第9講 関数（発展・応用編）

本講では、C++の関数に関するより高度な特性、特に参照、`const`修飾子、スコープ、記憶期間、関数ポインタ、関数のオーバーロード、インライン関数などについて深く掘り下げ、C言語における実装方法との比較も行います。

講義概要

1. ポインタまたは参照による関数外部の変数の変更
    - 値渡しの限界
    - C++の参照渡し
    - C言語のポインタ渡し
    - 比較とまとめ
2. **`const`**修飾子と引数
    - `const`定数
    - 引数の保護：`const`の応用
    - `const`とポインタ
    - `const`と参照 (C++)
    - 引数の入力/出力スタイル
3. スコープと記憶期間
    - スコープ (Scope)：ファイルスコープ vs. ブロックスコープ
    - スコープの利用ルール
    - スコープ解決演算子 `::`
    - 記憶期間 (Storage Duration)：自動記憶期間 vs. 静的記憶期間
    - `static`キーワード
4. 関数ポインタ
    - 宣言と使用
    - 関数引数としての利用（コールバック関数）
5. 関数のオーバーロード **(C++)**
    - 関数オーバーロードとは？
    - 関数シグネチャ (Signature)
6. インライン関数
    - 関数呼び出しのコスト
    - `inline`キーワードの役割
7. 関数マクロとその問題点
    - `#define`による関数マクロの作成
    - 関数マクロの副作用
8. 関数テンプレート
    - 関数テンプレートの必要性
    - 関数テンプレートの基本
    - 明示的な具現化

9. 大規模なプログラムの開発
    - 分割コンパイルと結合
    - 結合 (Linkage): 外部結合と内部結合
    - ヘッダファイル
10. 名前空間
    - 名前空間の定義
    - 名前空間のメンバ
    - `using`宣言と`using`指令

---

1. ポインタまたは参照による関数外部の変数の変更

デフォルトでは、関数の引数は値渡し **(pass by value)** であり、これは関数が実引数のコピーを受け取ることを意味し、元の変数を変更することはできません。関数内部で外部の変数を変更するために、CとC++は異なる仕組みを提供しています。

値渡しの限界：なぜswapは失敗するのか？

`List 6-16`に示す、2つの変数の値を交換しようとする関数を見てみましょう。

```C/C++
// List 6-16: 2つの整数の値を交換する（誤った例）
#include <iostream>

using namespace std;

//--- xとyの値を交換する（値渡し）---
void swap(int x, int y)
{
    int t = x;
    x = y;
    y = t;
}

int main()
{
    int a, b;
    cout << "変数a: "; cin >> a;
    cout << "変数b: "; cin >> b;

    swap(a, b);

    cout << "変数aの値は " << a << " です。\n";
    cout << "変数bの値は " << b << " です。\n";
```

```
}
```

プログラムの実行結果は期待通りにならないかもしれません。aとbの値が交換されないのです。これは値渡しの特性によるものです。swap関数が受け取るのはaとbのコピー（xとy）であり、関数内部で交換されるのはこれらコピーの値だけで、main関数内の元の変数aとbには何の影響もありません。

参照：変数の別名

上記の問題を解決するためには、参照 **(Reference)** を使用する必要があります。参照とは、既存の変数に対する別名 **(Alias)** です。参照が一度ある変数に初期化されると、その変数に生涯結び付けられ、参照に対するすべての操作は元の変数に対する操作と等価になります。

参照 **(Reference)** は**C++**の機能であり、**C**言語にはありません。

List 6-17は、参照の基本的な使い方を示しています。

C/C++
```cpp
// List 6-17: 参照オブジェクト
#include <iostream>

using namespace std;

int main()
{
    int x = 1;
    int y = 2;

    // aはxの参照（別名）であり、初期化後に生涯結び付けられる
    int& a = x;

    cout << "a = " << a << '\n';
    cout << "x = " << x << '\n';
    cout << "y = " << y << '\n';

    a = 5; // aに5を代入することは、xに5を代入することと等価

    cout << "a = " << a << '\n';
    cout << "x = " << x << '\n';
    cout << "y = " << y << '\n';
}
```

`int& a = x;`という宣言において、`&`はアドレス演算子ではなく、`a`が`int`型の参照であることを示します。`a`は`x`の別名となり、両者は同じメモリ位置を指します。したがって、`a`を変更することは`x`を変更することになります。

注意：参照は宣言時に必ず初期化しなければなりません。

C++の参照渡し

引数を参照型として宣言することで、関数は元の変数を直接操作できます。

```C/C++
// C++スタイル：参照を用いて2つの整数の値を交換する
#include <iostream>
using namespace std;

void swap_cpp(int& x, int& y) // xとyは参照
{
    int t = x;
    x = y;
    y = t;
}

int main()
{
    int a = 5, b = 10;
    cout << "交換前: a = " << a << ", b = " << b << '\n';
    swap_cpp(a, b); // 変数を直接渡す
    cout << "交換後: a = " << a << ", b = " << b << '\n';
}
```

C言語のポインタ渡し

C言語で同様の機能を実現するには、ポインタ **(Pointers)** を使用しなければなりません。ポインタとは、対応する変数のメモリアドレスを格納する変数です。変数のアドレスを関数に渡すことで、関数内でそのポインタを逆参照 **(Dereference)**（*演算子を使用）して、元の変数に直接アクセスし、変更することができます。

```C/C++
// Cスタイル：ポインタを用いて2つの整数の値を交換する
#include <stdio.h>

void swap_c(int* x, int* y) // xとyはポインタ
{
    int t = *x; // xを逆参照し、それが指す値を取得する
```

```
    *x = *y;
    *y = t;
}

int main()
{
    int a = 5, b = 10;
    printf("交換前: a = %d, b = %d\n", a, b);

    swap_c(&a, &b); // aとbのアドレスを渡す

    printf("交換後: a = %d, b = %d\n", a, b);
    return 0;
}
```

比較とまとめ

特性	値渡し (C/C++)	ポインタ渡し (C/C++)	参照渡し (C++)
定義	`func(int x)`	`func(int* x)`	`func(int& x)`
呼び出し	`func(a)`	`func(&a)`	`func(a)`
関数内部	コピーを操作	ポインタを操作、`*x`で逆参照が必要	元の変数そのものとして操作
元の値を変更	不可	可能	可能
null許容	非該当	`nullptr`を渡せる	nullにはなれない
安全性	安全	安全でない（nullポインタ、不正なポインタのリスク）	比較的安全（一度参照が確定すると変更不可）

結論:

- C++では、渡された引数を変更する必要がある場合、ポインタよりも安全で構文が簡潔なため、参照渡しを優先すべきです。
- ポインタ渡しは、「オプション」の引数を示す必要がある場合（`nullptr`を渡すことによる）や、C言語のコードと連携する際に依然として必要です。

---

2. const修飾子と引数

constキーワードは、変更不可能な定数を作成するために使用されます。関数引数と組み合わせて使用すると、コードの堅牢性と明確性を大幅に向上させることができます。

const定数

関数引数の話に入る前に、まずconstの基本的な使い方、つまり定数を定義することを理解する必要があります。一度初期化されると、その値は変更できません。

```C/C++
const double PI = 3.1415926535;
// PI = 3.14; // エラー！定数の値は変更できない
```

引数の保護：constの応用

constは、関数が渡された引数を意図せず変更しないことを保証します。これは、チームでの共同作業や大規模プロジェクトの作成において特に重要です。

constとポインタ

定数へのポインタ (**Pointer to const**):

```C/C++
const int* p;
```

これは、ポインタpを介してそれが指す値を変更できないことを意味しますが、ポインタp自体は他のアドレスを指すことができます。

```C/C++
int a = 10, b = 20;
const int* p = &a;
// *p = 15; // エラー！pを介してaの値を変更することはできない
p = &b;    // 正しい！pは他の場所を指すことができる
```

constと参照 (C++)

C++では、constと参照の組み合わせは、効率的で安全な引数渡しを実現するための鍵となります。

const参照:

```
C/C++
void func(const std::string& msg)
{
    std::cout << "受け取ったメッセージ: " << msg << std::endl;
    // msg[0] = 'X';  // ← これは const のためコンパイルエラーになる
}
```

ここでの const は、関数の中でパラメータ msg が参照している文字列の内容を変更できないことを意味します（例えば msg[0] = 'X' のように内容を書き換える操作は不可）。関数宣言で const を付けることで、この引数は「呼び出し前後で値が変わらない入力専用パラメータ」であり、出力用ではないことを示しています。

これは定数参照渡しと呼ばれる非常に一般的な関数引数のスタイルです。これには2つの主な利点があります:

1. 効率性: 値渡しのようにオブジェクト全体をコピーするのを避けるため（特にstringやvectorなどの大きなオブジェクトの場合）、パフォーマンスが向上します。
2. 安全性: 関数が渡された元のオブジェクトsを変更しないことを保証するため、呼び出し元は安心して引数を渡すことができます。

引数の入力/出力スタイル

constとポインタ/参照の組み合わせにより、引数の意図（入力・出力・入出力パラメータ）を明確に示すためのコーディングスタイルを確立できます。

意図	**C++** スタイル	**C** スタイル
「入力専用パラメータ」	`int` (値), `const string&` (定数参照)	`int` (値), `const char*` (ポインタ)
「出力パラメータ」あるいは「入出力パラメータ」	`int&` (参照), `string&` (参照)	`int*` (ポインタ) , `char*` (ポインタ)

---

3. スコープと記憶期間

C++では、各識別子（変数名、関数名など）にはそのスコープ **(Scope)** と記憶期間 **(Storage Duration)** があります。

スコープ (Scope)

スコープとは、識別子がプログラム内で有効な範囲のことです。C++には主に2種類のスコープがあります：

- ファイルスコープ **(File Scope)**: すべての関数の外で宣言された識別子はファイルスコープを持ちます。これらは宣言された時点からファイルの終わりまで有効で、一般にグローバル変数 **(Global Variables)** と呼ばれます。
- ブロックスコープ **(Block Scope)**: コードブロック`{}`内で宣言された識別子はブロックスコープを持ちます。これらは宣言された時点からそのコードブロックの閉じ括弧`}`で無効になり、一般にローカル変数 **(Local Variables)** と呼ばれます。

`List 6-20`は、異なるスコープを示しています。

```C/C++
// List 6-20: 識別子のスコープを確認する
#include <iostream>

using namespace std;

int x = 75; // ファイルスコープ

void print_x()
{
    cout << "print_x: x = " << x << '\n';
}

int main()
{
    cout << "x = " << x << '\n'; // ファイルスコープのxにアクセス

    int x = 999; // ブロックスコープ、ファイルスコープのxを隠蔽する
    cout << "x = " << x << '\n'; // ブロックスコープのxにアクセス

    for (int i = 1; i <= 5; i++)
    {
        int x = i * 11; // さらに内側のブロックスコープ
        cout << "x = " << x << '\n';
    }
```

```
    cout << "x = " << x << '\n';      // ブロックスコープ
    cout << "::x = " << ::x << '\n'; // スコープ解決演算子を使用してファイルスコープのx
にアクセス
    print_x();
}
```

内側のスコープに外側のスコープと同じ名前の識別子が存在する場合、外側のスコープの識別子は隠蔽（Hiding）されます。上記の例では、`main`関数内の`x`がグローバルの`x`を隠蔽しています。

スコープの利用ルール

ベストプラクティス：ネストしたスコープで同じ変数名を使用することは避けるべきです。C++の構文では変数の隠蔽が許されていますが、これはコードの可読性を低下させ、発見しにくいバグの原因となる可能性があります。明確な命名とスコープの明確な区分けが、高品質なコードを書くための鍵です。

スコープ解決演算子 `::`

ローカル変数によって隠蔽されたグローバル変数にアクセスする必要がある場合は、スコープ解決演算子 `::` を使用できます。識別子の前に`::`を付けることで、ファイルスコープ（グローバル）のバージョンを明示的に指定します（例：`::x`）。

記憶期間 (Storage Duration)

記憶期間は、オブジェクト（変数）がメモリ内に存在する時間を記述します。主に2種類あります：

- 自動記憶期間 **(Automatic Storage Duration)**: ブロックスコープで宣言された変数（`static`でない）は、通常、自動記憶期間を持ちます。プログラムの実行がその宣言に達したときに作成され、そのコードブロックを抜けるときに破棄されます。
- 静的記憶期間 **(Static Storage Duration)**: ファイルスコープで宣言された変数、または`static`キーワードで宣言された変数は、静的記憶期間を持ちます。これらはプログラムの実行開始前に作成され、プログラムの終了時にのみ破棄されます。明示的に初期化されない場合、自動的に0で初期化されます。

List 6-22 のプログラムで、記憶域期間に関する理解を深めましょう。

```C/C++
// List 6-22
#include <iostream>
```

```cpp
using namespace std;

int fx = 0;             // 静的記憶域期間＋ファイル有効範囲

void func()
{
    static int sx = 0; // 静的記憶域期間＋ブロック有効範囲
    int ax = 0;        // 自動記憶域期間＋ブロック有効範囲

    fx++;
    sx++;
    ax++;
    cout << fx << " " << sx << " " << ax << '\n';
}

int main()
{
    cout << "fx sx ax\n";
    cout << "--------\n";
    for (int i = 0; i < 8; i++)
        func();
}
```

---

4. 関数ポインタ

CとC++では、関数もメモリ内に格納されているため、アドレスを持ちます。関数ポインタ **(Pointer to Function)** とは、関数のアドレスを格納するポインタ変数です。

宣言と使用

関数ポインタの宣言構文では、関数の戻り値の型と引数の型リストを指定する必要があります。

```cpp
C/C++
// 「intを返し、2つのintを受け取る」関数を指すポインタを宣言
int (*func_ptr)(int, int);
```

ここでの括弧 `(*func_ptr)`は必須です。これがないと、`int *func_ptr(int, int);` は「int*を返す関数の宣言」として解釈されてしまいます。

```C/C++
#include <iostream>

// 2つの整数を足し算する関数
int add(int a, int b) { return a + b; }

// 2つの整数を引き算する関数
int subtract(int a, int b) { return a - b; }

int main()
{
    // 関数ポインタの宣言（int型を2つ受け取り、int型を返す関数へのポインタ）
    int (*op)(int, int);

    // ポインタを add 関数に設定
    op = add;
    // ポインタ経由で関数を呼び出す
    std::cout << "Addition: " << op(5, 3) << std::endl;

    // ポインタを subtract 関数に設定
    op = subtract;
    // 再びポインタ経由で関数を呼び出す
    std::cout << "Subtraction: " << op(5, 3) << std::endl;
}
```

関数引数としての利用（コールバック関数）

関数ポインタの最も強力な用途の1つは、他の関数への引数として渡すことです。これにより、コールバック **(Callback)** 機構を実装できます。これは、C言語で汎用アルゴリズムを実装する一般的な方法です。

```C/C++
#include <iostream>

// compute関数は、実行する演算を決定するために関数ポインタopを受け取る
void compute(int a, int b, int (*op)(int, int))
{
    std::cout << "Result: " << op(a, b) << std::endl;
}

int add(int a, int b) { return a + b; }
int multiply(int a, int b) { return a * b; }

int main()
{
```

```
    compute(10, 5, add);        // add関数をコールバックとして渡す
    compute(10, 5, multiply);   // multiply関数をコールバックとして渡す
}
```

このパターンはC++でも利用可能ですが、関数オブジェクト（**Functors**）やラムダ式といった、より現代的で強力な機能に置き換えられることが多いです。

---

5. 関数のオーバーロード (C++)

C++では、引数リストが異なっていれば、複数の関数が同じ名前を持つことができます。この機能を関数オーバーロード **(Function Overloading)** と呼びます。これは**C++**の機能であり、**C**言語は関数オーバーロードをサポートしていません。

例えば、`calc_max`関数に複数のバージョンを提供できます：

```C/C++
// List 6-25: 関数のオーバーロードの例
#include <iostream>

using namespace std;

//--- int, int の最大値 ---
int calc_max(int a, int b)
{
    return a > b ? a : b;
}

//--- float, float の最大値 ---
float calc_max(float a, float b)
{
    return a > b ? a : b;
}

//--- int, int, int の最大値 ---
int calc_max(int a, int b, int c)
{
    int max_val = a;
    if (b > max_val) max_val = b;
    if (c > max_val) max_val = c;
    return max_val;
```

```
}

int main()
{
    cout << "max(10, 20) = " << calc_max(10, 20) << '\n';
    cout << "max(3.14f, 1.23f) = " << calc_max(3.14f, 1.23f) << '\n';
    cout << "max(10, 20, 30) = " << calc_max(10, 20, 30) << '\n';
}
```

コンパイラは、呼び出し時に提供された引数の型と数に基づいて、最適な関数バージョンを自動的に選択します。

関数シグネチャ（Signature）

コンパイラは、関数シグネチャ（Signature）によって同じ名前の関数を区別します。関数シグネチャは、引数の個数と引数の型（順番も含むによって構成されます。関数名自体や戻り値の型はシグネチャの一部ではありません。

したがって、戻り値の型だけを変更して関数をオーバーロードすることはできません。次の宣言は衝突し、コンパイルエラーとなります。

```C/C++
// エラー：戻り値の型だけではオーバーロードできない
int sum(int a, int b);
double sum(int a, int b); // コンパイルエラー！上と同じシグネチャ
```

---

6. インライン関数

関数呼び出しのコスト

関数はコードを整理するための強力なツールですが、関数呼び出しにはコストがかからないわけではありません。関数を呼び出すたびに、プログラムは一連の操作を実行する必要があります：

1. 引数と戻りアドレスをスタックにプッシュする。
2. 関数のアドレスにジャンプする。
3. 関数本体を実行する。
4. 戻り値（もしあれば）を特定の場所に格納する。
5. スタックから戻りアドレスと引数をポップする。

6. 呼び出し元にジャンプして戻る。

非常に短く、頻繁に呼び出される関数にとって、これらのオーバーヘッドは、関数本体自体の実行時間を超えるほど顕著になる可能性があります。

`inline`キーワードの役割

この問題を解決するために、C++はインライン関数 **(Inline Function)** の概念を提供します。関数定義の前に`inline`キーワードを付けることで、コンパイラに対して、その関数を通常の関数呼び出し方法で実行しないように「提案」することができます。

```
C/C++
//--- a, b の最大値（インライン関数版）---//
inline int calc_max(int a, int b) { return a > b ? a : b; }
```

コンパイラがこの提案を受け入れると、関数呼び出し命令を生成する代わりに、関数本体のコードを各呼び出し元に直接「埋め込み」または「展開」します。これにより、関数呼び出しのオーバーヘッドがなくなり、あたかも呼び出し元に直接コードを書いたかのように動作しますが、関数のカプセル化と可読性の利点は維持されます。

```
C/C++
// コンパイル時、x = calc_max(fbi, cia); は以下のように展開される可能性がある：
x = fbi > cia ? fbi : cia;
```

重要：

- `inline`はコンパイラへの提案にすぎず、強制的な命令ではありません。コンパイラは、関数本体が長すぎる、ループや再帰を含むなど、さまざまな理由でこの提案を無視し、通常の関数呼び出しを実行することがあります。
- `main`関数をインラインとして宣言することはできません。
- インライン関数は通常、コードの構造を犠牲にすることなくパフォーマンスを向上させるために、短くて頻繁に呼び出される関数に使用されます。

---

7. 関数マクロとその問題点

C++の初期およびC言語では、関数形式マクロ **(Function-like Macro)** が、インライン関数と同様の目的でパフォーマンス向上の手段として使用されていました。

#defineによる関数マクロの作成

関数マクロは、#defineプリプロセッサ命令によって作成されます。これは関数呼び出しのように見えますが、コンパイル前にプリプロセッサが単純なテキスト置換を行います。

マクロ置換はプリプロセス段階で行われるのに対し、関数呼び出しやインライン展開はコンパイル段階で行われるため、両者のメカニズムは全く異なります。

```C/C++
// List 6C-2: 整数と浮動小数点数の2乗を求める（関数マクロ）
#include <iostream>

using namespace std;

#define sqr(x) ((x)*(x)) // 関数マクロを定義

int main()
{
    int n;
    double x;

    cout << "整数を入力してください：";
    cin >> n;
    cout << "その数の2乗は " << sqr(n) << " です。\n";

    cout << "実数を入力してください：";
    cin >> x;
    cout << "その数の2乗は " << sqr(x) << " です。\n";
}
```

コンパイル時、sqr(n)は直接((n)*(n))に置換されます。

関数マクロの副作用

関数マクロは関数呼び出しのオーバーヘッドをなくすことができますが、副作用 **(Side Effects)** と呼ばれる深刻な問題が存在します。

1. 引数の複数回評価 マクロに渡される引数にインクリメント演算子"++"やデクリメント演算子"--"が含まれている場合、予期せぬ結果を引き起こす可能性があります。例えば：

```C/C++
int a = 3;
```

```
int result = sqr(a++); // ((a++)*(a++)) に展開される
```

プログラマはresultが9になり、aが最終的に4になることを期待するかもしれません。しかし実際にはa++が2回実行され、resultの値は3 * 4 = 12となり、aは最終的に5になる可能性があります。これは未定義動作であり、結果はコンパイラによって異なります。

2. 演算子の優先順位の問題 マクロの定義で各引数と式全体を括弧で慎重に囲まないと、演算子の優先順位の問題が容易に発生します。

```
C/C++
#define add(x, y) x + y
int r = add(a, b) * c; // a + b * c に展開され、(a + b) * c という意図とは異なる
```

正しいマクロ定義は次のようになります：

```
C/C++
#define add(x, y) ((x) + (y))
```

重要: 型安全でなく、デバッグが困難で、予測不能な副作用があるなど多くの問題があるため、C++では関数マクロの代わりに常にインライン関数を優先すべきです。

インライン関数は型チェックとスコープ制御を経ており、関数マクロのような副作用を導入しません。

---

8. 関数テンプレート

関数テンプレートの必要性

同じロジックを異なるデータ型に対して適用したい場合があります。例えば、配列内の最大値を求める関数を考えてみましょう。int型の配列用とdouble型の配列用に、それぞれ別の関数を用意するのは非効率です。

List 9-1は、この問題点を示しています。averageという名前の関数は、引数の型が違うだけで2つ定義されています。

```C/C++
// List9-1:平均値を求める（多重定義版）
#include <iostream>
using namespace std;

//--- 平均値を返却（int版）---//
int average(int a, int b) { return (a + b) / 2; }

//--- 平均値を返却（double版）---//
double average(double a, double b) { return (a + b) / 2; }

int main()
{
    // 整数の平均値
    cout << "平均値は" << average(3, 6) << "です。\n";

    // 実数の平均値
    cout << "平均値は" << average(3.0, 6.0) << "です。\n";
}
```

このようなコードの重複を避け、より汎用的で保守しやすいコードを書くために、C++では関数テンプレートという仕組みが提供されています。

関数テンプレートの基本

C++の機能である関数テンプレートを使用すると、具体的な型をテンプレート仮引数（`Type`など）で置き換えた、関数のひな形を定義できます。

```C/C++
template <class Type>
Type average(Type a, Type b)
{
    Type sum = a + b;
    return sum / 2;
}
```

- `template <class Type>`: これからテンプレートを定義することを示します。`Type`がテンプレート仮引数です。`class`の代わりに`typename`も使用できます。
- `Type`: `int`や`double`のような具体的な型の代わりに使用されます。

このテンプレートから、コンパイラは関数呼び出し時に渡された実引数の型に応じて、適切な関数を自動的に生成します。このプロセスを具現化 **(Instantiation)** と呼びます。

average(3, 6)という呼び出しがあればint版のaverage関数が、average(3.0, 6.0)という呼び出しがあればdouble版のaverage関数が、コンパイラによって舞台裏で生成されるのです。

```
None
[関数テンプレート average]

      ==> int に具現化 --> int average(int, int)

      ==> double に具現化 --> double average(double, double)
```

```
C/C++
// List9-2:平均値を求める（関数テンプレート版）
#include <iostream>
using namespace std;

template <class Type>
Type average(Type a, Type b)
{
    Type sum = a + b;
    return sum / 2;
}

int main()
{
    // 整数の平均値
    cout << "平均値は" << average(3, 6) << "です。\n";

    // 実数の平均値
    cout << "平均値は" << average(3.0, 6.0) << "です。\n";
}
```

明示的な具現化

通常、コンパイラは関数に渡された引数からテンプレートの型を自動的に推論します（テンプレート引数推論）。しかし、型推論がうまくいかない場合もあります。

例えば、

```
C/C++
cout << "平均値は" << average(3, 6.0) << "です。\n";
```

int型の変数aとdouble型の変数bを引数としてaverage(a, b)を呼び出すと、コンパイラはTypeをintとdoubleのどちらにすればよいか判断できず、エラーとなります。

```
Shell
temp.cpp:11:6: note: template argument deduction/substitution failed:
temp.cpp:25:34: note:   deduced conflicting types for parameter 'Type' ('int'
and 'double')
   25 |     cout << "平均値は" << average(3, 6.0) << "です。\n";
      |                           ~~~~~~~^~~~~~~~
```

このような場合、プログラマが明示的に型を指定する必要があります。

```
C/C++
// List 9C-1: 関数テンプレートの明示的な呼び出し
#include <iostream>
using namespace std;

template <class Type>
Type average(Type a, Type b)
{
    Type sum = a + b;
    return sum / 2;
}

int main()
{
    cout << "平均値は" << average(3, 6) << "です。\n";
    cout << "平均値は" << average(3.0, 6.0) << "です。\n";

    cout << "平均値(int)は" << average<int>(3, 6.0) << "です。\n";
    cout << "平均値(double)は" << average<double>(3, 6.0) << "です。\n";
}
```

maxof<int>(a, b)と記述することで、Typeをintとして具現化するようコンパイラに指示し、double型のbはint型に変換されてから関数が呼び出されます。

maxof<double>(a, b)と記述することで、Typeをdoubleとして具現化する。

---

9. 大規模なプログラムの開発

大規模なプログラムは、複数のソースファイルで構成されます。本節では、そのようなプログラムの実現に必要となる、結合やヘッダの作成などを学習します。

分割コンパイルと結合

これまで作成してきたプログラムは、単一のソースファイルで実現される小規模なものでした。しかし、多数の関数で構成される大規模なプログラムは、開発や管理を容易にするために、複数のソースファイルに分割して実現するのが一般的です。

List 9-5、List 9-6は、2つのソースファイルに分割して実現した例です。

C/C++
```
// List 9-5: math_util.cpp
// --- べき乗を求める関数 ---
double power(double x, int n)
{
    double tmp = 1.0;
    for (int i = 1; i <= n; i++) {
        tmp *= x;
    }
    return tmp;
}
```

C/C++
```
// List 9-6: main_09_6.cpp
#include <iostream>
using namespace std;

// --- べき乗を求める ---
double power(double x, int n); // 関数powerの宣言

int main()
{
    double a;
    int b;
    cout << "aのb乗を求めます。\n";
    cout << "a："; cin >> a;
    cout << "b："; cin >> b;
    cout << "aの" << b << "乗は" << power(a, b) << "です。\n";
}
```

複数のソースファイルから実行プログラムを作成するまでの手順の概略をFig.9-2に示します。この作業を一般に分割コンパイルと呼びます。分割コンパイルの具体的な手順は開発環境に依存しますが、おおまかにはコンパイル・リンクを行います。

Fig.9-2　分割コンパイルの手順

```Shell
# 1. それぞれのソースファイルをオブジェクトファイル（.o）にコンパイル
#    -c : コンパイルのみ
g++ -c main_09_6.cpp
g++ -c math_util.cpp

# 2. オブジェクトファイルをリンクして実行可能ファイルを作成
#    -o : 出力ファイル名を指定
g++ main_09_6.o math_util.o -o main_09_6

# 3. 実行ファイルを実行する
./main_09_6
```

```
PS C:\Users\sxwin\Documents\Lecture\cpp\temp> g++ main_09_6.o math_util.o
PS C:\Users\sxwin\Documents\Lecture\cpp\temp> g++ -c main_09_6.cpp
PS C:\Users\sxwin\Documents\Lecture\cpp\temp> g++ -c math_util.cpp
PS C:\Users\sxwin\Documents\Lecture\cpp\temp> g++ main_09_6.o math_util.o -o main_09_6
PS C:\Users\sxwin\Documents\Lecture\cpp\temp> .\main_09_6
aのb乗を求めます。
a：4
b：3
aの3乗は64です。
```

先ほどの4つのコマンドは、次のように1行にまとめて書くこともできます。

```Shell
# g++ で main_09_6.cpp と math_util.cpp をコンパイルして main_09_6 という実行ファイル
# を作成し、そのコンパイルが成功した場合にだけ .\main_09_6 を実行する
```

```
g++ main_09_6.cpp math_util.cpp -o main_09_6; if($?) { .\main_09_6 }
```

結合 (Linkage)

C++では、呼び出し元と呼び出される関数や、他のソースファイル中で定義された関数や変数との結合方法をコンパイラに教えるための結合宣言が必要です。この宣言を省略すると、コンパイルエラーになります。

外部結合 (External Linkage)

複数のソースファイル間で共有される変数や関数は、外部結合 **(External Linkage)** を持ちます。これは、他のソースファイルから参照できることを意味します。

ソースファイル間で関数を共有するためには、`extern`キーワードを使用して、他のソースファイルで定義されている関数や変数を宣言する必要があります。

内部結合 (Internal Linkage)

`static`キーワードをグローバル変数や関数に適用すると、その変数や関数は内部結合 **(Internal Linkage)** を持ちます。これは、その変数や関数が定義されているソースファイル内でのみアクセス可能であり、他のソースファイルからは参照できないことを意味します。

`List 9-7`の`game.cpp`では、`static int rotase = 0;`と`static int gen_no;`が内部結合を持つ変数として宣言されています。これらは`game.cpp`内でのみ使用され、他のファイルからはアクセスできません。

重要: 複数のソースファイルで同じ名前のグローバル変数や関数を定義し、それらがすべて外部結合を持つ場合、多重定義エラー **(ODR: One Definition Rule)** となり、リンク時にエラーが発生します。内部結合を使用することで、この問題を回避できます。

ヘッダファイル

大規模なプログラムでは、多数の関数やグローバル変数が存在し、それらの宣言を各ソースファイルに記述するのは非効率的です。この問題を解決するために、ヘッダファイル **(Header File)** を使用します。

ヘッダファイルには、関数やグローバル変数の宣言（プロトタイプ宣言）を記述し、各ソースファイルで`#include`ディレクティブを使ってインクルードします。これにより、宣言の重複を防ぎ、コードの保守性を高めることができます。

下記のコードは、ユーザーからの入力に基づいてゲームを行うプログラムの例です。例えば、kazuate.hというヘッダファイルを作成し、game.cpp、io.cpp、kazuate.cppで共有される関数の宣言を記述します。

```C/C++
// kazuate.h (例)
// 数当てゲーム（ヘッダ部）
void initialize();                  //【初期化】 乱数の種を現在の時刻に基づいて設定
void gen_no();                          //【問題の作成】 0～max_noの値を乱数で生成
int judge(int cand);                //【解答の判定】 candが正解かどうかを判定
int input();                        //【解答の入力】 0～max_noの値を入力させる
bool confirm_retry();               //【続行の確認】 再ゲームを行うかを確認

extern int max_no;                  // 当てるべき数の最大値
```

そして、各ソースファイルでこのヘッダファイルをインクルードします。

```C/C++
// List 9-11: game.cpp
// 数当てゲーム（ゲーム部）
#include <ctime>
#include <cstdlib>
#include "kazuate.h"

using namespace std;

static int kotae = 0;

//--- 初期化 ---//
void initialize()
{
	srand(time(NULL));
}

//--- 問題（当てるべき数）の作成 ---//
void gen_no()
{
	kotae = rand() % (max_no + 1);
}

//--- 解答の判定 ---//
int judge(int cand)
{
	if (cand == kotae)          // 正解
		return 0;
```

```
        else if (cand > kotae)     // 大きい
                return 1;
        else                                // 小さい
                return 2;
}
```

C/C++

```
// List 9-12: io.cpp
// 数当てゲーム（入出力部）
#include <iostream>
#include "kazuate.h"

using namespace std;

//--- 入力を促す ---//
static void prompt()
{
        cout << "0〜" << max_no << "の数：";
}

//--- 解答の入力 ---//
int input()
{
        int val;
        do {
                prompt();     // 入力を促す
                cin >> val;
        } while (val < 0 || val > max_no);
        return val;
}

//--- 続行の確認 ---//
bool confirm_retry()
{
        int cont;
        cout << "もう一度しますか？\n"
                << "<Yes…1／No…0>：";
        cin >> cont;
        return static_cast<bool>(cont);
}
```

C/C++

```
// List 9-13: kazuate.cpp
// 数当てゲーム（メイン部）
#include <iostream>
#include "kazuate.h"

using namespace std;

int max_no = 9;                    // 当てるべき数の最大値

int main()
{
	initialize();		// 初期化
	cout << "数当てゲーム開始！\n";

	do {
		gen_no();		// 問題（当てるべき数）の作成
		int hantei;
		do {
			hantei = judge(input());	// 解答の判定
			if (hantei == 1)
				cout << "\aもっと小さいですよ。\n";
			else if (hantei == 2)
				cout << "\aもっと大きいですよ。\n";
		} while (hantei != 0);
		cout << "正解です。\n";
	} while (confirm_retry());
}
```

Shell

```
# kazuate.cpp・game.cpp・io.cpp をコンパイルして実行ファイル kazuate を作成し、
# コンパイルが成功した場合だけ .\kazuate を実行する
g++ kazuate.cpp game.cpp io.cpp -o kazuate; if ($?) { .\kazuate }
```

C/C++

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

この2つの #include の違いは、「どこからヘッダファイルを探すか」です。

- #include <iostream>
  - 山かっこ < > を使うと、コンパイラは「標準ライブラリやシステムのインクルードディレ

クトリ」からヘッダを探します。
    - 例：<iostream>, <vector>, <string>, <stdlib.h> など標準ライブラリ用。
- #include "kazuate.h"
    - 二重引用符 " " を使うと、まず「現在のソースファイルと同じディレクトリ」から kazuate.h を探し、見つからなければ、その後でシステムのインクルードディレクトリも探します。
    - 自分が作ったヘッダファイル（プロジェクト内のヘッダ）に使うのが普通です。

重要: ヘッダファイルには、通常、宣言のみを記述し、定義は対応する`.cpp`ファイルに記述します。また、ヘッダファイルの多重インクルードを防ぐために、`#ifndef`、`#define`、`#endif`といったインクルードガードを使用することが推奨されます。

```
C/C++
// kazuate.h (例)
#ifndef _KAZUATE_H
#define _KAZUATE_H

void initialize();                  //【初期化】 乱数の種を現在の時刻に基づいて設定
void gen_no();                            //【問題の作成】 0～max_noの値を乱数で生成
int judge(int cand);                //【解答の判定】 candが正解かどうかを判定
int input();                        //【解答の入力】 0～max_noの値を入力させる
bool confirm_retry();               //【続行の確認】 再ゲームを行うかを確認

extern int max_no;                  // 当てるべき数の最大値

#endif
```

動作原理：
1. コンパイラがこのファイルを初めて処理する際、`_KAZUATE_H` マクロは未定義なので、`#ifndef` 条件は真となり、コンパイラは `_KAZUATE_H` の定義とヘッダファイル全体のコンテンツを処理します。
2. コンパイラが同じコンパイル単位内で再び kazuate.h のインクルード要求に遭遇した場合、`_KAZUATE_H` マクロは既に定義されているため、`ifndef` 条件は偽となり、コンパイラはファイル全体の内容を`#endif`までスキップし、再定義エラーを回避します。

ベストプラクティス：すべてのヘッダファイルはヘッダガードを使用すべきです。マクロ名は通常、ファイル名に基づいており、他のマクロとの衝突を避けるためにユニークな形式（先頭の下線など）を使用します。

---

10. 名前空間

識別子の有効範囲を論理的に制御するための手段として、C++では名前空間 **(namespace)** が提供されています。本節では、名前空間の定義法や利用法などを学習します。

名前空間の定義

識別子の有効範囲が関数の外であるか内であるかによって、識別子の有効範囲が変わることを第6章で学びました。また、`static`の有無によって、識別子に与えられる結合（その識別子がソースファイルの内部にとどまるか、外部にまで通用するのか）が変わることも学びました。

source file中の宣言の静的な位置に有効範囲が依存することや、source fileという物理的な単位に名前の通用範囲が依存することは、C++のベースとなったC言語から引き継がれた性質です。

C++では、個々の識別子の通用する範囲を名前空間 **(namespace)** という論理的な単位で制御できるように改良されています。`List 9-14`は、異なる名前空間内に同一名の変数と関数を定義し、それらを使い分けるプログラム例です。

```C/C++
// List 9-14: 2つの名前空間
#include <iostream>

using namespace std;

namespace English {
    int x = 1;
    void print_x() {
        cout << "The value of x is " << x << ".\n";
    }
    void hello() {
        cout << "Hello!\n";
    }
}

namespace Japanese {
    int x = 2;
    void print_x() {
        cout << "変数の値は" << x << "です。\n";
    }
    void hello() {
        cout << "こんにちは！\n";
    }
}

int main()
{
    cout << "English::x = " << English::x << '\n';
    English::print_x();
    English::hello();
```

```
    cout << "Japanese::x = " << Japanese::x << '\n';
    Japanese::print_x();
    Japanese::hello();
}
```

`namespace 識別子 { ... }` の形式で名前空間を定義します。名前空間に属するメンバ（変数や関数）にアクセスするには、`名前空間名::メンバ名` のようにスコープ解決演算子 `::` を使用します。

`English`と`Japanese`という2つの異なる名前空間に、それぞれ`x`, `print_x`, `hello`が定義されています。これにより、グローバル空間での名前の衝突を避けることができます。

名前空間のメンバ

入れ子になった名前空間

名前空間は入れ子にすることができます。これにより、名前をさらに細かくグループ化できます。

```C/C++
namespace Outer {
    int x;
    namespace Inner {
        int x;
    }
}
```

`Outer::x`と`Outer::Inner::x`は異なる変数を指します。

名前なし名前空間 (Unnamed Namespace)

名前をつけずに名前空間を定義することもできます。これを名前なし名前空間と呼びます。

```C/C++
namespace {
    int kotake = 0; // 名前を与えない
}
```

名前なし名前空間に属するメンバーは、それが定義されたsource file内でのみ有効です。これは、`static`を付けて宣言された変数や関数が内部結合を持つことと実質的に同じ効果をもたらします。

重要: source file中で内部結合を与える(`static`を付けて宣言された)関数やオブジェクトは、名前空間に所属するのと等価として定義されます。

using宣言とusing指令

using宣言

毎回`名前空間名::`と記述するのは煩雑です。特定のメンバーを頻繁に利用する場合、**`using`**宣言を用いることで、そのメンバーを現在のスコープに導入できます。

```C/C++
// List 9-15: using宣言
#include <iostream>

namespace English { /* ... */ }
namespace Japanese { /* ... */ }

int main()
{
    using Japanese::hello; // Japanese::helloをこのスコープに導入

    English::hello();
    hello(); // Japanese::hello()と同じ
}
```

using指令

名前空間のすべてのメンバを一度に現在のスコープに導入したい場合は、**`using`**指令を使用します。

```C/C++
using namespace std;
```

この`using namespace std;`という記述は、C++の標準ライブラリが属する`std`名前空間のすべてのメンバをグローバルスコープに導入することを意味します。これにより、`std::cout`を`cout`のように、名前空間名を省略して記述できるようになります。

ただし、`using`指令は意図しない名前の衝突を引き起こす可能性があるため、特にヘッダファイル内での使用は避けるべきです。関数内などの限定されたスコープで使用するか、`using`宣言で個別にメンバを導入する方法がより安全です。

重要: `using`指令の使用は、極力`std`名前空間にとどめ、それ以外の名前空間に対しては`using`宣言を使うべきです。

本講のまとめ

本講では、CとC++における関数の高度なトピックを深く探求しました。C++固有の参照渡しや関数オーバーロードといった機能だけでなく、C言語におけるポインタ渡しや関数ポインタを用いた伝統的な実現方法も学びました。また、`const`キーワードが堅牢で明確なコードを書く上での重要性、スコープと記憶期間という中心的な概念の区別、そしてインライン関数と関数マクロを比較することで、C++がパフォーマンスを追求しつつも、より安全な言語機能を通じてリスクをいかに回避しているかを理解しました。これらのツールは、C/C++における強力で柔軟な関数プログラミングモデルを形成します。