第 1 章の解答

練習問題　1.2　P.015

1. 関数を呼び出すには関数名のあとに () を付けます。

2. 以下のようなプログラムになります。

▶ リスト App1.1　main() 関数で hello_world() 関数を呼び出す

```
#include <iostream>

int sum(int a, int b, int c)
{
    int ab = a + b;
    int abc = ab + c;
    return abc;
}

int main()
{
    std::cout << "sum(5, 1, -2): " << sum(5, 1, -2) << std::endl;
}
```

◆ 解説

仮引数は、関数定義の際、() の中にその関数が必要とする引数の一覧を書きます。

3. return 文のところで show_messages() 関数の処理が終了し、呼び出し元の main() 関数に戻ってしまうためです。

練習問題　1.3　P.022

1. 一例を示します。この場合、$3 \times -5 + 2 = -15 + 2 = -13$ となり正しいことが確認できます。

▶ リスト App1.2　3 種類以上の加減乗除を組み合わせる

```
#include <iostream>

int main()
{
    int a = 3;
    int b = -5;
    int c = 2;
    int res = a * b + c;
    std::cout << res << std::endl;
}
```

▶ 実行結果

```
-13
```

2. 1. の例をもとに、次のように変更します。この場合、$3 \times (-5 + 2) = 3 \times -3 = -9$ なので正しいことが確認できます。

▶ リスト App1.3　括弧を使って計算順序を変更する

```
#include <iostream>

int main()
{
    int a = 3;
    int b = -5;
    int c = 2;
    int res = a * (b + c);
    std::cout << res << std::endl;
}
```

▶ 実行結果

```
-9
```

3. 以下のようなプログラムになります。

▶ リスト App1.4　複合代入演算子を使わずに書き換える

```
#include <iostream>

int main()
{
    int i = 0;
    i = i * (2 + 4);
    i = i % 3;
    std::cout << i << std::endl;
}
```

◆ 解説

複合代入演算子の優先順位（表 1.4 p.21）はとても低くなっています。そのため、基本的に右辺の計算が先に行われてしまいます。一方、通常の演算子は優先順位が比較的高いため、そのまま展開すると計算の順序が変わってしまいます。同じ計算順序になるようにするには、先に計算すべき式を括弧で囲む必要があります。

練習問題　1.4　P.035

1. 回答例は 2 通りあります。両方を掲載するので参考にしてください。

▶ リスト App1.5　条件文全体を論理否定する

```
#include <iostream>

void show_message(int value)
{
    if (!(10 <= value && value < 20))
    {
        std::cout << "10 以上 20 未満ではありません" << std::endl;
    }
    else
    {
        std::cout << "10 以上 20 未満です" << std::endl;
    }
}

int main()
{
    show_message(9);  // 10 未満なので else 文のメッセージが表示される
    show_message(15); // 10 以上 20 未満なので if 文のメッセージが表示される
    show_message(20); // 20 以上なので else 文のメッセージが表示される
}
```

◆ 解説

もともとあった条件文全体を論理否定すると真偽値が入れ替わります。論理否定は演算子の優先順位としてもかなり上位にあるので、条件文を括弧で囲んでおかないと意図しない結果となります。

▶ リスト App1.6　ド・モルガンの法則を利用する

```
#include <iostream>

void show_message(int value)
{
    if (value < 10 || 20 <= value)
    {
        std::cout << "10 以上 20 未満ではありません" << std::endl;
```

```
    }
    else
    {
        std::cout << "10 以上 20 未満です" << std::endl;
    }
}
int main()
{
    show_message(9);  // 10 未満なので else 文のメッセージが表示される
    show_message(15); // 10 以上 20 未満なので if 文のメッセージが表示される
    show_message(20); // 20 以上なので else 文のメッセージが表示される
}
```

◆ 解説

ド・モルガンの法則を使うと全体を否定せずに真偽値を入れ替えることができます。ド・モルガンの法則については各自で調べてみましょう。

2. （省略）

3. 以下のようなプログラムになります。

▶ リスト App1.7　switch 文を if 文に書き直す

```
#include <iostream>

int main()
{
    int a = 2;

    int b = a + 1; // a + 1 を何度も書くのはミスのものとなので変数を使う
    if (b == 1)
    {
        std::cout << "a + 1 は 1 です" << std::endl;
    }
    else if (b == 2)
    {
        std::cout << "a + 1 は 2 です" << std::endl;
    }
    else if (b == 3)
    {
```

```
        std::cout << "a + 1 は 3 です" << std::endl;
    }
    else
    {
        std::cout << "a + 1 は 1 でも 2 でも 3 でもありません" << std::endl;
    }
}
```

◆ 解説

switch 文の case ラベルと default ラベルは好きな順番にすることができますが、if 文は上から順番に条件を調べていくので並べ替えなければならない場合があります。

練習問題　1.5　P.050

1. 修正例は以下のようなプログラムになります。int 型は整数を扱う型で、浮動小数点数を代入すると暗黙の型変換で小数点以下が切り捨てられてしまうためです。

▶ リスト App1.8　出力を 42.195 になるような修正

```
#include <iostream>

void show_value(float f)
{
    std::cout << f << std::endl;
}

int main()
{
    float i = 42.195f;
    show_value(i);
}
```

2. 一見、0.001 が出力されるように考えられますが、実際には 0.000999451 のような値が出力されます。コンピューターが扱う浮動小数点数はすべての実数を正確に表すことができず、多くの場合近似値が使われているためです。

3. 以下のようなプログラムになります。

▶ リスト App1.9　出力が 42 になるような、ポインターを使った修正

```
#include <iostream>

int main()
{
    int i = 0;

    int* p = &i;
    *p = 42;

    std::cout << i << std::endl;
}
```

練習問題　1.6　P.058

1. 以下のようなプログラムになります。

▶ リスト App1.10　配列 array の要素を順に表示

```
#include <iostream>

int main()
{
    int array[] = {4, 2, 1, 9, 5};

    std::cout << array[0] << std::endl;
    std::cout << array[1] << std::endl;
    std::cout << array[2] << std::endl;
    std::cout << array[3] << std::endl;
    std::cout << array[4] << std::endl;
}
```

◆ 解説

添え字演算子を使うと配列の各要素を取り出すことができます。このとき配列は 0 オリジンで数えるので 0 番目から順にアクセスします。

2. 以下のようなプログラムになります。ヌル文字が現れるところまでを文字列として扱うため、途中で現れた場合にそこで文字列が終わったと見なされてしまいます。

▶ リスト App1.11　配列 string を途中にヌル文字を含んだ文字列で初期化

```
#include <iostream>

int main()
{
    char string[] = "hello, \0 null character";

    std::cout << string << std::endl;
}
```

▶ 実行結果

```
hello,
```

3. 以下のようなプログラムになります。

▶ リスト App1.12　配列を std::string に変更

```
#include <iostream>
#include <string>

int main()
{
    std::string string = "hello, \0 null character";

    std::cout << string << std::endl;
}
```

▶ 実行結果

```
hello,
```

練習問題　1.7　P.068

1. 以下のようなプログラムになります。

▶ リスト App1.13　配列を最後から順番に処理

```
#include <iostream>

int main()
{
    int array[] = {4, 2, 1, 9, 5};

    int i = 5;
    while (i--)
    {
        std::cout << array[i] << std::endl;
    }
}
```

◆ 解説

後置のデクリメント演算子は減算する前の値を返すので、条件文が評価されるときには５ ４ ３ ２ １ ０となり、最後の０が偽となるのでそこでループが終了します。一方、ループの内側ではｉはデクリメントされたあとの値が使われて４ ３ ２ １ ０となるので、配列の添え字として正しくなります。

2. 以下のようなプログラムになります。

▶ リスト App1.14　for 文に書き換え

```
#include <iostream>

int main()
{
    int array[] = {4, 2, 1, 9, 5};

    for (int i = 5; i--; )
    {
        std::cout << array[i] << std::endl;
    }
}
```

3. 少し難しいですが、以下のようなプログラムになります。

▶ リスト App1.15　do-while 文に書き換え

```
#include <iostream>

int main()
{
    int array[] = {4, 2, 1, 9, 5};

    int i = 5;
    do
    {
        std::cout << array[--i] << std::endl;
    } while (i);
}
```

◆ 解説

do-while 文ではループ本体の実行が先に行われます。そのため i が有効な添え字となるように前置のデクリメント演算子で 4 3 2 1 0 となるようにします。条件文の評価がされるときにはすでに i はデクリメントされたあとの値となっているので、i が 0 となったとき、すなわち配列の先頭を表示した直後に条件が偽となりループが終了します。

この章の理解度チェック P.068

1. 以下のようなプログラムになります。

▶ リスト App1.16　動作するよう修正

```
#include <iostream>

int main()
{
    std::cout << "hello, message" << std::endl;
}
```

◆ 解説

std::cout を使うには<iostream>ヘッダーをインクルードする必要があります。

2. 実数をコンピューターが扱えるように近似値で表したものです。浮動小数点数には、float 型、double 型、long double 型があります。

3. 条件の組み合わせが増えた際に、ネストした if 文のインデントが深くなってしまい、プログラムの流れがわかりづらくなるのを防ぐことができます。

4. 以下のようなプログラムになります。

▶ リスト App1.17　文字列を変数に格納し、その文字列を 1 行に 1 文字ずつ表示

```
#include <iostream>
#include <string>

int main()
{
    std::string s = "Hello, string";

    for (auto c : s)
    {
        std::cout << c << std::endl;
    }
}
```

◆ 解説

std::string 型は基本的に配列と同じように使えるので、範囲 for 文を使って走査することができます。

第 2 章の解答

練習問題　2.1　P.080

1. 以下のようなプログラムになります。

▶ リスト App2.1　アロー演算子を使うように変更

```
struct product
{
    int id;
    int price;
    int stock;
};

// 引数として構造体変数を受け取る関数
void show_product(product product)
{
    std::cout << "商品 ID：" << (&product)->id << std::endl;
    std::cout << "単価：" << (&product)->price << std::endl;
    std::cout << "在庫数：" << (&product)->stock << std::endl;
}

int main()
{
    product pen =
    {
        0, // 商品 ID
        100, // 単価
        200, // 在庫数
    };
    show_product(pen); // 通常の変数と同じように関数に渡す
}
```

◆ 解説

演算子の優先順位ではアドレス演算子よりアロー演算子のほうが高いので、括弧で囲む必要があります。

2. 共用体はすべてのメンバー変数が同じメモリ領域を共有しているため、どれか 1 つでも変更するとすべてのメンバー変数が影響を受けます。一方、構造体はすべてのメンバー変数がそれぞれ異なったメモリ領域に配置されるので、メンバー変数の変更が他のメンバー変数に影響することはありません。

3. 実際に実行すると、10 が出力されます。共用体はメンバー変数のメモリ領域を共有する機能ですが、メンバー変数が構造体の場合、構造体変数と他のメンバー変数が同じメモリ領域を共有するだけで、構造体変数の中身までは影響を与えません。そのため、共用体のメンバーとなっている構造体変数のメンバー変数は、それぞれ専用のメモリ領域を与えられます。

練習問題　2.2　P.086

1. 長くなるので変更部分のみを抜粋します。

▶ リスト App2.2　メンバー関数を追加

```
class product
{
      int id;    // 商品 ID
      int price; // 単価
      int stock; // 在庫数

public:
    int get_id();            // 商品 ID の getter
    void set_id(int new_id); // 商品 ID の setter

    int get_price();               // 単価の getter
    void set_price(int new_price); // 単価の setter

    int get_stock();               // 在庫数の getter
    void set_stock(int new_stock); // 在庫数の setter

    void set(int new_id, int new_price, int new_stock); // 追加
};
```

```
// 追加
void product::set(int new_id, int new_price, int new_stock)
{
    id    = new_id;
    price = new_price;
    stock = new_stock;
}
```

2. 長くなるので変更部分のみを抜粋します。

▶ リスト App2.3　main() 関数でメンバー関数を使う

```
int main()
{
    product pen; // ペンに関するデータを持つ変数

    pen.set(0, 100, 200); // 修正箇所

    product* ptr = &pen; // インスタンスへのポインター

    // アロー演算子を使って get  ter から値を取得
    std::cout << "商品 ID：" << ptr->get_id() << std::endl;
    std::cout << "単価：" << ptr->get_price() << std::endl;
    std::cout << "在庫数：" << ptr->get_stock() << std::endl;
}
```

練習問題　2.3　P.091

1. 以下のようなプログラムになります。

▶ リスト App2.4　出力が 42 になるように、参照を使って修正

```
#include <iostream>

int main()
{
    int i = 0;
```

```
    int& j = i;
    j = 42;

    std::cout << i << std::endl;
}
```

◆ 解説

参照は変数に別名を与える機能なので、ポインターのように演算子を駆使することなく、普通の変数のように扱うことができます。

2. 以下のようなプログラムになります。

▶ リスト App2.5　エラーにならないよう r を修正

```
int main()
{
    const int i = 42;

    const int& r = i;
}
```

◆ 解説

通常の参照では const 修飾された変数への参照を格納できないので、const 参照を使う必要があります。

練習問題　2.4　P.096

1. 具体的な型名の代わりに auto を使います。

2. 配列の要素の型と同じ、int 型に推論されます。

3. 任意の式から型推論できます。

練習問題　2.5　P.101

1. 以下のようなプログラムになります。

▶ リスト App2.6　長い名前のクラスの、短い別名

```
class very_long_long_named_class
{
};

using vllnc = very_long_long_named_class;
```

◆ 解説

using 宣言を使うと型に別名を与えることができます。また、C 言語との互換性が必要な場面では typedef を使うことができます。

2. 以下のようなプログラムになります。

▶ リスト App2.7　ネストした型名を持つクラスとその型を持つメンバー変数、getter/setter

```
class A
{
    using integer = int;

    integer m_value;

public:
    void setter(integer value);
    integer getter() const;
};

void A::setter(integer value)
{
    m_value = value;
}

A::integer A::getter() const
{
```

```
    return m_value;
}
```

◆ 解説

ネストした型名を使う際には、先に宣言を行う必要があります。しかしアクセス指定子の影響を受けるため、クラスの外で使う必要がある場合には public な領域で宣言しなければなりません。この例ではクラス以外がネストした型名を使っていないので private で宣言していても問題ありません。

練習問題　2.6　P.104

1. 以下のようなプログラムになります。

▶ リスト App2.8　std::cin による入力を出力

```
#include <iostream>

int main()
{
    double d = 0;

    std::cin >> d; // 浮動小数点数の入力

    std::cout << "入力された小数は " << d << " です" << std::endl;
}
```

▶ 実行結果

```
1.23456
入力された小数は 1.23456 です
```

2. 以下のようなプログラムになります。

▶ リスト App2.9　1 行文字列の入力を出力

```
#include <iostream>
#include <string>

int main()
{
    std::string line;

    std::getline(std::cin, line);

    std::cout << "入力された行は \"" << line << "\" です" << std::endl;
}
```

▶ 実行結果

```
one liner
入力された行は "one liner" です
```

◆ 解説

コンソールから 1 行入力するには std::getline() 関数を使います。直接 std::string の変数に入力しようとして std::cin >> line; とすると、1 単語しか入力されません。

3. 以下のようなプログラムになります。プロンプトとして"> "を表示しています。

▶ リスト App2.10　空行が入力されるまで入力を出力

```
#include <iostream>
#include <string>

int main()
{
    std::string line;
```

```
    do
    {
        std::cout << "> ";

        std::getline(std::cin, line);

        std::cout << "入力された行は \"" << line << "\" です" << std::endl;
    } while (line == "");

    std::cout << "終了" << std::endl;
}
```

▶ 実行結果

```
> one liner
入力された行は "one liner" です
> second line
入力された行は "second line" です
>
入力された行は "" です
終了
```

◆ 解説

do-while 文は必ず 1 回はループの本体を実行するループ文でした。プロンプトを表示して入力を待つようなプログラムの場合、do-while 文を使うとうまく処理できることがあります。

練習問題　2.7　P.110

1. 長くなるので変更部分のみを抜粋します。

▶ リスト App2.11　4 引数の sum() 関数を追加

```
int sum(int a, int b, int c, int d)
{
    return sum(sum(a, b, c), d);
```

```
}

int main()
{
    int x = sum(10, 20); // 2 引数版のオーバーロードを呼び出す
    show_value(x);

    int y = sum(5, 15, 25); // 3 引数版のオーバーロードを呼び出す
    show_value(y);

    int z = sum(-10, 20, -15, 42); // 4 引数版のオーバーロードを呼び出す
    show_value(z);
}
```

▶ 実行結果

```
30
45
37
```

◆ 解説

オーバーロードは同じ関数名で異なった定義を追加する機能です。オーバーロードするには、引数の数もしくは引数の型が異なる必要があります。また、オーバーロードの中では同じ関数名の他のオーバーロードを呼び出すこともできます。

2. 長くなるので追加点のみを抜粋します。

▶ リスト App2.12　int 型と vector2d 型を返す sub() 関数

```
int sub(int a, int b)
{
    return a - b;
}

vector2d sub(vector2d a, vector2d b)
{
    vector2d r =
```

```
    {
        a.x - b.x,
        a.y - b.y,
    };
    return r;
}
```

3. 関数の引数リストの、後ろから順番に指定できます。

◆ 解説

デフォルト引数を指定できるのは引数リストの後ろからです。先頭や途中の引数にデフォルト引数を指定することはできません。

練習問題　2.8　P.121

1. 以下のようなプログラムになります。

▶ リスト App2.13　ラムダ式を使って show_value() 関数を書き直す

```
#include <iostream>

int main()
{
    auto show_value = [](int v) -> void
    {
        std::cout << v << std::endl;
    };
    show_value(42);
}
```

◆ 解説

ラムダ式はコンパイラーが自動でユニークな型名を付けるため auto を使って型推論する必要があります。

2. ラムダ式の戻り値の型を推論できないのは、ラムダ式の return 文で返す値の型のすべてが一致していない場合です。そのため、キャストなどを使い、すべての return 文で同じ型になるようにします。

3. 参照キャプチャを使うことで、ラムダ式はコピーではなく参照を持つことになります。そのため、ラムダ式の内部で行った変更が元の変数にも影響します。

この章の理解度チェック P.121

1. 以下のようなプログラムになります。

▶ リスト App2.14　vector3d クラス

```
class vector3d
{
public:
    using element_type = int;

private:
    element_type x;
    element_type y;
    element_type z;
};
```

2. 以下のようなプログラムになります。

▶ リスト App2.15　getter / setter の追加

```
class vector3d
{
public:
    using element_type = int;

private:
    element_type x;
    element_type y;
    element_type z;
```

```
public:
    element_type getX();
    void setX(element_type X);

    element_type getY();
    void setY(element_type Y);

    element_type getZ();
    void setZ(element_type Z);
};

vector3d::element_type vector3d::getX()
{
    return x;
}

void vector3d::setX(element_type X)
{
    x = X;
}

vector3d::element_type vector3d::getY()
{
    return y;
}

void vector3d::setY(element_type Y)
{
    y = Y;
}

vector3d::element_type vector3d::getZ()
{
    return z;
}

void vector3d::setZ(element_type z)
{
    z = Z;
}
```

3. 長くなるので追加点のみを抜粋します。

▶ リスト App2.16　入力値をセットする

```
#include <iostream>

// vector3d

int main()
{
    int x = 0, y = 0, z = 0;

    std::cin >> x >> y >> z;

    vector3d vec;

    vec.setX(x);
    vec.setY(y);
    vec.setZ(z);
}
```

4. 長くなるので追加点のみを抜粋します。

▶ リスト App2.17　オーバーロードの場合

```
vector3d set(vector3d::element_type x,
    vector3d::element_type y, vector3d::element_type z)
{
    vector3d vec;
    vec.setX(x);
    vec.setY(y);
    vec.setZ(z);
}

vector3d set(vector3d::element_type x, vector3d::element_type y)
{
    return set(x, y, 0);
}
```

▶ リスト App2.18　デフォルト引数の場合

```
vector3d set(vector3d::element_type x,
    vector3d::element_type y, vector3d::element_type z = 0)
{
    vector3d vec;
    vec.setX(x);
    vec.setY(y);
    vec.setZ(z);
}
```

5. デフォルトでコピーのキャプチャが行われ、関数の内側の変数のコピーがラムダ式の中に作られます。その際、ラムダ式本体で使われていない変数についてはコピーされることはありません。

第 3 章の解答

練習問題　3.1　P.131

1. 長くなるので変更点のみを抜粋します。

▶ リスト App3.1　コンパイルできるように修正

```
class A
{
    int v;

public:
    void set(int value);
    int get() const;
};

int A::get() const
{
    return v;
}
```

◆ 解説

インスタンス自体が const、もしくは const 参照を経由してメンバー関数にアクセスするには const メンバー関数となっている必要があります。

2. 以下のようなプログラムになります。

▶ リスト App3.2　メンバー変数を参照と const 参照で返すオーバーロード

```
class A
{
    int v;

public:
    int& get();
    const int& get() const;
};
```

```
int& A::get()
{
    return v;
}

const int& A::get() const
{
    return v;
}
```

◆ 解説

const メンバー関数と非 const メンバー関数はオーバーロードすることができます。

3. const メンバー関数からもメンバー変数を変更できるようになります。

練習問題　3.2　P.139

1. 1 つ目は、戻り値の型の指定がないことです。コンストラクターはインスタンスを返す関数と見なせるので、戻り値の型は決まっていると考えられるためです。2 つ目は、メンバー初期化リストがあることです。メンバー変数の初期化はメンバー初期化リストで行えます。

2. 以下のようなプログラムになります。

▶ リスト App3.3　コンストラクターとデストラクターを持ったクラス

```
#include <iostream>

class A
{
public:
    A();
    ~A();
};

A::A()
{
    std::cout << "コンストラクター呼び出し" << std::endl;
```

```
}

A::~A()
{
    std::cout << "デストラクター呼び出し" << std::endl;
}

int main()
{
    A a;

    std::cout << "main() 関数" << std::endl;
}
```

▶ 実行結果

```
コンストラクター呼び出し
main() 関数
デストラクター呼び出し
```

3. Resource Acquisition Is Initialization、略して RAII と言います。

練習問題　3.3　P.150

1. 以下のようなプログラムになります。

▶ リスト App3.4　引数でメンバー変数を初期化するコンストラクター

```
#include <string>

class Book
{
    std::string title;
    std::string writer;

    int         price;
```

```
public:
    Book(std::string title, std::string writer, int price);
};

Book::Book(std::string title, std::string writer, int price)
  : title{title}, writer{writer}, price{price}
{
}
```

2. 長くなるので追加部分のみを抜粋します。

▶ リスト App3.5　コピーコンストラクタを追加

```
class Book
{
    std::string title;
    std::string writer;
    int         price;

public:
    Book(std::string title, std::string writer, int price);

    Book(const Book& other);

    void show() const;
};

Book::Book(const Book& other)
  : title{other.title}, writer{other.writer}, price{other.price}
{
}

void Book::show() const
{
    std::cout << title << " " << writer << " " << price << std::endl;
}

int main()
{
    Book source{"dictionary", "someone", 1000};

    Book copy{source};
```

```
    copy.show();
}
```

▶ 実行結果

```
dictionary someone 1000
```

◆ 解説

コピーコンストラクターでは、メンバー初期化リストを使ってメンバー変数ごとにコピーする必要があります。

3. 呼び出そうとしているコンストラクターが explicit 指定されているときです。

練習問題　3.4　P.153

1. 以下のようなプログラムになります。

▶ リスト App3.6　NSDMI を使うような書き直し

```
class vector3d
{
    float x{0};
    float y{0};
    float z{0};
};
```

◆ 解説

NSDMI を使うとメンバー変数にデフォルト値を指定することができ、コンストラクターで初期化を書く必要がなくなります。この例の場合、デフォルトコンストラクターはメンバー変数の初期化しか行っていなかったので、デフォルトコンストラクター自体を定義せずに、コンパイラーが自動生成するデフォルトコンストラクターを使うようにしています。

練習問題　3.5　P.165

1. 継承するクラスを派生クラス、継承されるクラスを基底クラス（基本クラス）と呼びます。

2. 長くなるので追加点のみを抜粋します。

▶ リスト App3.7　MoreDerived クラス

```
class MoreDerived : public Derived
{
public:
    std::string name() const override;
};

std::string MoreDerived::name() const { return "MoreDerived"; }
```

3. 長くなるので変更点のみを抜粋します。

▶ リスト App3.8　純粋仮想関数の追加

```
class Base
{
public:
    virtual std::string name() const;

    virtual std::string most_name() const = 0;
};

class MoreDerived : public Derived
{
public:
    std::string name() const override;

    std::string most_name() const override;
};

std::string MoreDerived::most_name() const { return "MoreDerived"; }

int main()
{
```

```
    Derived d; // エラー。抽象クラスのインスタンスを作ることができない
}
```

◆ 解説

純粋仮想関数は宣言のみを持ち、定義を持たない仮想関数です。純粋仮想関数を持ったクラスは抽象クラスと呼ばれ、インスタンスを作ることができません。MoreDerived では純粋仮想関数をオーバーライドしているのでインスタンスを作ることができますが、Derived ではオーバーライドしていません。そのため Derived はまだ抽象クラスです。

練習問題　3.6　P.169

1. メンバー変数名と仮引数名や関数内の変数名が同じであるときです。

◆ 解説

仮引数名や関数内の変数名とメンバー変数名が同じであった場合、仮引数もしくは関数内の変数のほうが優先して使われます。メンバー変数を参照したい場合にはコンパイラーが自動的に生成する this ポインターを使い、間接参照します。

2. 以下のようなプログラムになります。

▶ リスト App3.9　this ポインターが必要になる例

```
class A
{
    int value = 0;

public:
    void set(int value);
    int get() const;
};

void A::set(int value)
{
    value = value; // どちらも仮引数
    // this->value = value; // this ポインターを使うほうはメンバー変数
}
```

```
int main()
{
    A a;

    a.set(42);

    std::cout << a.get() << std::endl;
}
```

this ポインターを使わなかった場合の実行結果は以下のとおりです。

▶ 実行結果

```
0
```

this ポインターを使った場合の実行結果は以下のとおりです。

▶ 実行結果

```
42
```

3. const メンバー関数の中で使われるときです。インスタンス自体が const なので、this ポインターを使っても変更できないよう、const ポインターとなっています。

練習問題　3.7　P.176

1. クラスと構造体に本質的な差はなく、デフォルトのアクセス指定子だけが異なります。

2. 2. の派生（継承）ができません。

```
union B
{
    int a;
};
```

```
union D : B // エラー。共用体は派生できない
{
    int b;
};
```

◆ 解説

共用体はコンストラクターやデストラクターを始めとしてメンバー関数を持つことができます。また、必要に応じてアクセス指定子を使って外部に公開するメンバーを選択できます。

3. static キーワードを付けます。

練習問題　3.8　P.179

1. フレンド関数は、プライベートメンバーにアクセスすることができる非メンバー関数です。

2. 長くなるので追加点のみを抜粋します。

▶ リスト App3.10　フレンド関数の追加

```
class vector3d
{
    float x;
    float y;
    float z;

public:
    friend vector3d sub(const vector3d& lhs, const vector3d& rhs);
};

vector3d sub(const vector3d& lhs, const vector3d& rhs)
{
    vector3d result =
    {
        lhs.x - rhs.x,
        lhs.y - rhs.y,
        lhs.z - rhs.z,
```

```
    };
    return result;
}
```

練習問題　3.9　P.184

1. static メンバーは特定のインスタンスにひも付かないメンバーであり、static メンバー変数と static メンバー関数とがあります。これらはどちらも、インスタンスが作られていない状態でも使うことができます。

2. 以下のようなプログラムになります。

▶ リスト App3.11　static メンバーの追加

```
class A
{
    static int counter; // static メンバー変数の宣言

public:
    A();

    static int get_counter();
};

int A::counter = 0; // static メンバー変数の定義と初期化

A::A()
{
    ++counter;
}

int A::get_counter()
{
    return counter;
}
```

この章の理解度チェック P.185

1. 引数を受け取らないデフォルトコンストラクターはコンパイラーが暗黙的に定義しますが、何か 1 つでもコンストラクターを定義することでコンパイラーによる自動生成がされなくなってしまうため、引数がない変数宣言がエラーとなるからです。

2. 以下のようなプログラムになります。

▶ リスト App3.12　NSDMI を使った形式への書き換え

```
class vector3d
{
    float x = 0;
    float y = 0;
    float z = 0;

public:
    vector3d();
    vector3d(float x, float y);
    vector3d(float x, float y, float z);
};

vector3d::vector3d()
{
}

vector3d::vector3d(float x, float y)
  : x(x), y(y)
{
}

vector3d::vector3d(float x, float y, float z)
  : x(x), y(y), z(z)
{
}
```

◆ 解説

NSDMI を使うとメンバー変数にデフォルト値を設定できるので、メンバー初期化リストの一部を省略することができます。しかしこの例ではコンストラクターが複数宣言されているので、デフォルトコンストラクターの宣言自体を省略できません。

3. 以下のようなプログラムになります。

▶ リスト App3.13　名前の隠蔽を回避しつつオーバーロードを追加

```
class Base
{
public:
    void show();
};

void Base::show()
{
    std::cout << "Base::show()" << std::endl;
}

class Derived : public Base
{
public:
    using Base::show;

    void show(int value);
};

void Derived::show(int value)
{
    std::cout << "Derived::show(" << value << ")" << std::endl;
}
```

◆ 解説

using 宣言を使うと、一度隠蔽されてしまった基底クラスのメンバーも派生クラスのメンバーと同様に扱えるようになります。

4. 以下のようなプログラムになります。

▶ リスト App3.14　フレンド関数によるクラスのプライベートメンバーへのアクセス

```
#include <iostream>

class A
{
    int private_value = 42;

    friend void disclose(const A& a);
};

void disclose(const A& a)
{
    std::cout << a.private_value << std::endl;
}

int main()
{
    A xx;

    disclose(xx);
}
```

5. 以下のようなプログラムになります。

▶ リスト App3.15　const メンバー関数からの static メンバー変数の変更

```
#include <iostream>

class A
{
    static int value;

public:
    void set_value(int value) const;
    int get_value() const;
};

int A::value = 0;
```

```
void A::set_value(int value) const
{
    // this ポインターは使えないのでスコープ解決演算子を使う
    A::value = value;
}

int A::get_value() const
{
    return value;
}

int main()
{
    A a;

    a.set_value(10);

    std::cout << a.get_value() << std::endl;
}
```

◆ 解説

const メンバー関数は、インスタンスが const となっていることを前提とする、メンバー変数を変更できないメンバー関数です。しかし static メンバー変数はインスタンスとはひも付かないので、インスタンスが const かどうかに関係なく変更できます。

第 4 章の解答

練習問題　4.2　P.195

1. 以下のようなプログラムになります。

▶ リスト App4.1　前方宣言の追加

```
#include <iostream>

int sum(int lhs, int rhs); // 追加した前方宣言

int main()
{
    sum(10, 5);
}

int sum(int lhs, int rhs)
{
    return lhs + rhs;
}
```

2. 以下のようなプログラムになります。

▶ リスト App4.2　クラス定義の中でメンバー関数を定義

```
class A
{
    int m_v;

public:
    explicit A(int v)
      : m_v(v)
    {
    }

    int v() const
    {
        return m_v;
    }
};
```

練習問題　4.3　P.204

1. スコープから抜けた際に、そのスコープで宣言された変数は破棄されます。このときクラスであればデストラクターが呼ばれます。

2. 長くなるので、変更点のみを抜粋します。

▶ リスト App4.3　カウンターが 1 ずつ増えるように修正

```
int get_counter()
{
    static int counter = 0;
    return counter++;
}
```

練習問題　4.4　P.207

1. （省略）

2. 条件文の中で同じ関数呼び出しを何度もするような場合に、1 回の呼び出しだけで済むようにスコープが区切られた変数を宣言できます。また、スコープが `if` 文と同じだけに区切られるので、誤った変数の使い回しによる暗黙の型変換などささいなミスを防ぐことができる。

3. `else: 0` が表示されます。

◆ 解説

初期化構文付き条件分岐のほうがさらに内側にスコープを作るので、そちらの `i`（0）が条件文では使用され、結果 `else` となる。

練習問題　4.5　P.213

1. ビルドは、ソースファイルのプリプロセス・コンパイルを経て生成されたオブジェクトファイルを、最終的にリンクで 1 つの実行形式ファイルにまとめます。

2. 以下のようなプログラムになります。

▶ リスト App4.4　main.cpp

```
extern int value;

void show_value();

int main()
{
    value = 42;
    show_value();
}
```

▶ リスト App4.5　value.cpp

```
#include <iostream>

int value = 0;

void show_value()
{
    std::cout << value << std::endl;
}
```

◆ 解説

変数は宣言と同時に暗黙的に定義がされますが、extern 変数を使うと宣言のみとなります。extern 変数は宣言のみなので、別のソースファイルで実際に使う変数宣言（と定義）をすると、どちらのソースからも同じ変数を参照できるようになります。

練習問題　4.6　P.219

1. 以下のようなプログラムになります。

```
inline int succ(int v)
{
    return v + 1;
}
```

2. 以下のようなプログラムになります。

▶ リスト App4.6　メンバー関数をインライン関数にする

```
class product
{
    int price;

public:
    product(int price) : price(price) { } // 自動インライン化

    inline int get_price() const;
    inline void set_price(int price);
};

int product::get_price() const { return price; }
void product::set_price(int price) { this->price = price; }
```

◆ 解説

メンバー関数をインライン関数にするには、自動インライン化を使うか、メンバー関数の宣言で inline 指定をします。

練習問題　4.7　P.230

1. 以下のようなプログラムになります。

▶ リスト App4.7　module 名前空間とグローバル名前空間

```
#include <iostream>

namespace module
{
    void method()
    {
        std::cout << "module::method()" << std::endl;
    }
}

void method()
{
    std::cout << "method()" << std::endl;
}

int main()
{
    module::method();

    method();
}
```

▶ 実行結果

```
module::method()
method()
```

2. 以下のようなプログラムになります。

▶ リスト App4.8　using 宣言の利用

```
#include <iostream>

namespace module
{
    void method()
    {
        std::cout << "module::method()" << std::endl;
    }
}

void method()
{
    std::cout << "method()" << std::endl;
}

int main()
{
    using module::method;

    method();
}
```

▶ 実行結果

```
module::method()
```

◆ 解説

using 宣言を使うことで名前空間を省略して呼び出すことができるようになりますが、この例ではもともとグローバル名前空間にあった関数と名前が一致しています。このような場合、using 宣言で導入した名前のほうが優先して使用されます。

3. static 変数とするか、無名名前空間を定義しその中で変数宣言します。

練習問題　4.8　P.234

1. C++ の関数を C 言語のソースコードから呼び出せるようにするときに必要となります。

2.（省略）

練習問題　4.9　P.246

1. ヘッダーファイルを取り込み、ヘッダーファイルに書かれた前方宣言やクラス定義などを使える状態にする命令です。

2. 以下のようなプログラムになります。

▶ リスト App4.9　マクロを使って異なる関数呼び出しを行う

```
#include <iostream>

void original()
{
    std::cout << "original()" << std::endl;
}

void injected()
{
    std::cout << "injected()" << std::endl;
}

int main()
{
#define original injected
    original();
}
```

▶ 実行結果

```
injected()
```

3. #undef を使います。

この章の理解度チェック P.246

1. まだ関数の定義がされていなくても、その関数を呼び出せるようになります。

2. 手続きは「リンク」、その際のツールは「リンカー」です。

◆ 解説

リンカーは複数のオブジェクトファイルと、標準ライブラリなどのあらかじめ用意されている機能をリンクして、1 つの実行可能形式ファイルを出力します。

3. 以下のようなプログラムになります。

```
inline int next(int value)
{
    return value + 1;
}
```

4. 以下のようなプログラムになります。

▶ リスト App4.10　ネストした名前空間の別名

```
#include <iostream>

namespace A::B::C
{
    void message()
    {
        std::cout << "A::B::C::message()" << std::endl;
    }
}

namespace abc = A::B::C;

int main()
{
```

```
    abc::message();
}
```

5. 以下のようなプログラムになります。

▶ リスト App4.11　インクルードガード

```
#ifndef HEADER_FILE_IDENTIFER
#defien HEADER_FILE_IDENTIFER

……

#endif
```

第 5 章の解答

練習問題　5.1　P.253

1. 長くなるので、追加点のみを抜粋します。

▶ リスト App5.1　reverse() 関数

```
void reverse(int* array, int size)
{
    for (int i = 0; i < size / 2; ++i)
    {
        int tmp = array[i];
        array[i] = array[size - i - 1];
        array[size - i - 1] = tmp;
    }
}
```

◆ 解説

配列を逆順にするには、i 番目の要素と (長さ - i - 1) 番目の要素を入れ替えます。長さが奇数の場合、長さ ÷ 2 は小数点以下が切り捨てられた値になり、真ん中の要素が処理されないことになりますが、真ん中の要素はどことも入れ替わらないので問題ありません。

2. 以下のようなプログラムになります。

▶ リスト App5.2　ポインターのみを使った配列要素の列挙

```
#include <iostream>

int main()
{
    int array[] = {1, 2, 3, 5, 7, 11};

    for (int* ptr = array; ptr != (array + 6); ++ptr)
    {
        std::cout << *ptr << std::endl;
    }
}
```

▶ 実行結果

```
1
2
3
5
7
11
```

◆ 解説

配列は配列の先頭へのポインターへ暗黙変換ができます。また、ポインターに n を足すと n 個先の要素を指すようになります。for 文で先頭から配列の末尾の次まで走査すると、配列の中身をすべて列挙できます。

練習問題　5.2　P.262

1. 長くなるので、変更点のみを抜粋します。

▶ リスト App5.3　コンパイルできるように修正

```
class product
{
    int         id    = 0;
    std::string name  = "not available";
    int         price = 0;

public:
    explicit product(int id, std::string name, int price)
      : id(id), name(name), price(price) {}
};
```

◆ 解説

配列の長さよりも配列の初期化リストのほうが短かった場合には、不足分はデフォルトコンストラクターを使って初期化されます。NSDMI を使ってデフォルト値を指定しておけば、デフォルトコンストラクターはこれらの値に初期化します。

2. 長くなるので、変更点のみを抜粋します。

▶ リスト App5.4　std::vector を使って書き直し

```
#include <vector>

int main()
{
    std::vector<product> p =
    {
        product{1, "smart phone", 60000},
        product{2, "tablet", 35000},
    };

    p.push_back(product{});
    p.push_back(product{});
}
```

◆ 解説

push_back() メンバー関数と pop_back() メンバー関数を使うと、std::vector に要素を追加したり削除したりできます。

練習問題　5.3　P.265

1. 以下のようなプログラムになります。

▶ リスト App5.5　new 演算子による動的確保

```
#include <iostream>

class A
{
public:
    A()
    {
        std::cout << "コンストラクター" << std::endl;
    }
    ~A()
```

```
    {
        std::cout << "デストラクター" << std::endl;
    }
};

int main()
{
    std::cout << "動的確保前" << std::endl;

    A* ptr = new A{}; // 動的確保

    std::cout << "動的確保後" << std::endl;

    std::cout << "解放前" << std::endl;

    delete ptr;

    std::cout << "解放後" << std::endl;
}
```

▶ 実行結果

```
動的確保前
コンストラクター
動的確保後
解放前
デストラクター
解放後
```

2. 動的確保したインスタンスが破棄されずに残り続けるメモリリークが起きます。以下に実行結果を示します。

▶ 実行結果

```
動的確保前
コンストラクター
動的確保後
解放前
解放後
```

練習問題　5.4　P.270

1. 以下のようなプログラムになります。

▶ リスト App5.6　new 演算子で動的確保したときにコンストラクターを呼び分ける

```
#include <iostream>

class A
{
public:
    A()
    {
        std::cout << "A::A()" << std::endl;
    }

    explicit A(int i)
    {
        std::cout << "A::A(" << i << ")" << std::endl;
    }

    explicit A(int i, int j)
    {
        std::cout << "A::A(" << i << ", " << j << ")" << std::endl;
    }
};

int main()
{
    A* ptr = new A{};

    delete ptr;

    ptr = new A{42};

    delete ptr;

    ptr = new A{3, 10};

    delete ptr;
}
```

▶ 実行結果

```
A::A()
A::A(42)
A::A(3, 10)
```

2. 長くなるので、変更点のみを抜粋します。

▶ リスト App5.7　配列の動的確保

```
int main()
{
    A* array = new A[3]
    {
        A{42},
        A{3, 10},
    };

    delete [] array;
}
```

▶ 実行結果

```
A::A(42)
A::A(3, 10)
A::A()
```

◆ 解説

配列の new 演算子は確保された配列の先頭から順番に初期化リストで与えられた初期値で初期化します。
初期化リストに不足している部分がある場合にはデフォルトコンストラクターを使って初期化します。

練習問題　5.5　P.275

1. {}を使った初期化では、精度が悪くなる暗黙の型変換（縮小変換）が禁止されているためです。

2. 以下のようなプログラムになります。

▶ リスト App5.8　std::initializer_list<int>を受け取るコンストラクター

```
#include <iostream>
#include <initializer_list>

class A
{
public:
    explicit A(std::initializer_list<int> list)
    {
        for (int e : list)
        {
            std::cout << e << std::endl;
        }
    }
};

int main()
{
    A a{0, 1, 2, 3, 4};
}
```

▶ 実行結果

```
0
1
2
3
4
```

練習問題　5.6　P.277

1. それぞれ、値渡し（call by value）と参照渡し（call by reference）と呼びます。

練習問題　5.7　P.282

1. ローカル変数への参照を関数が返しており、main() 関数で参照を受け取った段階では破棄されてしまっています。

2. 以下のようなプログラムになります。

▶ リスト App5.9　メンバー変数への参照を直接返すメンバー関数を追加

```
class A
{
    int m_value;

public:
    int& value()
    {
        return m_value;
    }

    const int& value() const
    {
        return m_value;
    }
};
```

練習問題　5.8　P.290

1. 以下のようなプログラムになります。

▶ リスト App5.10　右辺値参照を受け取る関数

```
#include <iostream>
#include <utility>

void show(int&& rref)
{
    std::cout << rref << std::endl;
}

int main()
{
    int i = 42;

    show(std::move(i));
}
```

◆ 解説

変数は左辺値なのでそのままで右辺値参照へ渡すことができません。<utility>ヘッダーで提供されている std::move() 関数を使うことで左辺値を強制的に右辺値参照に変換できます。

2. 以下のようなプログラムになります。

▶ リスト App5.11　ムーブコンストラクターとコピーコンストラクター

```
#include <iostream>
#include <utility>

class A
{
public:
    // 何かコンストラクターが定義されると
    // デフォルトコンストラクターは自動生成されないので定義する
    A() { }
```

```
    A(const A& other)
    {
        std::cout << "コピーコンストラクター" << std::endl;
    }

    A(A&& other)
    {
        std::cout << "ムーブコンストラクター" << std::endl;
    }
};

int main()
{
    A a;

    A c(a); // コピー

    A m(std::move(a)); // ムーブ
}
```

▶ 実行結果

```
コピーコンストラクター
ムーブコンストラクター
```

練習問題　5.9　P.295

1. 以下のようなプログラムになります。

▶ リスト App5.12　関数ポインター

```
#include <iostream>

void message()
{
    std::cout << "Hello, function pointer" << std::endl;
}
```

```
int main()
{
    void (*fp)() = message;

    fp();
}
```

▶ 実行結果

```
Hello, function pointer
```

2. 高階関数とは、関数ポインターや関数リファレンスを引数で受け取るか、戻り値として返す関数のことです。処理の内容を関数によってカスタマイズすることができます。

練習問題　5.10　P.299

1. 長くなるので変更点のみを抜粋します。

▶ リスト App5.13　コンパイルできるように修正

```
void A::foo()
{
    auto lambda = [this]()
    {
        std::cout << value << std::endl;
    };

    lambda();
}
```

◆ 解説

this ポインターをキャプチャしていないとラムダ式はメンバー変数やメンバー関数にアクセスすることができません。

2. 以下のようなプログラムになります。

▶ リスト App5.14　インスタンス全体のコピーのキャプチャ

```
#include <iostream>

class A
{
    int value;

public:
    void set(int value) { this->value = value; }
    void foo();
};

void A::foo()
{
    // auto lambda = [*this]() // インスタンスのコピーをキャプチャ
    auto lambda = [this]() // this ポインターをキャプチャ
    {
        std::cout << value << std::endl;
    };

    value = 0;

    lambda();
}

int main()
{
    A a;
    a.set(42);
    a.foo();
}
```

インスタンスのコピーをキャプチャする場合の実行結果は、以下のとおりです。

▶ 実行結果

```
42
```

this ポインターをキャプチャする場合の実行結果は、以下のとおりです。

▶ 実行結果

```
0
```

◆ 解説

this ポインターのキャプチャでは個々のメンバー変数はキャプチャされないので、インスタンスに変更が加えられると参照キャプチャのようにラムダ式にも影響を及ぼします。インスタンス全体のコピーをキャプチャすると、すべてのメンバー変数のコピーまでキャプチャするのでメンバー変数の変更が相互に作用しません。

練習問題　5.11　P.301

1. 何もキャプチャしないラムダ式は普通の関数と同じ動作をするので、関数ポインターに暗黙の型変換できるようになっています。

この章の理解度チェック　P.302

1. 以下のようなプログラムになります。

```
int* allocate(int n)
{
    return new int[n]{};
}
```

◆ 解説

配列の動的確保の際に渡される初期化リストの長さが、実際に確保された配列よりも短い場合には 0 で初期化されます。そのためすべての要素を 0 で初期化したい場合は、初期化リストの中身をすべて省略するだけで実現できます（初期化リスト自体を省略してはいけません）。

2. 以下のようなプログラムになります。

```
void clear(int& variable)
{
    variable = 0;
}
```

3. 以下のようなプログラムになります。

▶ リスト App5.15　参照を受け取る関数と右辺値参照を受け取る関数のオーバーロード

```
#include <iostream>
#include <utility>

void show(int& value)
{
    std::cout << "参照" << std::endl;
}

void show(int&& value)
{
    std::cout << "右辺値参照" << std::endl;
}

int main()
{
    int i = 0;

    show(i); // 参照

    show(std::move(i)); // 右辺値参照
}
```

▶ 実行結果

```
参照
右辺値参照
```

4. 長くなるので、追加点のみを抜粋します。

▶ リスト App5.16　enumerate() 関数

```
void enumerate(int* first, int* last, void (*func)(int))
{
    for ( ; first != last; ++first)
    {
        func(*first);
    }
}
```

5. 長くなるので、変更点のみを抜粋します。

▶ リスト App5.17　show() 関数をラムダ式に書き換え

```
int main()
{
    int array[] = {1, 2, 3, 5, 7, 11, 13};

    std::size_t length = sizeof(array) / sizeof(array[0]);
    enumerate(array, array + length, [](int v)
    {
        std::cout << v << std::endl;
    });
}
```

◆ 解説

キャプチャを持たないラムダ式は関数ポインターへ暗黙変換できるので、実引数として直接渡すことができます。

第 6 章の解答

練習問題　6.2　P.310

1. 以下のようなプログラムになります。

▶ リスト App6.1　Float クラス

```
class Float
{
    float value = 0.0f;

public:
    Float() { }
    Float(float f) : value(f) { }

    Float operator+(Float rhs) const;
    Float operator-(Float rhs) const;
};

Float Float::operator+(Float rhs) const
{
    return Float{value + rhs.value};
}

Float Float::operator-(Float rhs) const
{
    return Float{value - rhs.value};
}
```

2. 長くなるので、追加点のみを抜粋します。

▶ リスト App6.2　乗算と除算演算子のオーバーロード

```
class Float
{
    float value = 0.0f;

public:
    Float() { }
    Float(float f) : value(f) { }
```

```
    Float operator+(Float rhs) const;
    Float operator-(Float rhs) const;
    Float operator*(Float rhs) const;
    Float operator/(Float rhs) const;
};

Float operator*(Float rhs) const
{
    return Float{value * rhs.value};
}

Float operator/(Float rhs) const
{
    return Float{value / rhs.value};
}
```

練習問題　6.3　P.314

1. 長くなるので、追加点のみを抜粋します。

▶ リスト App6.3　正負を表す + と-演算子のオーバーロード

```
class Float
{
    float value = 0.0f;

public:
    Float() { }
    Float(float f) : value(f) { }

    Float operator+(Float rhs) const;
    Float operator-(Float rhs) const;
    Float operator*(Float rhs) const;
    Float operator/(Float rhs) const;

    const Float& operator+() const;
    Float operator-() const;
};

const Float& Float::operator+() const
{
```

```
    return *this;
}

Float Float::operator-() const
{
    return Float{-value};
}
```

◆ 解説

+ は符号反転などを行わないので、自分自身への参照を返しても正しい値となります。

2. 以下のようなプログラムになります。

▶ リスト App6.4　前置／後置のインクリメント演算子のオーバーロード

```
#include <iostream>

class A
{
public:
    A& operator++();   // 前置
    A operator++(int); // 後置
};

A& A::operator++()
{
    std::cout << "前置インクリメント演算子" << std::endl;
    return *this;
}

A A::operator++(int)
{
    auto tmp = *this;
    std::cout << "後置インクリメント演算子" << std::endl;
    return tmp;
}

int main()
{
    A a;
```

```
    ++a;

    a++;
}
```

◆ 解説

後置のインクリメント演算子をオーバーロードするには、int 型を受け取るダミーの仮引数を 1 つ追加する必要があります。

練習問題　6.5　P.321

1. 以下のようなプログラムになります。

▶ リスト App6.5　小なり演算子と論理演算子による関係演算子の表現

```
bool operator==(T a, T b)
{
    return !(a < b) && !(b < a);
}

bool operator!=(T a, T b)
{
    return (a < b) || (b < a);
}

bool operator<=(T a, T b)
{
    return !(b < a);
}

bool operator>(T a, T b)
{
    return b < a;
}

bool operator>=(T a, T b)
{
    return !(a < b);
```

```
}
```

練習問題　6.6　P.322

1. 配列のように動作するコンテナクラスや、連想配列を作ることができます。

練習問題　6.7　P.329

1. std::unique_ptr はスマートポインターの一種であり、スコープから抜けて std::unique_ptr が破棄されると同時に、管理している動的確保したメモリ領域を delete 演算子で破棄してくれます。

▶ リスト App6.6　std::unique_ptr の利用例

```
#include <memory>
#include <iostream>

class A
{
public:
    A()
    {
        std::cout << "コンストラクター" << std::endl;
    }

    ~A()
    {
        std::cout << "デストラクター" << std::endl;
    }
};

int main()
{
    auto ptr = std::make_unique<A>();
}
```

▶ 実行結果

```
コンストラクター
デストラクター
```

練習問題　6.8　P.331

1. 関数呼び出し演算子がオーバーロードされたクラスから作られたオブジェクトのことを関数オブジェクトと呼びます。

練習問題　6.9　P.334

1. 二項演算子のうち、右辺と左辺を入れ替えたときに同じ結果になることが期待される演算子です。

◆ 解説

メンバー関数として演算子オーバーロードをすると、左辺に演算子オーバーロードを追加したクラスがあるときにしかオーバーロードが呼ばれません。フレンド関数で実装すると右辺にクラスがある場合についてもオーバーロードすることができます。

2. 以下のようなプログラムになります。

▶ リスト App6.7　演算子オーバーロードをフレンド関数として定義

```
class Float
{
    float value = 0.0f;

public:
    Float() { }
    Float(float f) : value(f) { }

    friend Float operator+(Float lhs, Float rhs);
```

```
    friend Float operator-(Float lhs, Float rhs);
    friend Float operator*(Float lhs, Float rhs);
    friend Float operator/(Float lhs, Float rhs);
};

Float operator+(Float lhs, Float rhs)
{
    return Float{lhs.value + rhs.value};
}

Float operator-(Float lhs, Float rhs)
{
    return Float{lhs.value - rhs.value};
}

Float operator*(Float lhs, Float rhs)
{
    return Float{lhs.value * rhs.value};
}

Float operator/(Float lhs, Float rhs)
{
    return Float{lhs.value / rhs.value};
}
```

練習問題　6.10　P.338

1. 長くなるので、追加点のみを抜粋します。

▶ リスト App6.8　コピー代入演算子

```
class Float
{
    float value = 0.0f;

public:
    Float() { }
    Float(float f) : value(f) { }

    Float& operator=(const Float& other);
```

```
    friend Float operator+(Float lhs, Float rhs);
    friend Float operator-(Float lhs, Float rhs);
    friend Float operator*(Float lhs, Float rhs);
    friend Float operator/(Float lhs, Float rhs);
};

Float& Float::operator=(const Float& other)
{
    value = other.value;
    return *this;
}
```

練習問題　6.11　P.345

1. 変換関数を explicit 指定します。

2. 長くなるので、変更点のみを抜粋します。

```
int main()
{
    B b;

    static_cast<const A&>(b).foo();
}
```

この章の理解度チェック　P.345

1. 演算子オーバーロードをしても、演算子の項の数、優先順位および結合順序は変更できません。また、項が組み込み型のみの演算子オーバーロードはエラーとなります。

2. ダミーの int 型引数を 1 つ受け取るようにします。

3. 以下のようなプログラムになります。

▶ リスト App6.9　四則演算ができる整数クラス

```
class Integer
{
    int value = 0;

public:
    Integer() { }
    Integer(int value) : value(value) { }

    friend Integer operator+(Integer lhs, Integer rhs);
    friend Integer operator-(Integer lhs, Integer rhs);
    friend Integer operator*(Integer lhs, Integer rhs);
    friend Integer operator/(Integer lhs, Integer rhs);
};

Integer operator+(Integer lhs, Integer rhs)
{
    return Integer{lhs.value + rhs.value};
}

Integer operator-(Integer lhs, Integer rhs)
{
    return Integer{lhs.value - rhs.value};
}

Integer operator*(Integer lhs, Integer rhs)
{
    return Integer{lhs.value * rhs.value};
}

Integer operator/(Integer lhs, Integer rhs)
{
    return Integer{lhs.value / rhs.value};
}
```

4. 以下のようなプログラムになります。

▶ リスト App6.10　スマートポインター

```
class A
{
};

class smart_ptr
{
    A* ptr = nullptr;

public:
    explicit smart_ptr(A* ptr) : ptr(ptr) { }

    ~smart_ptr()
    {
        delete ptr;
    }

    A* operator->() const
    {
        return ptr;
    }

    A& operator*() const
    {
        return *ptr;
    }
};
```

第 7 章の解答

練習問題　7.1　P.351

1. 基底クラスを private で継承すると、基底クラスの公開メンバーは派生クラスの非公開メンバーとして継承されます。

練習問題　7.2　P.354

1. 以下のようなプログラムになります。コンストラクターは最も基底にあるクラスのものから順番に、より派生されたものが呼ばれます。また、デストラクターはコンストラクターとは逆順に呼ばれます。

▶ リスト App7.1　派生／基底クラスのコンストラクター／デストラクターが呼ばれる順

```
#include <iostream>

class Base
{
public:
    Base()
    {
        std::cout << "Base コンストラクター" << std::endl;
    }

    ~Base()
    {
        std::cout << "Base デストラクター" << std::endl;
    }
};

class Derived : public Base
{
public:
    Derived()
    {
        std::cout << "Derived コンストラクター" << std::endl;
    }

    ~Derived()
    {
```

```
        std::cout << "Derived デストラクター" << std::endl;
    }
};

int main()
{
    Derived derived;
}
```

▶ 実行結果

```
Base コンストラクター
Derived コンストラクター
Derived デストラクター
Base デストラクター
```

2. 以下のようなプログラムになります。

▶ リスト App7.2　派生クラスのコンストラクターから、基底クラスの引数があるコンストラクターを呼び出す

```
class Base
{
public:
    explicit Base(int value)
    {
        std::cout << "Base(" << value << ")" << std::endl;
    }
};

class Derived : public Base
{
public:
    Derived() : Base(0) // 基底クラスのコンストラクター呼び出し
    {
    }
};
```

◆ 解説

メンバー初期化リストにコンストラクターに渡す引数とともに列挙すると、基底クラスのコンストラクターが呼び出されます。

練習問題　7.3　P.357

1. 派生クラスのポインターは基底クラスのポインターに暗黙変換できます。しかし基底クラスへのポインターでできることは基底クラスでできることのみであり、派生クラスのメンバー関数などを呼び出すことはできません。

練習問題　7.4　P.364

1. 以下のようなプログラムになります。

▶ リスト App7.3　基底クラスのポインターからの仮想関数呼び出し

```
#include <iostream>

class Base
{
public:
    virtual void show_class_name() const
    {
        std::cout << "Base" << std::endl;
    }
};

class Derived : public Base
{
public:
    void show_class_name() const override
    {
        std::cout << "Derived" << std::endl;
    }
};
```

```
void show_class(Base& base)
{
    base.show_class_name();
}

int main()
{
    Derived derived;

    show_class(derived);
}
```

▶ 実行結果

```
Derived
```

◆ 解説

仮想関数のオーバーライドは基底クラスの動作を変更するので、基底クラスへのポインターを使った呼び出しでも、派生クラスでオーバーライドしたメンバー関数が呼び出されます。

2. 派生クラスから直接基底クラスの仮想関数を呼び出すことはできませんが、オーバーライドはできます。

3. 以下のようなプログラムになります。

▶ リスト App7.4　仮想デストラクターが必要になる例

```
#include <iostream>

class Base
{
public:
    virtual ~Base()
    {
        std::cout << "Base デストラクター" << std::endl;
    }
```

```
};

class Derived : public Base
{
public:
    ~Derived()
    {
        std::cout << "Derived デストラクター" << std::endl;
    }
};

void deallocate(Base* base)
{
    delete base;
}

int main()
{
    auto ptr = new Derived{};

    deallocate(ptr);
}
```

練習問題　7.5　P.371

1. 以下のようなプログラムになります。

▶ リスト App7.5　2 種類の基底クラスを多重継承した派生クラス

```
#include <iostream>

class BaseA
{
public:
    void method_from_A()
    {
        std::cout << "BaseA::method_from_A()" << std::endl;
    }
};
```

```
class BaseB
{
public:
    void method_from_B()
    {
        std::cout << "BaseB::method_from_B()" << std::endl;
    }
};

// 多重継承
class Derived : public BaseA, public BaseB
{
};

int main()
{
    Derived derived;

    derived.method_from_A();

    derived.method_from_B();
}
```

▶ 実行結果

```
BaseA::method_from_A()
BaseB::method_from_B()
```

2. 継承した際に列挙した順番で呼ばれます。

◆ 解説

コンストラクターのメンバー初期化リストに書かれた順番ではないことに注意してください。

練習問題　7.6　P.377

1. 以下のようなプログラムになります。

▶ リスト App7.6　ひし形継承

```
#include <iostream>
#include <string>

class Base
{
public:
    explicit Base(std::string name)
    {
        std::cout << "Base(\"" << name << "\")" << std::endl;
    }
};

class DerivedA : virtual public Base
{
public:
    DerivedA() : Base("DerivedA") { }
};

class DerivedB : virtual public Base
{
public:
    DerivedB() : Base("DerivedB") { }
};

class MostDerived : public DerivedA, public DerivedB
{
public:
    MostDerived() : Base("MostDerived") { }
};

int main()
{
    MostDerived md;
}
```

▶ 実行結果

```
Base("MostDerived")
```

練習問題　7.7　P.379

1. クラスを final 指定します。

2. 以下のようなプログラムになります。

▶ リスト App7.7　それ以上のオーバーライドを禁止した派生クラス

```
class Base
{
public:
    virtual void method();
};

class Derived : public Base
{
public:
    void method() final;
};
```

この章の理解度チェック　P.379

1. 以下のようなプログラムになります。

▶ リスト App7.8　仮想関数が基底クラスの動作を変える

```
#include <iostream>

class Base
{
public:
```

```
    virtual std::string name() const
    {
        return "Base";
    }
};

class Derived : public Base
{
public:
    std::string name() const override
    {
        return "Derived";
    }
};

void show_name(Base& base)
{
    std::cout << base.name() << std::endl;
}

int main()
{
    Derived derived;

    show_name(derived);
}
```

▶ 実行結果

```
Derived
```

2. オブジェクトを動的確保したあと、そのオブジェクトの基底クラスへのポインターを使って delete 演算子を呼び出すような場面です。

3. 同じ基底クラスから派生した 2 つのクラスから、さらにそれらを基底クラスとして多重継承したとき、これをひし形継承と言います。ひし形継承では、インスタンスの中で基底クラスがただ 1 つだけ存在するようになっています。

第 8 章の解答

練習問題　8.2　P.390

1. 以下のようなプログラムになります。

▶ リスト App8.1　科学技術表記による円周率表示

```
#include <iostream>

int main()
{
    const double pi = 3.14159265359;

    std::cout.setf(std::ios::scientific);

    std::cout << pi << std::endl;
}
```

▶ 実行結果

```
3.141593e+00
```

◆ 解説

setf() メンバー関数を使うことで書式を変更できます。

2. 以下のようなプログラムになります。

▶ リスト App8.2　整数の表示

```
#include <iostream>

int main()
{
    std::cout.setf(std::ios::right | std::ios::oct, std::ios::basefield);
```

```
    std::cout.width(8);
    std::cout.fill('-');

    std::cout << 1234 << std::endl;
}
```

▶ 実行結果

```
----2322
```

◆ 解説

setf() メンバー関数の第 2 引数には、フラグをセットする前にクリアするべき値のマスクを指定できます。width() メンバー関数は桁数を、fill() メンバー関数は充填文字を設定します。

練習問題　8.3　P.391

1. 以下のようなプログラムになります。

▶ リスト App8.3　空行が入力されるまで入力された行を出力

```
#include <iostream>

int main()
{
    char buffer[1024];

    while (true)
    {
        std::cout << "> ";

        std::cin.getline(buffer, sizeof(buffer));
        if (buffer[0] == '\0')
        {
            // バッファーの先頭がヌル文字だった場合、空行となる
            break;
```

```
        }

        std::cout << "入力された行は \"" << buffer << "\" です" << std::endl;
    }

    std::cout << "終了" << std::endl;
}
```

▶ 実行結果

```
> one liner
入力された行は "one liner" です
> second line
入力された行は "second line" です
>
終了
```

◆ 解説

空行（0 文字の行）であっても getline() メンバー関数はヌル文字で終端します。

練習問題　8.4　P.393

1. 以下のようなプログラムになります。

▶ リスト App8.4　マニピュレーターを利用した科学技術表記による円周率表示

```
#include <iostream>
#include <iomanip>

int main()
{
    const double pi = 3.14159265359;

    std::cout << std::scientific << pi << std::endl;
}
```

▶ 実行結果

```
3.141593e+00
```

2. 以下のようなプログラムになります。

▶ リスト App8.5　マニピュレータを使用した整数表示

```
#include <iostream>
#include <iomanip>

int main()
{
    std::cout << std::right << std::oct
              << std::setw(8) << std::setfill('0')
              << 1234 << std::endl;
}
```

▶ 実行結果

```
00002322
```

練習問題　8.5　P.400

1. std::fstream 型です。

2. 以下のようなプログラムになります。

▶ リスト App8.6　ファイルを読み込んで画面に表示

```
#include <fstream>
#include <iostream>
```

```
int main()
{
    std::ifstream in{__FILE__};

    std::string line;
    while (!in.eof())
    {
        std::getline(in, line);
        std::cout << line << std::endl;
    }
}
```

◆ 解説

__FILE__はソースファイル名に置き換わる識別子なので、これを直接開くことでソースファイル自身を開くことができます。ソースファイル自身を出力するプログラムのことをクワイン（Quine）と呼びますが、この例のようにファイルストリームを使った場合には一般にクワインとは見なされません。

練習問題　8.6　P.404

1. 以下のようなプログラムになります。

▶ リスト App8.7　バイナリのファイル出入力

```
#include <iostream>
#include <fstream>

int main()
{
    const char message[] = "Hello, block IO";

    std::ofstream out{"block.bin", std::ios::binary};

    out.write(message, sizeof(message));

    out.close();

    std::ifstream in{"block.bin", std::ios::binary};

    char buffer[1024] = {};
```

```
    in.read(buffer, sizeof(buffer));

    for (auto remain = in.gcount(); remain--; )
    {
        if (message[remain] != buffer[remain])
        {
            std::cout << "不一致" << std::endl;
            break;
        }
    }

    std::cout << "終了" << std::endl;
}
```

▶ 実行結果

```
終了
```

◆ 解説

直前のブロック入力で入力したバイト数は、gcount() メンバー関数を使うことで得られます。

練習問題　8.7　P.407

1. 以下のようなプログラムになります。

▶ リスト App8.8　ファイル出力

```
#include <fstream>

int main()
{
    std::ofstream out{"seek.txt"};

    out << "Hello, world" << std::endl;
```

```
    out.seekp(7, std::ios::beg);

    out << "file seek" << std::endl;
}
```

◆ 解説

出力用ファイルストリームでは、seekp() メンバー関数を使うことでファイルのシークができます。入力用ファイルストリームでシークするには seekg() メンバー関数を使用します。

練習問題　8.8　P.411

1. 以下のようなプログラムになります。

▶ リスト App8.9　ファイル末尾への移動

```
#include <iostream>
#include <iomanip>
#include <fstream>

int main()
{
    std::ifstream in{__FILE__};

    in.seekg(0, std::ios::end);

    std::cout << in.rdstate() << std::endl;
    std::cout << "eof: " << std::boolalpha
        << bool(in.rdstate() & std::ios::eofbit) << std::endl;
    std::cout << "fail: " << std::boolalpha
        << bool(in.rdstate() & std::ios::failbit) << std::endl;
    std::cout << "bad: " << std::boolalpha
        << bool(in.rdstate() & std::ios::badbit) << std::endl;

    std::cout << std::endl;

    in.get();

    std::cout << in.rdstate() << std::endl;
    std::cout << "eof: " << std::boolalpha
```

```
        << bool(in.rdstate() & std::ios::eofbit) << std::endl;
    std::cout << "fail: " << std::boolalpha
        << bool(in.rdstate() & std::ios::failbit) << std::endl;
    std::cout << "bad: " << std::boolalpha
        << bool(in.rdstate() & std::ios::badbit) << std::endl;
}
```

▶ 実行結果

```
0
eof: false
fail: false
bad: false

6
eof: true
fail: true
bad: false
```

この章の理解度チェック P.411

1. 以下のようなプログラムになります。

▶ リスト App8.10　任意の整数の表示

```
#include <iostream>
#include <iomanip>

int main()
{
    //  メンバー関数を使用する方法

    std::cout.setf(std::ios::hex, std::ios::basefield);
    std::cout.setf(std::ios::left, std::ios::adjustfield);
    std::cout.width(16);
    std::cout.fill('=');
```

```
    std::cout << 0xdeadbeef << std::endl;

    // マニピュレータを使用する方法

    std::cout << std::hex << std::left
              << std::setw(16) << std::setfill('=')
              << 0xdeadbeef << std::endl;
}
```

▶ 実行結果

```
deadbeef========
deadbeef========
```

2. 以下のようなプログラムになります。

▶ リスト App8.11　入力された名前のファイルからの読み込み

```
#include <iostream>
#include <fstream>
#include <string>

int main()
{
    std::string line;

    std::cout << "> "; // プロンプト
    std::getline(std::cin, line);

    std::ifstream in{line};

    std::getline(in, line);
    std::cout << line << std::endl;
}
```

▶ 実行結果

```
> prog.cpp
#include <iostream>
```

3. 以下のようなプログラムになります。

▶ リスト App8.12　バイナリのブロック入力

```
#include <fstream>

int main()
{
    std::ifstream in{"input.bin", std::ios::binary};

    char buffer[100];
    in.read(buffer, sizeof(buffer));
}
```

◆ 解説

バイナリ入出力するときにはファイルストリームをバイナリモードで開きます。

4. tellg() メンバー関数を使うことで、入力ストリームの現在の位置を取得できます。

◆ 解説

出力ストリームの現在位置を取得するには tellp() メンバー関数を使用します。

第 9 章の解答

練習問題　9.1　P.420

1. 以下のようなプログラムになります。

```
template <typename T>
T id(T value)
{
    return value;
}
```

2. 長くなるので、追加部分のみを抜粋します。

▶ リスト App9.1　id() 関数の呼び出し

```
#include <iostream>
#include <string>

int main()
{
    std::cout << id<int>(0) << std::endl;

    std::cout << id<std::string>("Hello, template") << std::endl;
}
```

練習問題　9.2　P.424

1. 以下のようなプログラムになります。

```
template <typename T>
class A
{
public:
    void member()
    {
        std::cout << "A<T>::member()" << std::endl;
    }
};
```

2. 以下のようなプログラムになります。

▶ リスト App9.2　クラステンプレートのメンバー関数の定義

```
template <typename T>
class A
{
public:
    void member();
};

template <typename T>
void A<T>::member()
{
    std::cout << "A<T>::member()" << std::endl;
}
```

練習問題　9.3　P.429

1. 以下のようなプログラムになります。

▶ リスト App9.3　クラステンプレートでの仮想関数の使用

```
#include <iostream>

class Base
{
public:
    virtual void show_message() const
    {
        std::cout << "Base のメンバー関数" << std::endl;
    }
};

template <typename T>
class Derived : public Base
{
public:
    void show_message() const override
    {
```

```
        std::cout << "Derived<T>のメンバー関数" << std::endl;
    }
};

int main()
{
    Derived<int> d;

    Base& b = d;

    b.show_message();
}
```

▶ 実行結果

```
Derived<T>のメンバー関数
```

2. 関数テンプレートは呼び出すときにテンプレートパラメーターが決まってコピーが作られるので、あらかじめ型が決まっている必要がある仮想関数テーブルに登録することができません。

3. クラステンプレートが実体化されるたびに static メンバー変数もそれぞれ実体化されます。そのため、実体化されたクラスの中では共通の変数として使えますが、クラステンプレート全体で共通の変数としては使えません。

練習問題　9.4　P.436

1. 以下のようなプログラムになります。

▶ リスト App9.4　関数テンプレートの明示的特殊化

```
#include <iostream>

template <typename T>
void show_message(T value)
{
    std::cout << "Hello, primary template (" << value << ")" << std::endl;
}

template <>
void show_message<int>(int value)
{
    std::cout << "Hello, specialization (" << value << ")" << std::endl;
}

int main()
{
    show_message<double>(3.14159265);

    show_message<int>(42);
}
```

▶ 実行結果

```
Hello, primary template (3.14159)
Hello, specialization (42)
```

2. 以下のようなプログラムになります。

▶ リスト App9.5　メンバー変数を持つクラステンプレートの明示的特殊化

```
#include <iostream>

template <typename T>
class A
{
    T value;

public:
    explicit A(T value) : value(value) {}

    void show_value() const
    {
        std::cout << value << std::endl;
    }
};

template <>
class A<void>
{
public:
    void show_value() const
    {
        std::cout << "(void)" << std::endl;
    }
};

int main()
{
    A<int> ai{42};
    ai.show_value();

    A<void> av;
    av.show_value();
}
```

▶ 実行結果

```
42
(void)
```

3. 部分特殊化はクラステンプレートでのみ使える機能であり、テンプレートパラメーターの一部分だけを指定したテンプレートを作る点が異なります。

練習問題　9.5　P.446

1. 関数の戻り値の型を return 文から推論します。

2. 以下のようなプログラムになります。

▶ リスト App9.6　型推論を使った関数テンプレート呼び出し

```
#include <iostream>

template <typename T>
void show_value(T value)
{
    std::cout << value << std::endl;
}

int main()
{
    show_value(0);

    show_value(42.195f);
}
```

▶ 実行結果

```
0
42.195
```

3. 以下のようなプログラムになります。

▶ リスト App9.7　クラステンプレートを関数の仮引数にする

```
#include <iostream>

template <typename T>
class A
{
public:
    void method()
    {
        std::cout << "A<T>::method()" << std::endl;
    }
};

template <typename T>
void call_method(A<T> a)
{
    a.method();
}

int main()
{
    A<int> a;

    call_method(a);
}
```

▶ 実行結果

```
A<T>::method()
```

練習問題　9.6　P.448

1. 以下のようなプログラムになります。

▶ リスト App9.8　関数テンプレートのテンプレートパラメーターで整数を受け取る

```
#include <iostream>

template <int i>
void show_parameter()
{
    std::cout << i << std::endl;
}

int main()
{
    show_parameter<42>();
}
```

▶ 実行結果

```
42
```

2. 以下のようなプログラムになります。

▶ リスト App9.9　整数以外も受け取れるように変更

```
#include <iostream>

template <auto i>
void show_parameter()
{
    std::cout << i << std::endl;
}

int main()
{
    show_parameter<42>();
```

```
}
```

▶ 実行結果

```
42
```

練習問題　9.7　P.455

1. 以下のようなプログラムになります。

▶ リスト App9.10　常に const 参照で引数を受け取る関数テンプレート

```
#include <iostream>

template <typename T>
void show_value(const T& value)
{
    std::cout << value << std::endl;
}

int main()
{
    const int i = 42;

    show_value(i);
}
```

▶ 実行結果

```
42
```

2. 以下のようなプログラムになります。

▶ リスト App9.11　フォワーディング参照を使った関数テンプレート

```
#include <iostream>

template <typename T>
void show_value(T&& value)
{
    std::cout << value << std::endl;
}

int main()
{
    const int i = 42;

    show_value(i);

    show_value(2.71828181846);
}
```

▶ 実行結果

```
42
2.71828
```

3. <utility>ヘッダーで提供されている std::forward() 関数です。この関数を使うことで、フォワーディング参照で受け取った引数を、右辺値なら右辺値として、左辺値なら左辺値として別の関数に渡すことができます。

練習問題　9.8　P.462

1. 依存名の前に typename キーワードを付けます。

2. 以下のようなプログラムになります。

▶ リスト App9.12　エイリアステンプレート

```
#include <iostream>

template <typename T>
class A
{
    T value;

public:
    using value_type = T;

    explicit A(T value) : value(value) {}

    T& get_value() { return value; }
};

template <typename T>
using A_value = typename A<T>::value_type;

template <typename T>
A_value<T>& get(A<T>& a)
{
    return a.get_value();
}

int main()
{
    A<int> a{42};

    std::cout << get(a) << std::endl;
}
```

▶ 実行結果

```
42
```

練習問題　9.9　P.466

1. 以下のようなプログラムになります。

▶ リスト App9.13　可変引数テンプレート

```
#include <iostream>

template <typename... T>
void show_length(T... value)
{
    std::cout << sizeof...(value) << std::endl;
}

int main()
{
    show_length(0, 1);

    show_length(2, 3, 4, 5, 6);
}
```

▶ 実行結果

```
2
5
```

2. 以下のようなプログラムになります。

▶ リスト App9.14　テンプレートパラメーターパックの展開

```
#include <iostream>

void expanded(int a)
{
    std::cout << "{" << a << "}" << std::endl;
}

void expanded(int a, int b)
```

```
{
    std::cout << "{" << a << ", " << b << "}" << std::endl;
}

void expanded(int a, int b, int c)
{
    std::cout << "{" << a << ", " << b << ", " << c << "}" << std::endl;
}

template <typename... T>
void expanding(T... args)
{
    expanded(args...);
}

int main()
{
    expanding(0);

    expanding(10, 2);

    expanding(72, 42, 11);
}
```

▶ 実行結果

```
{0}
{10, 2}
{72, 42, 11}
```

練習問題　9.10　P.471

1. static_assert を使います。条件文が false となる場合にコンパイルエラーにすることができます。

2. 一例を示します。

- std::is_class<T>は、T がクラス型の場合に static メンバー変数 value が true を返す

練習問題　9.11　P.473

1. ジェネリックラムダ式とは、ラムダ式の引数の型に auto と書いたものであり、引数の型を呼び出したときの実引数から推論するラムダ式のことです。

2. 以下のようなプログラムになります。

▶ リスト App9.15　第 1 引数を表示するジェネリックラムダ式

```
#include <iostream>

int main()
{
    auto lambda = [](auto v)
    {
        std::cout << v << std::endl;
    };

    lambda(42);

    lambda("Hello, generic lambda");
}
```

▶ 実行結果

```
42
Hello, generic lambda
```

練習問題　9.12　P.476

1. 以下のようなプログラムになります。

▶ リスト App9.16　クラステンプレートのテンプレートパラメーターの型推論

```
#include <iostream>

template <typename T, typename U>
class A
{
public:
    explicit A(T l, U r) {}
};

void show(const A<int, const char*>& a)
{
    std::cout << "A<int, const char*>" << std::endl;
}

int main()
{
    A a{ 10, "Hello, deduction" };

    show(a);
}
```

▶ 実行結果

```
A<int, const char*>
```

この章の理解度チェック P.476

1. 以下のようなプログラムになります。

▶ リスト App9.17　関数テンプレートとして定義されるクラステンプレートのメンバー関数

```
#include <iostream>

template <typename T>
class A
{
public:
    template <typename U>
    void show(U u);
};

template <typename T>
template <typename U>
void A<T>::show(U u)
{
    std::cout << u << std::endl;
}

int main()
{
    A<int> a;

    a.show(42);
}
```

▶ 実行結果

```
42
```

2. 以下のようなプログラムになります。

▶ リスト App9.18　部分特殊化したクラステンプレート

```
#include <iostream>

template <typename T, typename U, typename V>
class A
{
public:
    A() { std::cout << "プライマリーテンプレート" << std::endl; }
};

template <typename T, typename U>
class A<T, U, int>
{
public:
    A() { std::cout << "部分特殊化 A<T, U, int>" << std::endl; }
};

int main()
{
    A<void, int, float> vif;

    A<char, int*, int> cipi;
}
```

▶ 実行結果

```
プライマリーテンプレート
部分特殊化 A<T, U, int>
```

3. 以下のようなプログラムになります。

▶ リスト App9.19　可変引数テンプレートな関数テンプレート

```
#include <utility>

template <typename... T>
void eat(const T&...) { }

template <typename... T>
void fwd(T&&... values)
{
    eat(std::forward<T>(values)...);
}

int main()
{
    fwd(0, 1, 2, 3);
}
```

◆ 解説

可変引数テンプレートの関数パラメーターパックで参照やフォワーディング参照を受け取るには `T&...` や `T&&...` と記述します。また、それら関数パラメーターパックを完全転送するには `std::forward()` 関数を使いますが、パラメーターパック拡張を使い、要素ごとに `std::forward()` 関数で完全転送されるようにします。

4. 以下のようなプログラムになります。

▶ リスト App9.20　引数が整数型でないときにコンパイルエラーになる関数テンプレート

```
#include <type_traits>

template <typename T>
void foo(T v)
{
    static_assert(std::is_integer<T>::value);
}
```

5. クラステンプレートのテンプレートパラメーターの型推論は、コンストラクターに渡された実引数の型にもとづいて、クラステンプレートのテンプレートパラメーターを型推論します。

第 10 章の解答

練習問題　10.1　P.483

1. 以下のようなプログラムになります。

▶ リスト App10.1　例外によるプログラムの異常終了

```
#include <iostream>

int main()
{
    throw 0;

    std::cout << "throw のあと" << std::endl;
}
```

▶ 実行結果

```
terminating with uncaught exception of type int

Aborted
```

2. 以下のようなプログラムになります。

▶ リスト App10.2　try ブロックの追加

```
#include <iostream>

int main()
{
    try
    {
        throw 0;
    }
    catch (int e)
    {
        std::cout << "catch: " << e << std::endl;
    }
```

```
    std::cout << "try ブロックのあと" << std::endl;
}
```

▶ 実行結果

```
catch: 0
try ブロックのあと
```

練習問題　10.2　P.489

1. すべての例外を捕まえる catch ブロックについて説明してください。

A. `catch(...)` を使うことですべての例外をキャッチすることができます。しかし、投げられた例外オブジェクトについて具体的に知ることはできません。

2. `foo()` 関数には例外を投げないことを表す `noexcept` 指定がされているので、この関数が例外を投げると `std::terminate()` 関数が呼ばれて、プログラムは異常終了します。

練習問題　10.3　P.493

1. メモリ領域を確保できなかった場合に `std::bad_alloc` 例外を投げます。配列の `new` 演算子では、動的確保した配列の長さが初期化リストの長さより短い場合に `std::bad_array_new_length` 例外を投げます。

2. 自動的に解放されます。そのため、例外ハンドラーで特別な処理をせずともメモリリークしないようになっています。

練習問題　10.4　P.496

1. 以下のようなプログラムになります。

▶ リスト App10.3　std::exception から派生した独自の例外クラス

```
#include <exception>

class MyException : public std::exception
{
public:
    const char* what() const noexcept override;
};

const char* MyException::what() const noexcept
{
    return "MyException";
}
```

この章の理解度チェック　P.496

1. 以下のようなプログラムになります。

▶ リスト App10.4　例外を投げ、それを捕まえる

```
#include <iostream>

int main()
{
    std::cout << "try ブロックの前" << std::endl;

    try
    {
        std::cout << "throw の前" << std::endl;

        throw 0;

        std::cout << "throw のあと" << std::endl;
    }
```

```
    catch (int e)
    {
        std::cout << "catch ブロック" << std::endl;
    }

    std::cout << "終了" << std::endl;
}
```

▶ 実行結果

```
try ブロックの前
throw の前
catch ブロック
終了
```

2. 以下のようなプログラムになります。

▶ リスト App10.5　特定条件下で例外を投げるクラス

```
#include <iostream>

class A
{
    int i = -1;

public:
    A(int i) : i(i)
    {
        std::cout << "コンストラクター: " << i << std::endl;
        if (i < 0)
        {
            throw 0;
        }
    }

    ~A()
    {
        std::cout << "デストラクター: " << i << std::endl;
    }
```

```
};

int main()
{
    try
    {
        A* array = new A[]
        {
            A{0},
            A{1},
            A{2},
            A{-1},
            A{10},
            A{5},
        };
    }
    catch (int e)
    {
    }
}
```

▶ 実行結果

```
コンストラクター: 0
コンストラクター: 1
コンストラクター: 2
コンストラクター: -1
デストラクター: 2
デストラクター: 1
デストラクター: 0
```

◆ 解説

配列の動的確保をしている途中で例外が投げられると、それまでに初期化されたすべてのインスタンスのデストラクターが呼ばれます。

3. what() メンバー関数があり、例外の理由などさまざまなメッセージを返します。

第 11 章の解答

練習問題　11.1　P.503

1. 以下のようなプログラムになります。

▶ リスト App11.1　定義がまったく同じ 2 つのラムダ式

```
#include <iostream>
#include <typeinfo>

int main()
{
    auto lambda1 = [](){};
    auto lambda2 = [](){};

    if (typeid(lambda1) != typeid(lambda2))
    {
        std::cout << "ラムダ式は異なる型を持っています" << std::endl;
    }
}
```

▶ 実行結果

```
ラムダ式は異なる型を持っています
```

◆ 解説

typeid 演算子を使うことで実行時型情報を取得できます。

2. 以下のようなプログラムになります。

▶ リスト App11.2　継承関係がある 2 つのクラス

```
#include <typeinfo>
#include <iostream>

class Base
{
public:
    virtual ~Base() { }
};

class Derived : public Base
{
};

void check_rtti(const Base& base)
{
    if (typeid(base) == typeid(Derived))
    {
        std::cout << "base は Derived 型です" << std::endl;
    }
    else
    {
        std::cout << "base は Derived 型ではありません" << std::endl;
    }
}

int main()
{
    Base base;
    check_rtti(base);

    Derived derived;
    check_rtti(derived);
}
```

▶ 実行結果

```
base は Derived 型ではありません
base は Derived 型です
```

練習問題　11.3　P.512

1. 以下のようなプログラムになります。

▶ リスト App11.3　static_cast を使ったダウンキャスト

```
class Base
{
};

class Derived : public Base
{
};

int main()
{
    Derived derived;

    Base& rbase = derived;

    auto& rderived = static_cast<Derived&>(rbase);
}
```

2. 以下のようなプログラムになります。

▶ リスト App11.4　dynamic_cast を使ったダウンキャスト

```
#include <typeinfo>
#include <iostream>

class Base
{
public:
    virtual ~Base() { }
};

class Derived : public Base
{
};

class OtherDerived : public Base
```

```
{
};

int main()
{
    Derived derived;

    Base& rbase = derived;

    try
    {
        auto& oderived = dynamic_cast<OtherDerived&>(rbase);
    }
    catch (std::bad_cast& e)
    {
        std::cout << "ダウンキャスト失敗" << std::endl;
    }
}
```

▶ 実行結果

```
ダウンキャスト失敗
```

◆ 解説

`dynamic_cast` を使って参照型をキャストする場合に、キャストに失敗すると `std::bad_cast` 例外が送出されます。

この章の理解度チェック P.512

1. RunTime Type Information の略であり、日本語では実行時型情報と言います。これはポリモーフィックなオブジェクトの実際の型について、コンパイル時ではなく実行時にその情報を取得する機能です。

2. 以下のようなプログラムになります。

▶ リスト App11.5　dynamic_cast を使ったダウンキャスト

```
#include <iostream>

class Base
{
public:
    virtual ~Base() { }
};

class Derived : public Base
{
};

int main()
{
    Derived derived;

    Base& rbase = derived;

    // ダウンキャスト
    auto pderived = dynamic_cast<Derived*>(&rbase);
}
```

第 12 章の解答

練習問題　12.1　P.519

1. 以下の 5 種類です。

- 入力イテレーター
- 出力イテレーター
- 順方向イテレーター
- 双方向イテレーター
- ランダムアクセスイテレーター

練習問題　12.2　P.536

1. std::vector はランダムアクセスできるコンテナなので、そのイテレーターの種類はランダムアクセスイテレーターとなります。

2. 以下のようなプログラムになります。

▶ リスト App12.1　要素を 1 から 5 まで順番に並べる std::vector の初期化

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

int main()
{
    std::vector iv = {1, 2, 3, 4, 5};

    for (int e : iv)
    {
        std::cout << e << std::endl;
    }
}
```

▶ 実行結果

```
1
2
3
4
5
```

3. 以下のようなプログラムになります。

▶ リスト App12.2　std::vector の末尾に値を追加

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

int main()
{
    std::vector iv = {1, 2, 3, 4, 5};

    iv.push_back(42);

    for (int e : iv)
    {
        std::cout << e << std::endl;
    }
}
```

▶ 実行結果

```
1
2
3
4
5
42
```

練習問題　12.3　P.545

1. std::list のイテレーターは、前後の要素だけがわかっている双方向イテレーターです。

2. std::list のイテレーターは前後に 1 つずつしか進めないので、3 回インクリメントするか、std::advance() 関数もしくは std::next() 関数を使います。

3. 以下のようなプログラムになります。

▶ リスト App12.3　std::list のソート

```
#include <iostream>
#include <list>

int main()
{
    std::list il = {2, 4, 0, -1, 3};

    il.sort();

    for (int e : il)
    {
        std::cout << e << std::endl;
    }
}
```

▶ 実行結果

```
-1
0
2
3
4
```

練習問題　12.4　P.549

1. std::pair クラスを使うと 2 つ組が作れます。

2. 以下のようなプログラムになります。

▶ リスト App12.4　std::tuple を使った 3 つ組

```
#include <tuple>
#include <iostream>

int main()
{
    std::tuple t{3.14159265, "tuple", 42};

    std::cout << std::get<0>(t) << std::endl;
    std::cout << std::get<1>(t) << std::endl;
    std::cout << std::get<2>(t) << std::endl;

    std::cout << std::endl;

    std::get<0>(t) = 2.718281828;

    std::cout << std::get<0>(t) << std::endl;
    std::cout << std::get<1>(t) << std::endl;
    std::cout << std::get<2>(t) << std::endl;
}
```

▶ 実行結果

```
3.14159
tuple
42

2.71828
tuple
42
```

3. 構造化束縛とは、タプルの要素を個別の変数に展開する機能のことです。

練習問題　12.5　P.558

1. std::set は要素の重複を、std::map はキーの重複を許さないコンテナです。std::map では、値の重複は問題ありません。

2. 以下のようなプログラムになります。

▶ リスト App12.5　std::set

```
#include <iostream>
#include <set>

int main()
{
    std::set s = {0, 0, 0, 0, 1, 1, 2, 2};

    for (auto e : s)
    {
        std::cout << e << ' ';
    }
    std::cout << std::endl;

    s.insert(3);

    for (auto e : s)
    {
        std::cout << e << ' ';
    }
    std::cout << std::endl;

    s.insert(3);

    for (auto e : s)
    {
        std::cout << e << ' ';
    }
    std::cout << std::endl;
}
```

▶ 実行結果

```
0 1 2
0 1 2 3
0 1 2 3
```

練習問題　12.6　P.568

1. find アルゴリズムを使うことで、先頭から検査して最初に見つかった要素へのイテレーターが得られます。

2. 以下のようなプログラムになります。

▶ リスト App12.6　std::vector の降順ソート

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

int main()
{
    std::vector iv = { 3, 7, -1, 2, 0 };

    std::sort(iv.begin(), iv.end(), [](int l, int r) { return l > r; });

    for (auto e : iv)
    {
        std::cout << e << std::endl;
    }
}
```

▶ 実行結果

```
7
3
2
```

```
0
-1
```

3. 以下のようなプログラムになります。

▶ リスト App12.7　他のコンテナの先頭位置に正順になるようにコピー

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

int main()
{
    std::vector iv0 = { 3, 7, -1, 2, 0 };
    std::vector iv1 = { 0, 1, 2, 3 };

    std::copy(iv0.begin(), iv0.end(), std::inserter(iv1, iv1.begin()));

    for (auto e : iv1)
    {
        std::cout << e << std::endl;
    }
}
```

▶ 実行結果

```
3
7
-1
2
0
0
1
2
3
```

◆ 解説

std::inserter() 関数はコンテナの指定した位置に要素を挿入し、要素は挿入された順番通りに並びます。一方 std::front_inserter() 関数は常に先頭に要素を挿入します。この場合要素は逆順に並びます。

この章の理解度チェック P.568

1. 前のイテレーターを取得できるイテレーターは双方向イテレーターとランダムアクセスイテレーターなので、それ以外が答えとなります。

- 入力イテレーター
- 出力イテレーター
- 順方向イテレーター

2. 以下のようなプログラムになります。

▶ リスト App12.8　std::vector の、先頭と末尾以外の要素の削除

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

int main()
{
    std::vector iv = { 2, 4, 6, 8, 10, 12 };

    iv.erase(iv.begin() + 3);

    for (int e : iv)
    {
        std::cout << e << std::endl;
    }
}
```

▶ 実行結果

```
2
4
6
10
12
```

3. 以下のようなプログラムになります。

▶ リスト App12.9　構造化束縛を使ったタプルの展開

```
#include <tuple>
#include <iostream>

int main()
{
    std::tuple t{42, "structured binding", nullptr};

    auto [i, s, p] = t;

    std::cout << i << std::endl;
    std::cout << s << std::endl;
    std::cout << p << std::endl;
}
```

▶ 実行結果

```
42
structured binding
nullptr
```

4. `std::set` では要素の重複が許されていない一方、`std::multiset` では要素の重複が許されています。

5. 以下のようなプログラムになります。

▶ リスト App12.10　先頭の要素と同じ値を持つ要素の個数を数える

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

int main()
{
    std::vector iv = {0, 1, 0, 2, 0, 3};

    auto count = std::count(iv.begin() + 1, iv.end(), iv[0]);

    std::cout << "コンテナの先頭の要素と同じ要素は"
        << count << "個あります" << std::endl;
}
```

▶ 実行結果

```
コンテナの先頭の要素と同じ要素は 2 個あります
```

第 13 章の解答

練習問題　13.3　P.587

1. 以下のようなプログラムになります。

▶ リスト App13.1　std::string からの部分文字列の切り出し

```
#include <iostream>
#include <string>

int main()
{
    std::string line = "Hello, substring";

    auto substr = line.substr(7, 3);

    std::cout << substr << std::endl;
}
```

▶ 実行結果

```
sub
```

2. 以下のようなプログラムになります。

▶ リスト App13.2　data() メンバー関数

```
#include <string>
#include <iostream>

int main()
{
    std::string line = "Hello, C-interface";

    for (auto s = line.data(); *s; ++s)
    {
        std::cout.put(*s);
    }
```

```
    std::cout << std::endl;
}
```

▶ 実行結果

```
Hello, C-interface
```

◆ 解説

ヌル文字は数値としては 0 です。そのことを使い、ヌル文字が現れるまでループを回しています。

3. 以下のようなプログラムになります。

▶ リスト App13.3　std::string の比較

```
#include <string>
#include <iostream>
#include <iomanip>

int main()
{
    std::string a = "foo";

    std::string b = "Bar";

    std::cout << std::boolalpha << (a < b) << std::endl;
}
```

▶ 実行結果

```
false
```

◆ 解説

比較するときに std::string は辞書順となるように比較しますが、ASCII コード上ではアルファベットの大文字のほうが順番の上では先にくるように並んでいます。

練習問題　13.4　P.593

1. 以下のようなプログラムになります。

▶ リスト App13.4　整数を文字列に変換

```
#include <iostream>
#include <string>

int main()
{
    std::string s = std::to_string(1024);

    std::cout << s << std::endl;
}
```

▶ 実行結果

```
1024
```

2. 以下のようなプログラムになります。

▶ リスト App13.5　"3.14159265"を浮動小数点数に変換

```
#include <iostream>
#include <iomanip>
#include <string>

int main()
{
    std::string s = "3.14159265";
```

```
    double pi = std::stod(s);

    std::cout << std::setprecision(9) << pi << std::endl;
}
```

▶ 実行結果

```
3.14159265
```

練習問題　13.5　P.605

1. 以下のようなプログラムになります。

▶ リスト App13.6　入力された文字列と正規表現による完全マッチ

```
#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::string line, pattern;

    std::cout << "文字列> ";
    std::getline(std::cin, line);

    std::cout << "正規表現> ";
    std::getline(std::cin, pattern);

    std::regex re{pattern};

    if (std::regex_match(line, re))
    {
        std::cout << "正規表現に完全マッチしました" << std::endl;
    }
    else
    {
```

```
        std::cout << "正規表現に完全マッチしませんでした" << std::endl;
    }
}
```

▶ 実行結果

```
文字列> hello, regex
正規表現> \w+,\s*regex
正規表現に完全マッチしました
```

◆ 解説

コンソールからの入力は文字列を使わないのでエスケープは不要です。

2. 以下のようなプログラムになります。

▶ リスト App13.7　生文字列リテラル

```
#include <iostream>

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

▶ 実行結果

```
Hello,
 raw string
```

3. 以下のようなプログラムになります。

▶ リスト App13.8　入力された文字列と正規表現による部分マッチ

```
#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::string line, pattern;

    std::cout << "文字列> ";
    std::getline(std::cin, line);

    std::cout << "正規表現> ";
    std::getline(std::cin, pattern);

    std::regex re{pattern};
    std::smatch match;

    if (std::regex_search(line, match, re))
    {
        for (std::size_t i = 0; i < match.size(); ++i)
        {
            std::cout << match.str(i) << std::endl;
        }
    }
    else
    {
        std::cout << "正規表現にマッチしませんでした" << std::endl;
    }
}
```

▶ 実行結果

```
文字列> hello, regex
正規表現> (\w+),\s*(re)gex
hello, regex
hello
re
```

この章の理解度チェック P.605

1. 以下のようなプログラムになります。

▶ リスト App13.9　長い文字列

```
#include <string>
#include <iostream>

int main()
{
    std::string first = "Hello";
    std::string second = ", string";
    std::string third = " concatenation";

    auto line = first + second + third;

    std::cout << line << std::endl;
}
```

▶ 実行結果

```
Hello, string concatenation
```

2. 以下のようなプログラムになります。

▶ リスト App13.10　整数を文字列に変換し、再度整数に変換

```
#include <string>
#include <iostream>
#include <iomanip>

int main()
{
    const int value = 2048;

    auto s = std::to_string(value);
```

```
    auto i = std::stoi(s);

    std::cout << std::boolalpha << (i == value) << std::endl;
}
```

▶ 実行結果

```
true
```

3. 生文字列リテラルはエスケープが不要な文字列リテラルであり、改行等も含めてソースコードに書いたことがそのまま文字列となるリテラルです。

4. 以下のようなプログラムになります。

▶ リスト App13.11　正規表現を使った文字列の置換

```
#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::string line = "Hello, regex";

    std::regex re{"r\\w+"};

    auto s = std::regex_replace(line, re, "replacement");

    std::cout << s << std::endl;
}
```

▶ 実行結果

```
Hello, replacement
```