第6回 関数の基本的知識と演習

本講では、C++プログラミングにおける関数の基本的な概念と使用方法について解説します。

講義概要

1. 関数入門
    - 関数とは何か？
    - 簡単な例：最大値を求める
    - 関数の定義と呼び出し
2. 関数の詳細
    - 関数定義：返却値の型、関数名、仮引数宣言
    - 関数本体と関数頭部
    - 関数呼び出しと実引数
    - `return`文
3. **`main`**関数
    - `main`関数の特殊性
4. 関数宣言
    - なぜ関数宣言が必要か？
    - 関数プロトタイプ
5. 引数渡し
    - 値渡し (Pass by Value)
6. **`void`**関数
    - 返却値のない関数
7. 関数の設計
    - 汎用性を高める
    - 関数間の呼び出し
8. 特殊な関数
    - 引数を受け取らない関数
    - 再帰呼び出し
    - デフォルト実引数を持つ関数
9. 標準ライブラリ関数
    - `abort`と`exit`
10. ビット単位の演算と関連関数
    - ビット単位の論理演算子
    - シフト演算子
    - サンプルコード：ビット演算と補助関数

---

1. 関数入門

関数は、特定のタスクを実行するためのC++プログラム内のコードブロックです。コードを関数にまとめることで、プログラムのモジュール性が高まり、理解や保守が容易になります。

関数とは何か？

関数はプログラムの独立した部分であり、何らかの入力（引数）を受け取り、特定の計算や操作を実行し、結果を返すことがあります。関数は「ブラックボックス」と見なすことができ、その内部実装の詳細を気にすることなく、その機能と使用方法を知っていれば十分です。

簡単な例：最大値を求める

簡単な例から始めましょう。以下のコードList 6-1は、3人の数学と英語の成績を読み込み、各科目の最高点をそれぞれ計算して表示するプログラムです。

C/C++

```
// List 6-1: 3人の数学・英語の最高点を求めて表示
#include <iostream>

using namespace std;

int main()
{
    int math[3]; // 数学の点数
    int eng[3];  // 英語の点数

    // 点数の読み込み
    for (int i = 0; i < 3; i++)
    {
        cout << "数学 " << i + 1 << ": ";
        cin >> math[i];
        cout << "英語 " << i + 1 << ": ";
        cin >> eng[i];
    }

    // 数学の最高点
    int max_math = math[0];
    if (math[1] > max_math) max_math = math[1];
    if (math[2] > max_math) max_math = math[2];

    // 英語の最高点
    int max_eng = eng[0];
    if (eng[1] > max_eng) max_eng = eng[1];
    if (eng[2] > max_eng) max_eng = eng[2];
```

```
    cout << "数学の最高点は " << max_math << " です。\n";
    cout << "英語の最高点は " << max_eng << " です。\n";
}
```

この例では、最高点を見つけるためにほぼ同じロジックを2回繰り返しています。このロジックを関数に抽出することで、コードを簡素化し、再利用性を高めることができます。

関数の定義と呼び出し

3つの整数を入力として受け取り、その中の最大値を返す`calc_max`という名前の関数を定義できます。

```C/C++
// a, b, cの最大値を返す
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;
}
```

そして、`main`関数でこの`calc_max`関数を「呼び出す」ことで、数学と英語の最高点を計算できます（`List 6-2`参照）。

```C/C++
// List 6-2: 関数版）3人の数学・英語の最高点を求めて表示
#include <iostream>

using namespace std;

// a, b, cの最大値を返す
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()
{
    int math[3];
    int eng[3];

    for (int i = 0; i < 3; i++)
    {
        cout << "数学 " << i + 1 << ": ";
        cin >> math[i];
        cout << "英語 " << i + 1 << ": ";
        cin >> eng[i];
    }

    int max_math = calc_max(math[0], math[1], math[2]); // calc_max関数を呼び出す
    int max_eng = calc_max(eng[0], eng[1], eng[2]);   // 再度calc_max関数を呼び出
す

    cout << "数学の最高点は " << max_math << " です。\n";
    cout << "英語の最高点は " << max_eng << " です。\n";
}
```

関数を使用することで、コードがより簡潔で理解しやすくなります。

---

2. 関数の詳細

関数の構成要素をさらに詳しく見ていきましょう。

関数定義

関数定義は関数頭部 **(function header)** と関数本体 **(function body)** から構成されます。

C/C++
```
int calc_max(int a, int b, int c)
{
    // 関数本体
    int max_val = a;
    if (b > max_val) max_val = b;
    if (c > max_val) max_val = c;
    return max_val;
}
```

関数頭部

関数頭部は、関数の「シグネチャ」を指定します。これには以下が含まれます。

1. 返却値の型 **(Return Type)**: 関数が実行を終えた後、呼び出し元に返す値の型。`calc_max`関数では、返却値の型は`int`です。
2. 関数名 **(Function Name)**: 関数を呼び出すために使用される名前。例:`calc_max`。
3. 仮引数宣言 **(Parameter Declaration List)**: 関数が受け取る入力のリスト。各仮引数は、指定された型と名前を持つ、関数内で使用される変数のようなものです。`calc_max`関数は、`int`型の3つの仮引数`a`、`b`、`c`を受け取ります。

関数本体

関数本体は`{}`で囲まれたコードブロックで、関数の具体的な実装が含まれています。関数本体内で宣言された変数(`calc_max`関数の`max_val`変数など)はローカル変数であり、その関数内でのみアクセス可能です。

関数呼び出しと引数渡し

関数呼び出し **(Function Call)** とは、関数名を使用し、実引数を提供して関数を実行するプロセスです。

```C/C++
int max_math = calc_max(math[0], math[1], math[2]);
```

- 実引数 **(Argument)**: 関数呼び出し時に、関数に渡される値。上記の例では、`math[0]`、`math[1]`、`math[2]`が実引数です。
- 仮引数 **(Parameter)**: 関数定義で宣言され、実引数の値を受け取るための変数。`calc_max`関数では、`a`、`b`、`c`が仮引数です。

関数が呼び出されると、実引数の値が対応する仮引数にコピーされます。このプロセスを値渡し **(Pass by Value)** と呼びます。

以下の`List 6-3`は、3つの整数の最大値を計算して表示する別の例です。

```C/C++
// List 6-3: 三つの整数の最大値を求めて表示
#include <iostream>

using namespace std;
```

```cpp
// a, b, cの最大値を返す
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()
{
    int a, b, c;
    cout << "整数a: "; cin >> a;
    cout << "整数b: "; cin >> b;
    cout << "整数c: "; cin >> c;

    cout << "最大値は " << calc_max(a, b, c) << " です。\n";
}
```

return文

return文は、関数の結果を呼び出し元に返すために使用されます。return文が実行されると、関数は即座に終了し、指定された値が返されます。1つの関数に複数のreturn文を含めることができます。

---

3. main関数

main関数は、C++プログラムにおいて非常に特殊な関数です。これはプログラムの開始点であり、プログラムが起動するとmain関数が自動的に呼び出されます。

返却値の型と返却値

main関数の返却値の型は通常intです。main関数が実行を終えて値を返すと、プログラムは終了します。その返却値はオペレーティングシステムに渡され、プログラムの実行結果を示します。

- `return 0;` はプログラムが正常に終了したことを示します。
- 0以外の値を返すと、通常はプログラムで何らかのエラーが発生したことを示します。

main関数の末尾では、たとえ`return 0;`を明示的に記述しなくても、コンパイラが自動的に追加するため、省略することが可能です。

`main`関数の制限

他の関数と比較して、`main`関数にはいくつかの特別な制限があります。

(1) 多重定義できない **(Overload)**: 1つのプログラムに`main`関数は1つしか存在できません。
(2) インライン関数にできない **(Inline)**。
(3) 再帰呼び出しできない **(Recursion)**。
(4) アドレスを取得できない。

なお、C言語では (3) と (4) の制限はありません。
これらの概念については、今後の講義で学びます。

---

4. 関数宣言

C++コンパイラはソースコードを上から下に順に処理します。これは、関数を定義する前に呼び出そうとすると、コンパイラがその関数の定義を見つけられずにエラーを出すことを意味します。

例えば、`calc_max`関数の定義を`main`関数の後に置くと、コンパイルエラーが発生します。

```C/C++
// コンパイルエラーの例
#include <iostream>

using namespace std;

int main()
{
    int a = 1, b = 2, c = 3;
    cout << "最大値は " << calc_max(a, b, c) << " です。\n"; // エラー: 'calc_max'
がこのスコープで宣言されていません
}

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;
}
```

この問題を解決するために、関数を呼び出す前に関数宣言 **(Function Declaration)**、別名関数プロトタイプ **(Function Prototype)** を提供することができます。

関数宣言は、コンパイラに関数の名前、返却値の型、仮引数のリストを伝えますが、関数本体は含みません。これにより、関数の完全な定義が後にあっても、コンパイラはそれを正しく呼び出す方法を知ることができます。

関数宣言の形式は以下の通りです。末尾のセミコロンに注意してください。

```C/C++
int calc_max(int a, int b, int c);
```

関数宣言では、仮引数名を省略できます。

```C/C++
int calc_max(int, int, int);
```

List 6-5は、関数宣言を使って上記のコードを修正する方法を示しています。

```C/C++
// List 6-5: 関数宣言の追加）三つの整数の最大値を求める
#include <iostream>

using namespace std;

// 関数宣言（関数プロトタイプ）
int calc_max(int a, int b, int c);

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

    cout << "最大値は " << calc_max(a, b, c) << " です。\n";
}

// 関数定義
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;
}
```

重要: まだ定義されていない関数を呼び出す前には、必ず関数宣言が必要です。私たちが以前使用したcoutやその他の標準ライブラリ機能の宣言は、#include <iostream>などのヘッダファイル内にあります。

---

5. 引数渡し

C++では、デフォルトの引数渡しの方法は値渡し **(Pass by Value)** です。

値渡し (Pass by Value)

値渡しを使用すると、関数は各仮引数のコピーを作成します。関数内部でこれらのコピーに対して行われたいかなる変更も、呼び出し元が提供した元の実引数には影響しません。

List 6-6に示す、べき乗を計算するpower関数を通してこの概念を理解しましょう。

```C/C++
// List 6-6: xのn乗を求める
#include <iostream>

using namespace std;

// xのn乗を返す
double power(double x, int n)
{
    double tmp = 1.0;
    for (int i = 1; i <= n; i++)
    {
        tmp *= x;
    }
    return tmp;
}

int main()
{
```

```
    double a;
    int b;

    cout << "aのb乗を求めます。\n";
    cout << "実数a: "; cin >> a;
    cout << "整数b: "; cin >> b;

    cout << a << " の " << b << " 乗は " << power(a, b) << " です。\n";
}
```

`power(a, b)`の呼び出しでは:

1. `main`関数内の変数`a`の値が、`power`関数の仮引数`x`にコピーされます。
2. `main`関数内の変数`b`の値が、`power`関数の仮引数`n`にコピーされます。

`power`関数は、これら`x`と`n`という2つのコピーを操作します。たとえ`power`関数内で`x`や`n`の値を変更しても、`main`関数内の`a`と`b`は一切変更されません。これは、家の鍵のコピーを友人に渡すようなものです。友人はコピーにどんな印をつけても、あなたが持っている元の鍵には影響しません。

`List 6-7`は、`while`ループを使用して実装された同等の`power`関数を示しています。このバージョンでは、仮引数`n`の値がループ内でデクリメントされますが、これも同様に`main`関数内の元の変数`b`の値には影響しません。

```C/C++
// List 6-7: while文によるxのn乗
double power(double x, int n)
{
    double tmp = 1.0;
    while (n-- > 0)
    {
        tmp *= x;
    }
    return tmp;
}
```

重要: 値渡しは、C++における関数への引数渡しの基本的な方法です。これにより、関数が呼び出し元のデータを予期せず変更することを防ぎ、コードの安全性と予測可能性を高めます。

---

6. void関数

一部の関数は、特定の操作を実行しますが、値を返す必要がありません。例えば、画面にテキストを表示するだけの関数などです。このような関数については、返却値の型としてvoidを使用できます。

voidキーワードは「何もない」または「空」を意味します。

List 6-8は、指定された数のアスタリスク(*)を印字するput_starsという名前のvoid関数を示しています。

```C/C++
// List 6-8: 左下が直角の二等辺三角形を表示
#include <iostream>

using namespace std;

// n個の'*'を連続表示
void put_stars(int n)
{
    while (n-- > 0)
    {
        cout << '*';
    }
}

int main()
{
    int n;
    cout << "何個*を表示しますか：";
    cin >> n;

    for (int i = 1; i <= n; i++)
    {
        put_stars(i);
        cout << '\n';
    }
}
```

```
何個*を表示しますか：6
*
**
***
****
*****
******
```

`put_stars`関数は値を返さないため、`cout << put_stars(5);`のような式では使用できません。

`void`関数内では、`return;`文を使用して関数の実行を早期に終了させることができますが、その後に値を続けることはできません。関数が最後まで実行されると自動的に戻るため、関数の末尾にある`return;`は不要です。

---

7. 関数の設計

良い関数設計の目標の一つは、汎用性 **(Reusability)** を高めることです。設計の良い関数は、さまざまな状況で使用できるほど汎用的であるべきです。

汎用性を高める

以前作成した`put_stars`関数はアスタリスクしか印字できませんでした。もし"+"のような他の文字を印字したい場合は、別の`put_plus`関数を作成する必要があります。これはコードの冗長につながります。

より良い方法は、印字する文字を引数として受け取ることができる、より汎用的な関数を作成することです。`List 6-9`の`put_nchar`関数がその一例です。

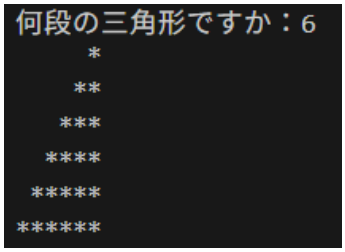
```C/C++
// List 6-9: 右下が直角の二等辺三角形を表示
#include <iostream>

using namespace std;

// 文字cをn個連続表示
void put_nchar(char c, int n)
{
    while (n-- > 0)
    {
        cout << c;
    }
}

int main()
{
    int n;
    cout << "何段の三角形ですか：";
    cin >> n;
```

```
    for (int i = 1; i <= n; i++)
    {
        put_nchar(' ', n - i); // 空白を印字
        put_nchar('*', i);     // アスタリスクを印字
        cout << '\n';
    }
}
```



put_starsをput_ncharに汎用化することで、より強力で再利用性の高いツールを手に入れることができます。

関数間の呼び出し

関数はmain関数からだけでなく、他の関数からも呼び出すことができます。これにより、単純な関数を組み合わせて、より複雑な機能を構築することができます。

List 6-10は、以前作成したput_nchar関数を使用して、正方形を印字する関数と長方形を印字する関数の2つの新しい関数を作成する方法を示しています。

C/C++
```
// List 6-10: 正方形と長方形を表示
#include <iostream>

using namespace std;

// 文字cをn個連続表示
void put_nchar(char c, int n)
{
    while (n-- > 0)
    {
        cout << c;
    }
}

// 文字cからなる一辺nの正方形を表示
void put_square(int n, char c)
{
```

```
    for (int i = 1; i <= n; i++)
    {
        put_nchar(c, n);
        cout << '\n';
    }
}

// 文字cからなる高さh、幅wの長方形を表示
void put_rectangle(int h, int w, char c)
{
    for (int i = 1; i <= h; i++)
    {
        put_nchar(c, w);
        cout << '\n';
    }
}

int main()
{
    int n, h, w;
    cout << "正方形の一辺："; cin >> n;
    put_square(n, '*');

    cout << "長方形の高さ："; cin >> h;
    cout << "長方形の幅："; cin >> w;
    put_rectangle(h, w, '+');
}
```

重要: 関数を設計する際は、できるだけ汎用的で再利用可能にするよう努めるべきです。単純な関数を組み合わせることで、コードの明瞭さとモジュール性を保ちながら、複雑なプログラムを構築できます。

---

8. 特殊な関数

これまで説明してきた基本的な関数に加えて、C++は特殊な振る舞いを持ついくつかの関数型をサポートしています。

引数を受け取らない関数

関数が呼び出し元から情報を受け取る必要がない場合、それを引数を受け取らない関数として定義できます。定義時、仮引数リストは`(void)`になります。

C言語では、`()` と書くこともできますが、この場合コンパイラは引数の有無や型をチェックしません。したがって、明確に「この関数は引数を受け取りません」と示すために、`(void)` と書くことが推奨されます。

なお、C++ では `()` と `(void)` の意味は同じですが、C では動作が異なる点に注意が必要です。

`List 6-11`の`confirm_retry`関数がその一例です。これはユーザーに再試行するかどうかを尋ね、引数を一切受け取りません。

```C/C++
// List 6-11: 続行の確認
#include <ctime>
#include <cstdlib>
#include <iostream>

using namespace std;

// 続行の意思を確認
bool confirm_retry(void)
{
    int retry;
    do
    {
        cout << "もう一度？<Yes...1/No...0>: ";
        cin >> retry;
    } while (retry != 0 && retry != 1);
    return static_cast<bool>(retry);
}

int main()
{
    srand(time(NULL));
    cout << "暗算トレーニング開始!!\n";

    do
    {
        for(int i = 0; i < 3; i++)
        {
            int x = rand() % 90 + 10; // 2桁の数
            int y = rand() % 90 + 10; // 2桁の数
            cout << "問題" << i + 1 << ": " << x << " + " << y << " = ";
            int k; // 読み込んだ値
            cin >> k;
            if (k == x + y) // 正解
                cout << "正解!\n";
            else
```

```
                cout << "違いますよ!\n";
        }
    } while (confirm_retry()); // 引数を与えない
}
```

引数を受け取らない関数を呼び出す際も、関数名の後の括弧`()`は必要です。

再帰呼び出し

再帰関数 **(Recursive Function)** とは、その関数本体内で自身を直接または間接的に呼び出す関数です。再帰は、元の問題と似た部分問題に分解できる特定の問題を解決するための強力なツールです。

典型的な例は階乗の計算です。nの階乗（n!と表記）は、n以下のすべての正の整数の積です。例えば、5! = 5 × 4 × 3 × 2 × 1 = 120です。階乗は次のように定義できます。

- `n! = n * (n-1)!` (n > 0 の場合)
- `0! = 1`

`List 6C-1`は、再帰呼び出しを用いて階乗を計算する方法を示しています。

C/C++
```
// List 6C-1: 再帰呼び出しを用いて階乗値を求める
#include <iostream>

using namespace std;

// 非負整数nの階乗値を求める
int factorial(int n)
{
    if (n > 0)
        return n * factorial(n - 1); // 再帰呼び出し
    else
        return 1; // 再帰の基底部
}

int main()
{
    int x;
    cout << "整数を入力せよ：";
    cin >> x;
    cout << x << " の階乗は " << factorial(x) << " です。\n";
```

```
}
```

`factorial`関数において、`factorial(n - 1)`が再帰呼び出しです。無限再帰を防ぐためには、再帰を終了させるための基底部 **(Base Case)** が必要です。この例では、`n == 0`が基底部です。

デフォルト実引数を持つ関数

C++では、関数の仮引数にデフォルト値を指定することができます。関数呼び出し時にその仮引数に対する実引数が提供されない場合、コンパイラは自動的にこのデフォルト値を使用します。

`List 6-12`の`alerts`関数は、デフォルト実引数の使用法を示しています。これは指定された回数だけ警告音を鳴らします。

```C/C++
// List 6-12: 警報を発する関数（デフォルト実引数）
#include <iostream>

using namespace std;

// n回警報を発する
void alerts(int n = 3) // nのデフォルト値は3
{
    while (n-- > 0)
    {
        cout << "\a"; // \aは警報文字
    }
}

int main()
{
    alerts();      // 引数を与えずに呼び出すと、nはデフォルト値3を使用
    cout << "再度警報！\n";
    alerts(5);     // 引数5を与えて呼び出すと、nの値は5になる
}
```

重要: デフォルト実引数は、仮引数リストの右側から指定する必要があります。つまり、ある仮引数にデフォルト値を設定した場合、その右側にあるすべての仮引数にもデフォルト値を設定しなければなりません。

```C/C++
void func(int a, int b = 5, int c = 10); // 正しい
void func(int a = 1, int b, int c = 10); // 誤り
```

---

9. 標準ライブラリ関数

C++標準ライブラリは、私たちがプログラムで使用できる、あらかじめ作成された多数の関数を提供しています。これらの関数は、入出力から数学計算まで、さまざまな一般的な機能をカバーしています。これらの関数を使用するには、通常、対応するヘッダファイルをインクルードする必要があります。

例えば、`<cstdlib>`ヘッダファイルは、`abort`や`exit`など、プログラム制御用のいくつかの関数を提供します。

`abort`と`exit`：プログラムの終了

状況によっては、`main`関数が終了する前にプログラムの実行を終了させたい場合があります。

- **`abort()`**: プログラムを強制的に終了させます。これは異常終了であり、通常、プログラムが安全に続行できない深刻なエラーに遭遇した場合に使用されます。

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

// ...
abort(); // プログラムは即座に終了する
```

- **`exit(status)`**: プログラムを正常に終了させます。`exit`関数はいくつかのクリーンアップ処理を実行してからプログラムを終了します。これは終了ステータスコードとして整数引数を受け取り、その役割は`main`関数の`return`文と似ています。`exit(0)`は通常、成功裡の終了を示します。

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

// ...
exit(0); // プログラムは正常に終了する
```

これら2つの関数は、プログラムのどこからでも呼び出すことができ、即座にプログラムを終了させます。

10. ビット単位の演算と関連関数

C++は、整数のビットを直接操作するための一連の強力なビット単位演算子を提供します。これらは、低レベルのプログラミング、パフォーマンスの最適化、ハードウェアとのやり取りなどのシナリオで非常に役立ちます。

ビット単位の論理演算子

ビット単位の論理演算子は、オペランドの各ビットに対して論理演算を実行します。

- & (ビット積 / Bitwise AND): 両方のオペランドの対応するビットが両方とも1の場合にのみ、結果のビットが1になります。
- | (ビット和 / Bitwise OR): 両方のオペランドの対応するビットの少なくとも1つが1の場合に、結果のビットが1になります。
- ^ (排他的論理和 / Bitwise XOR): 両方のオペランドの対応するビットが異なる場合に、結果のビットが1になります。
- ~ (補数 / Complement): オペランドのすべてのビットを反転させます。0は1に、1は0になります。

List 6-13は、これらの演算子の使用法を示し、整数の内部表現を確認するためのいくつかの補助関数を含んでいます。

```C/C++
// List 6-13: 符号無し整数の論理積・論理和・排他的論理和・1の補数を表示
#include <iostream>

using namespace std;

//--- 整数x中の"1"のビット数を求める ---//
int count_bits(unsigned x)
{
    int bits = 0;
    while (x)
    {
        if (x & 1U) bits++;
        x >>= 1;
    }
    return bits;
}
```

```
//--- unsigned型のビット数を求める ---//
int int_bits()
{
    return count_bits(~0U);
}

//--- unsigned型のビット構成を表示 ---//
void print_bits(unsigned x)
{
    for (int i = int_bits() - 1; i >= 0; i--)
    {
        // 1U は、符号なし整数（unsigned int 型）の定数 1
        cout << ((x >> i) & 1U ? '1' : '0');
    }
}

int main()
{
    unsigned a, b;
    cout << "非負の整数を二つ入力せよ。\n";
    cout << "a: "; cin >> a;
    cout << "b: "; cin >> b;

    cout << "a      = "; print_bits(a); cout << '\n';
    cout << "b      = "; print_bits(b); cout << '\n';
    cout << "a & b  = "; print_bits(a & b); cout << '\n';
    cout << "a | b  = "; print_bits(a | b); cout << '\n';
    cout << "a ^ b  = "; print_bits(a ^ b); cout << '\n';
    cout << "~a     = "; print_bits(~a); cout << '\n';
    cout << "~b     = "; print_bits(~b); cout << '\n';
}
```

補助関数の説明:

- `print_bits(unsigned x)`: 符号無し整数`x`のビット構成を表示します。
- `count_bits(unsigned x)`: `x`に含まれる1のビット数を数えます。
- `int_bits()`: `unsigned`型の総ビット数を返します（例：32ビット環境では通常32）。

シフト演算子

シフト演算子は、オペランドの全ビットを指定されたビット数だけ左または右に移動させます。

- `<<` (左シフト / Left Shift): `x << n` は `x` の全ビットを `n` ビット左にシフトします。右側から空いたビットは0で埋められます。`n` ビットの左シフトは、2の`n`乗を乗算することと等価です。

- `>>` (右シフト / Right Shift): `x >> n` は `x` の全ビットを `n` ビット右にシフトします。符号無し数の場合、左側から空いたビットは0で埋められます。符号付き数の場合、論理シフト（0で埋める）か算術シフト（符号ビットで埋める）かは処理系に依存します。

`List 6-14`は、シフト演算子の効果を示しています。

```C/C++
// List 6-14: 符号無し整数をシフトした値を表示
#include <iostream>

// (List 6-13と同じcount_bits, int_bits, print_bits関数は省略)

int main()
{
    unsigned x, n;
    cout << "非負の整数："; cin >> x;
    cout << "シフトするビット数："; cin >> n;

    cout << "整数                 = "; print_bits(x); cout << '\n';
    cout << "左に " << n << " ビットシフト = "; print_bits(x << n); cout << '\n';
    cout << "右に " << n << " ビットシフト = "; print_bits(x >> n); cout << '\n';
}
```

論理シフトと算術シフト

- 論理シフト: 符号ビットを特別扱いせず、空いたビットを常に0で埋めます。C++では、符号無し整数に対する右シフトは論理シフトとして実行されます。
- 算術シフト: 右シフトの際、元の値が正であれば空いたビットを0で、負であれば符号ビット（1）で埋めます。これにより、数値の算術的な意味（2のべき乗での除算にほぼ等しい）が保たれます。符号付き整数の右シフトは、通常、算術シフトです。

本回のまとめ

本回では、C/C++の中核的な概念である「関数」について深く学びました。関数の基本的な定義と呼び出しから始め、関数を通じてコードをモジュール化し、コードの再利用性と可読性を向上させる方法を理解しました。関数頭部と関数本体の構成を探り、仮引数と実引数、そして値渡しのメカニズムを学びました。さらに、`main`関数の特殊性、関数宣言の必要性、そして`void`関数、再帰呼び出し、デフォルト実引数を持つ関数など、さまざまな特殊な関数の使用法を紹介しました。最後に、標準ライブラリ関数に触れ、それらを使ってプログラムを強化する方法を学びました。

関数の習得は、C++プログラミングにおける重要な一歩です。今後の学習では、より複雑で強力なアプリケーションを構築するために、絶えず関数を使用していくことになります。

課題

(1)関数を作成し、与えられた正の整数が素数かどうかを判定しなさい。さらに、main関数内で 100 から200までの整数の中に含まれるすべての素数 を求め、出力しなさい。出力の際は、整数の間を1つの空白で区切って表示しなさい。
（プログラム名：**ex06_1.cpp**）

入力：なし
出力：

```
101 103 107
```

(2)関数を作成し、与えられた西暦年がうるう年（閏年）かどうかを判定しなさい。さらに、main関数内で **21**世紀（**2000**年～**2099**年）に含まれるすべてのうるう年を求め、出力しなさい。出力の際は、整数の間を1つの空白で区切って表示しなさい。
（プログラム名：**ex06_2.cpp**）

うるう年の条件：

- 400で割り切れる年、または
- 4で割り切れて100で割り切れない年。

入力：なし
出力：

```
2000 2004 2008
```

(3)再帰関数を用いてフィボナッチ数を出力しなさい。
整数 n を読み込み、次の規則で処理せよ。計算は再帰関数を定義して行うこと。

- n <= 0 のとき：「無効な値」を出力する
- n > 30 のとき：「数値が大きすぎます」を出力する
- それ以外のとき：「**F(n)=値**」の形式で第 n 項を出力する

フィボナッチ数列の定義：

F(1) = 1，F(2) = 1，F(k) = F(k-1) + F(k-2)（k ≥ 3）。
（プログラム名：**ex06_3.cpp**）

【実行例】

	入力	出力
例1	**3**⏎	**F(3)=2**
例2	**10**⏎	**F(10)=55**
例3	**-2**⏎	無効な値
例4	**100**⏎	数値が大きすぎます