyState('S')) {  //移動状態に切り替え。  ChangeState<PlayerStateMove>();  }  if (GetAsyncKeyState('D')) {  //死亡状態に切り替え。  ChangeState<PlayerStateDeath>();  } |

　キーボードの入力に応じて、状態を切り替えています。この処理も状態に依存する場合は、各状態を表すクラスに委譲しても構いません。例えば、走り攻撃状態は、走り状態からしか遷移しないといった場合は、走り状態から遷移するべきでしょう。

Player.cpp(17行目) 状態を切り替える関数

|  |
| --- |
| template<class T> void Player::ChangeState()  {  //待機状態に切り替え。  if (currentState != nullptr) {  //現在のステートを破棄する。  delete currentState;  }  //次の状態のインスタンスを作成する。  currentState = new T;  } |

ChangeState関数はテンプレート関数と呼ばれる特殊な関数です。詳細はテンプレートのチャプターで説明しますが、Tがテンプレート引数で渡された型に置き換えらえます。

　この関数は、テンプレート引数に指定された型のインスタンスを作成しています。

#### 実習(new/deleteを離床する場合)

Lesson\_14\_StatePattern\_01を改造して、StateFreezeを実装しなさい。仕様は下記のようになります。

・キーボードのFが入力されると、StateFreeze状態に遷移する。

・StateFreeze状態ではジャンプはできない。

・StateFreeze状態は無敵ではない。

#### 実装(Enter/Leave)

さて、先ほどnew/deleteを使って状態を切り替える方法を見ていきました。しかし、new/deleteという処理はメモリの動的確保という処理が行われるため、一般的にとても重い処理となります。そのため、頻繁にnew/deleteが発生するようなコードはゲームプログラムでは御法度です。ゲームにおける状態切り替えは頻繁に起きる場合が多いです。そこで、今回はnew/deleteを利用しないStateパターンの実装方法を見ていきましょう。

　状態は切り替えるときに、状態が始まるときに行いたい処理と、終了するときに行いたい処理がある場合はあります。new/deleteを利用する場合は、コンストラクタとデストラクタを利用していたのですが、今回はそれを利用できません。そこで、IStateクラスにEnterとLeaveという純粋仮想関数を用意してみます。

Lesson\_14\_StatePattern\_02/IPlayerState.h

|  |
| --- |
| class IPlayerState {  public:  **virtual void Enter() = 0;**  **virtual void Leave() = 0;**  virtual void Update() = 0;  //ジャンプできる？  virtual bool IsPossibleJump() = 0;  //無敵？  virtual bool IsInvincible() = 0;  }; |

Enterは状態が始まるときに行う処理、Leaveは状態が終了するときに行う処理を記述します。

ではPlayerクラスを見ていきましょう

Lesson\_14\_StatePattern\_02/Player.h

|  |
| --- |
| class Player  {  public:  Player();  void Update();  private:  void ChangeState(IPlayerState\* nextState);  private:  **PlayerStateIdle idleState; //待機ステート。**  **PlayerStateMove moveState; //移動ステート。**  **PlayerStateDeath deathState; //死亡ステート。**  IPlayerState\* currentState; //現在のステート。  }; |

　Playerクラスは各状態を表すクラスのインスタンスをメンバ変数として保持するようになっています。

|  |
| --- |
| Player::Player()  {  //初期ステートは待機。  ChangeState(&idleState);  }  void Player::ChangeState(IPlayerState\* nextState)  {  状態を切り替えるときにnew/deleteを使用しないようになっている。代わりにLeave、Enter関数を呼び出すことで切り替えた時の処理を実行できるようになっている。  //待機状態に切り替え。  if (currentState != nullptr) {  currentState->Leave();  }  currentState = nextState;  currentState->Enter();  }  void Player::Update()  {  currentState->Update();  if (GetAsyncKeyState('J')) {  if (currentState->IsPossibleJump()) {  std::cout << "ジャンプしました！\n";  }  }  if (currentState->IsInvincible()) {  std::cout << "無敵です！。\n";  }  if (GetAsyncKeyState('A')) {  ChangeState(&idleState);  }  ChangeState関数は、切り替えたい状態クラスのインスタンスのアドレスを引数で渡すようになっている。  if (GetAsyncKeyState('S')) {  //移動状態に切り替え。  ChangeState(&moveState);  }  if (GetAsyncKeyState('D')) {  //死亡状態に切り替え。  ChangeState(&deathState);  }  } |

#### 実習(Enter/Leave)

Lesson\_14\_StatePattern\_01を改造して、StateFreezeを実装しなさい。仕様は下記のようになります。

・キーボードのFが入力されると、StateFreeze状態に遷移する。

・StateFreeze状態ではジャンプはできない。

・StateFreeze状態は無敵ではない。

## Ex.1.2 Hands-On ゲームのデータをセーブしてみよう(テキスト版)

では簡単なゲームデータをテキスト形式で保存してみましょう。下記のプログラムはRPGのプレイヤーの情報をセーブするプログラムです。Visual Studioで新しいプロジェクトを作成して、入力してみてください。

|  |
| --- |
| #define \_CRT\_SECURE\_NO\_WARNINGS  #include <iostream>  //プレイヤー構造体。  struct SPlayer {  int level; //レベル。  int hp; //体力。  int mp; //マジックポイント。  int attack; //攻撃力。  int def; //守備力。  };  int main()  {  //プレイヤーの情報を作成。  SPlayer player;  player.level = 30;  player.hp = 200;  player.mp = 100;  player.attack = 250;  player.def = 100;  //ファイルを書き込みモードで開く。  FILE\* fp = fopen("save.txt", "w");  if (fp != nullptr) { //ファイルが開けたかチェックする。  //sprintf関数を利用して、書き込む文字列を作成する。  //sprintf関数はC言語の標準関数。  char text[1024];  sprintf(  text,  "%d,%d,%d,%d,%d",  player.level,  player.hp,  player.mp,  player.attack,  player.def  );  //作成した文字列をファイルに書き込む。  fputs(text, fp);  //ファイルを閉じる。  fclose(fp);  }  return 0;  } |

このプログラムを実行すると、VisualStudioのプロジェクトがある場所にsave.txtというファイルが作成されています。ではsave.txtを開いて中身を確認してみましょう。

CSV形式(カンマ区切り)

|  |
| --- |
| 30,200,100,250,100 |

ＨＰ

レベル

ＭＰ

攻撃力

守備力

## Ex.1.3 Hands-On ゲームのデータをロードしてみよう(テキスト版)

　では、先ほどセーブしたプレイヤーの情報をロードするプログラムを作成してみましょう。下記のプログラムを入力してみてください。

|  |
| --- |
| #define \_CRT\_SECURE\_NO\_WARNINGS  #include <iostream>  //プレイヤー構造体。  struct SPlayer {  int level; //レベル。  int hp; //体力。  int mp; //マジックポイント。  int attack; //攻撃力。  int def; //守備力。  };  int main()  {  //ファイルを読み込みモードで開く。  FILE\* fp = fopen("save.txt", "r");  if (fp != nullptr) { //ファイルが開けたかチェックする。  //まずファイルから文字列を読み込む。  char text[1024];  fgets(text, 1023, fp);  //ファイルを閉じる。  fclose(fp);  //プレイヤー  SPlayer player;  //sscanfは指定した文字列から書式文字列に従ってデータを取得します。  sscanf(  text,  "%d,%d,%d,%d,%d",  &player.level,  &player.hp,  &player.mp,  &player.attack,  &player.def  );  //ロードした内容を表示してみる。  printf("プレイヤーのレベル　：%d\n", player.level);  printf("プレイヤーのHP　 ：%d\n", player.hp);  printf("プレイヤーのMP　 ：%d\n", player.mp);  printf("プレイヤーの攻撃力　：%d\n", player.attack);  printf("プレイヤーの守備力　：%d\n", player.def);  }  return 0;  } |

## Ex.1.4 ファイルの書き込み(バイナリモード)

では、続いてバイナリモードでのファイルの書き込みを学んでいきましょう。

バイナリモードでファイルに書き込まれるデータは0と1の羅列です。そして、このようなデータはバイナリデータと呼ばれます。単にバイナリとよばれることもあります。コンピュータで表現できるデータは0と1のみですので、正確にはテキストデータもバイナリデータになります。

### Ex.1.4.1　ファイルを開く

　バイナリデータの保存を行うときもファイルを開く必要があります。ただし、バイナリモードで開く必要があります。

wbを指定するとバイナリ書き込みモードになる。

|  |
| --- |
| FILE\* fp = fopen("save.bin", "wb"); |

### Ex.1.4.2　データの書き込み

　データの書き込みにはfwrite関数を利用します。

|  |
| --- |
| //hpのデータを書き込んでみる。  int hp = 10;  fwrite(  &hp, //第一引数は書き込むデータが格納されているアドレス。  sizeof(hp), //第二引数は書き込むデータのサイズ。  1, //第三引数は書き込むデータの要素数。  fp //第四引数は書き込み先となるファイルのファイルポインタ。  ); |

### Ex1.4.3 ファイルを閉じる

　データの書き込みが終了したら、ファイルをきちんと閉じましょう。これはバイナリモードでも同じです。

|  |
| --- |
| fclose(fp); |

では、Sample6.cppを入力してバイナリデータを保存するプログラムを作成してみましょう。

### Sample6.cpp

|  |
| --- |
| #define \_CRT\_SECURE\_NO\_WARNINGS  #include <iostream>  int main()  {  int hp = 10;  //ファイルをバイナリデータ書き込みモードで開く。  FILE\* fp = fopen("save.bin", "wb");  //fwrite関数を利用して、データをファイルに書き込む  fwrite(  &hp, //第一引数は書き込むデータが格納されているアドレス。  sizeof(hp), //第二引数は書き込むデータのサイズ。  1, //第三引数は書き込むデータの要素数。  fp //第四引数は書き込み先となるファイルのファイルポインタ。  );  //書き込みが終わったのでファイルを閉じる。  fclose(fp);  return 0;  } |

これで、VisualStudioのプロジェクトがある場所にsave.binというファイルが作成されています。

## Ex.1.5 ファイルの読み込み(バイナリモード)

では、続いてバイナリデータを読み込む方法について見ていきましょう。

### Ex.1.5.1 ファイルを開く

　バイナリデータを読み込むときはバイナリデータ読み込みモードでファイルを開く必要があります。

rbを指定するとバイナリ読み込みモードになる。

|  |
| --- |
| FILE\* fp = fopen("save.bin", "rb"); |

### Ex.1.5.2 データの読み込み

　データの読み込みにはfread関数を利用します。

|  |
| --- |
| int hp;  //fwrite関数を利用して、ファイルを書き込む  fread(  &hp, //第一引数はデータの読み込み先のアドレス。  sizeof(hp), //第二引数は読み込むデータのサイズ。  1, //第三引数は読み込むデータの要素数。  fp //第四引数は読み込み元となるファイルのファイルポインタ。  ); |

### Ex 1.5.3 ファイルを閉じる

　バイナリデータの読み込みの時も、必ず最後にファイルを閉じてください。

|  |
| --- |
| fclose(fp); |

では、Sample7.cppを入力してバイナリデータを読み込むプログラムを作成してみましょう。

### Sample7.cpp

|  |
| --- |
| #define \_CRT\_SECURE\_NO\_WARNINGS  #include <iostream>  int main()  {    //ファイルを読み込みモードで開く。  FILE\* fp = fopen("save.bin", "rb");  int hp;  //fwrite関数を利用して、ファイルを書き込む  fread(  &hp, //第一引数はデータの読み込み先のアドレス。  sizeof(hp), //第二引数は読み込むデータのサイズ。  1, //第三引数は読み込むデータの要素数。  fp //第四引数は読み込み元となるファイルのファイルポインタ。  );  //書き込みが終わったのでファイルを閉じる。  fclose(fp);  //読み込んだHPを表示する。  printf("hp = %d\n", hp);  return 0;  } |

## Ex.1.6 Hands-On ゲームデータをセーブしてみよう(バイナリ版)

では、今度はゲームデータをバイナリ形式で保存してみましょう。下記のプログラムを入力してみてください。

|  |
| --- |
| #define \_CRT\_SECURE\_NO\_WARNINGS  #include <iostream>  //プレイヤー構造体。  struct SPlayer {  int level; //レベル。  int hp; //体力。  int mp; //マジックポイント。  int attack; //攻撃力。  int def; //守備力。  };  int main()  {  //プレイヤーの情報を作成。  SPlayer player;  player.level = 30;  player.hp = 200;  player.mp = 100;  player.attack = 250;  player.def = 100;  //ファイルをバイナリ書き込みモードで開く。  FILE\* fp = fopen("save.bin", "wb");  if (fp != nullptr) { //ファイルが開けたかチェックする。  //fwrite関数を利用して書き込む。  fwrite(  &player,  sizeof( player),  1,  fp  );  //ファイルを閉じる。  fclose(fp);  }  return 0;  } |

　このプログラムを実行すると、VisualStudioのプロジェクトがある場所にsave.binというファイルが作成されています。

## Ex.1.7 Hands-On ゲームデータをロードしてみよう(バイナリ版)

では、先ほどセーブしたプレイヤーの情報をロードしてみましょう。下記のプログラムを入力してください。

|  |
| --- |
| #define \_CRT\_SECURE\_NO\_WARNINGS  #include <iostream>  //プレイヤー構造体。  struct SPlayer {  int level; //レベル。  int hp; //体力。  int mp; //マジックポイント。  int attack; //攻撃力。  int def; //守備力。  };  int main()  {  //ファイルをバイナリ読み込みモードで開く。  FILE\* fp = fopen("save.bin", "rb");  if (fp != nullptr) { //ファイルが開けたかチェックする。  //fwrite関数を利用して読み込む。  SPlayer player;  fread(  &player,  sizeof( player),  1,  fp  );  //ファイルを閉じる。  fclose(fp);  //ロードしたプレイヤーの情報を表示する。  //ロードした内容を表示してみる。  printf("プレイヤーのレベル　：%d\n", player.level);  printf("プレイヤーのHP　 ：%d\n", player.hp);  printf("プレイヤーのMP　 ：%d\n", player.mp);  printf("プレイヤーの攻撃力　：%d\n", player.attack);  printf("プレイヤーの守備力　：%d\n", player.def);  }  return 0;  } |

Lesson Ex.1 中間テスト２

　下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSf6piDd3u3IKIJaB0YXb_iGYaKfmX47yT7jNq8-fmY0KDqKvg/viewform?usp=sf_link>

# Chapter 15「#defineよりもconst、enum、inline関数を利用しよう」

## 15.1 プリプロセッサとは

　さて、この章を読み進める前にプリプロセッサについて勉強しましょう。プリプロセッサはコンパイラの前に実行されるソフトウェアの名称で、ソースコードの整形を行ってくれます。#から始まるキーワードがプリプロセッサへの命令です。では、プリプロセッサへの命令のいくつかを見ていこうと思います。

* #define 置き換え前の文字列　置き換え後の文字列

#defineはプログラム中の置き換え前の文字列を、置き換え後の文字列に変換してくれます。例えば下記のリスト15.1のソースコードの場合、プリプロセッサが変換すると、リスト15.2のようなソースコードに変形されます。

**[リスト15.1]**

|  |
| --- |
| #define PI 3.14  int main(void)  {  //円の半径  float radius;  //円の面積  float area;  //円周の長さ  float length;  //球の半径を入力。  printf("円の半径を入力: r = ");  scanf\_s("%f", &radius);  //円の面積を計算・出力  area = **PI** \* radius \* radius;  printf("円の面積: S = %.3f\n", area);  //円周の長さを計算・出力  length = 2 \* **PI** \* radius;  printf("円周の長さ: L = %.3f\n", length);  return 0;  } |

**[リスト15.2 ]**

|  |
| --- |
| #define PI 3.14  int main(void)  {  //円の半径  float radius;  //円の面積  float area;  //円周の長さ  float length;  //球の半径を入力。  printf("円の半径を入力: r = ");  scanf\_s("%f", &radius);  //円の面積を計算・出力  area = 3.14 \* radius \* radius;  printf("円の面積: S = %.3f\n", area);  //円周の長さを計算・出力  length = 2 \* 3.14 \* radius;  printf("円周の長さ: L = %.3f\n", length);  return 0;  } |

* #include ファイルの展開

#includeは指定されたファイルをプログラム中に展開します。

# 15.2 マジックナンバーより定数を使おう

　基本的にマジックナンバーは悪です。ヒューマンエラーの元ですし、修正も困難です。また数字が何を意味しているのかも分かりません。マジックナンバーの代わりに#defineマクロを利用するのはいい方針です。リスト15.1で見たように、#defineを利用することで、意味のある定数を利用することができます。

## 15.3 関数マクロ

　#defineはあくまで文字列を置き換えているだけです。ですので、#defineを利用することで、関数のようなものを作成することができます。#defineを利用して作成させる関数のようなマクロは関数マクロと呼ばれます。リスト15.3は関数マクロを利用したプログラムです。

[リスト15.3]

|  |
| --- |
| #define MAX(v, a, b ) \  if( a > b ) { \  v = a; \  }else{ \  v = b; \  }  int main()  {  int hoge;  MAX(hoge, 10, 20); //大きいほうの数値がhogeに代入される。  printf("hoge = %d\n", hoge); //hoge = 20と表示される。  } |

## 15.4 関数マクロの利点

関数マクロはまるで関数のように振る舞いますが、マクロ文字列が置きかえされるだけなので、通常の関数呼び出しで発生する処理負荷がありません。そのため、関数呼び出しよりも高速になるケースがあります。

1. 正しくは構造体にも関数を記述することはできます。ここではクラスとの考え方の違いを明確にするためにこのように記述しています。 [↑](#footnote-ref-1)