C++プログラミングI

- 第2回：bool型と論理式
- 担当：二瓶芙巳雄

bool型の基本

- `bool` 型の値は `true` と `false`
- C言語との互換性のために整数も扱う
  - 0が `false`, 0以外が `true`
  - `cout` の出力, 初期化に注意
- `std::boolalpha` : bool値を文字列として入出力する
  - ※ `std::cout` （等）に併用して，出力の挙動を変えるものをマニピュレータと呼ぶ

```cpp
#include <iostream>

int main(){
  bool x{true}, y{}; // {} は0 → false
  std::cout << x << " " << y << "\n";
  std::cout << std::boolalpha << x << " " << y << "\n";

  y = x;
  std::cout << x << " " << y << "\n";

  bool a{1}, b{0};
  std::cout << a << " " << b << "\n";

  // bool c{2}; // エラー. narrowing conversion.
  // bool c = 2; // エラーにはならない. おすすめしません.
}
```

```
% ./a.out
1 0
true false
true true
true false
```

比較演算子

- 2つの値を比べる演算子
  - 二項演算子：オペランド（被演算子）が二つ
- **演算結果は`bool`型の値**

```cpp
#include <iostream>

int main(){
  bool x{ 3 < 4 }; // 比較結果でxを初期化

  int i{3}, j{4};
  bool y{};
  y = i > j;       // 比較結果をyに代入

  std::cout << std::boolalpha
            << (3 == 4) << " "
            << (3 != 4) << " "
            << x << " "
            << y << "\n";
}
```

演算子	意味
`<`	小なり
`<=`	小なりイコール
`>`	大なり
`>=`	大なりイコール
`==`	等しい
`!=`	等しくない

```
% ./a.out
false true true false
```

`bool` 変数と比較演算

- 比較の結果は `bool` 型変数の初期化や代入に利用できる
- 実数値に対する `==`, `!=` の使用は注意が必要
  - コンピュータ上での実数の扱いは概数、正確ではない場合があるため
- `char` 型でも大小比較が可能．ASCIIコード順（辞書順）に比較．

```cpp
#include <iostream>

int main(){
  bool x{3 < 4};   // xの初期値
  std::cout << std::boolalpha << x <<"\n";

  double z = 0.1 + 0.2;
  std::cout << (z==0.3) <<"\n"; // false!

  char a{'m'}, b{'n'};
  std::cout <<(a == b)<<" "<<(a <= b)<<"\n";
}
```

```
% ./a.out
true
false
false true
```

論理演算子

- 論理積：`&&`, 論理和：`||`, 論理否定：`!`
- 複数の条件を組み合わせるのに利用する
- `&&` と `||` は比較演算子よりも優先度が低い
  - 比較演算子：`>=`, `<`, など

`A`	`B`	`A && B`	`A || B`	`!A`
true	true	**true**	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	**false**	true

```cpp
#include <iostream>

int main(){
  int x {3};
  std::cout << std::boolalpha
            << x << ": "
            << (1 <= x && x <= 3) <<" "
            << (x <= -50 || x >= 100) <<"\n";
}
```

```
% ./a.out
3: true false
```

短絡評価

- `&&` と `||` の特別ルール
  - `&&` : 左側の式が `false` →右側の式の評価が省略
  - `||` : 左側の式が `true` →右側の式の評価が省略

```
1  int main() {
2    int x{0}, y{10};
3
4    bool b = (x == 0) || (y == 10);
5    // ↑の式は，true || true
6    // or条件は，一方がtrueなら演算結果はtrueなので，
7    // 左オペランド == trueの時点で
8    // 右オペランドを見る必要がなくなる
9  }
```

- 一つ目のオペランドで演算結果が決まるため，二つ目のオペランドは評価（計算）しない
- 二つ目の演算が副作用を持つ場合に重要
  - 入力や代入も結果を持つ式であることに注意

- 短絡評価の応用

```
1  #include <iostream>
2
3  int main(){
4    int x { 10 };
5    // x >= 0 でなければ，cout する（エラー処理）
6    x >= 0 || std::cout <<"A: xの入力エラー\n";
7
8    x = -1;
9    x >= 0 || std::cout <<"B: xの入力エラー\n";
10
11   // cin が成功しなかったら，cout する（エラー処理）
12   std::cin >> x || std::cout<<"C: error\n";
13 }
```

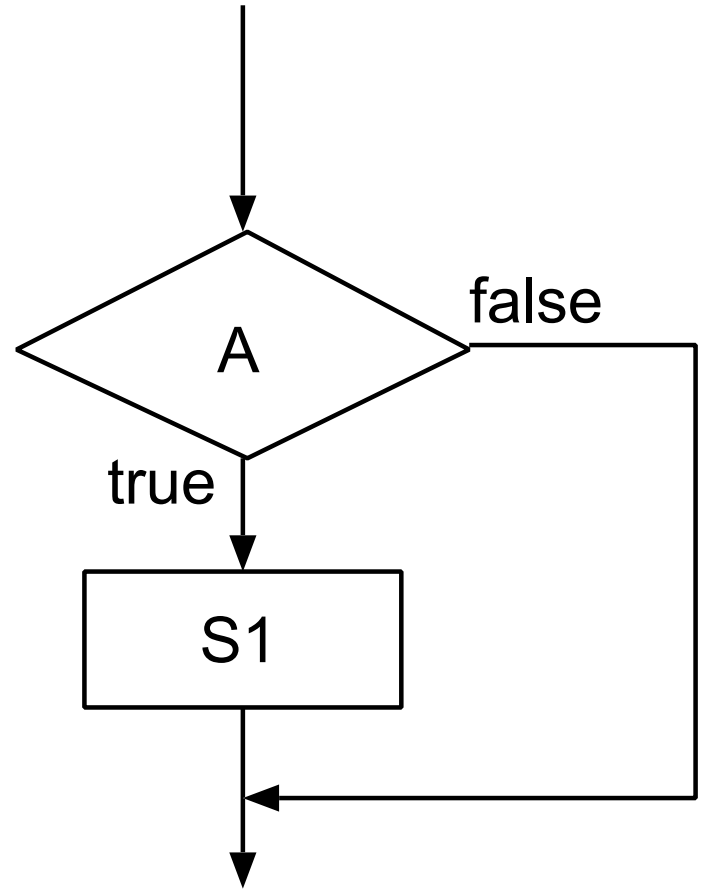
- `std::cin >> x` の戻り値は，`std::cin`．`bool(std::cin)` で，`true` or `false` が得られる
- `cin` が成功すると `true`，失敗すると `false`．失敗とは，期待しない型の値が入力された，`Ctrl + D` が入力された，など

条件文

if文

- `A`が`true`ならば`S1`を行う
  - 条件`A`は`bool`型の式
  - 文`S1`は一つの実行文
- 処理を複数行書く時は，中括弧`{}`で**複文**にする
- `if`文を1行に書いても良い
- インデントの位置決めも任意
  - 読みやすさのため，積極的に使用しましょう

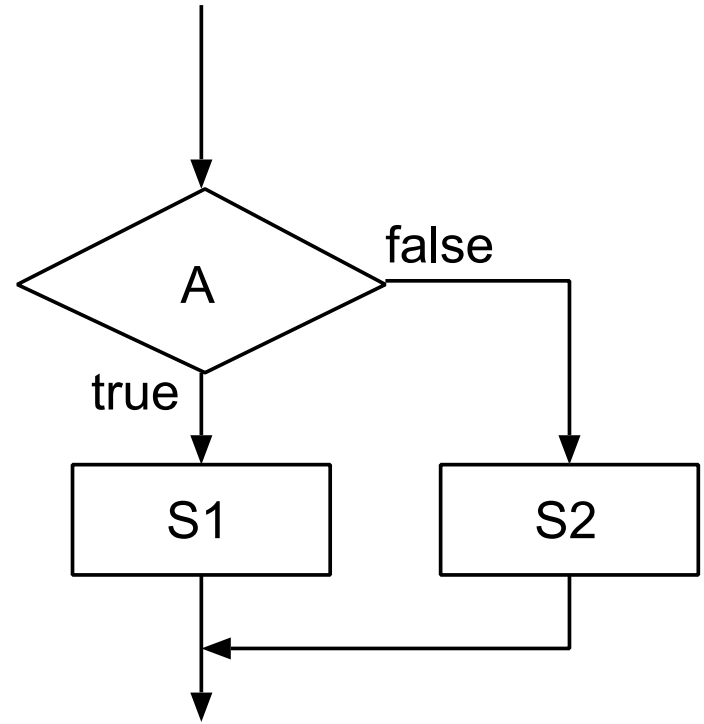
```
1   if (条件A)
2     文S1
```



if-else文

- `if` 文に加えて条件が `false` の場合の処理がある
- `S1` と `S2` はどちらも複文で指定できる

```
1  if (条件A)
2    文S1
3  else
4    文S2
```



if文if-else文の例

```cpp
#include <iostream>

int main(){
  double x {};
  std::cin >> x;

  if (x < 0.0) x = -x;

  if (x < 1.0) x += 10.0;
  else         x *= 10.0;

  if (x <= 10.0) {
     std::cout << "input error\n";
     x = 10.0;
  } else {
     std::cout << "Result is "<< x <<"\n";
  }
}
```

```
% ./a.out
-10
Result is 100
% ./a.out
-0.5
Result is 10.5
% ./a.out
-1
input error
% ./a.out
0.5
Result is 10.5
% ./a.out
1
input error
% ./a.out
10
Result is 100
```

- 複文を使うかどうかを検討する
- 処理全体が短ければ1行に書く方法もある

pythonとの比較：`if`文

```cpp
// cppのプログラム
#include <iostream>

int main(){
  double x {};
  std::cin >> x;

  if (x < 0.0) x = -x;

  if (x < 1.0) x += 10.0;
  else         x *= 10.0;

  if (x <= 10.0) {
     std::cout << "input error\n";
     x = 10.0;
  } else {
     std::cout << "Result is "<< x <<"\n";
  }
}
```

- `if( ... )`
- 中括弧`{}`で処理を制御

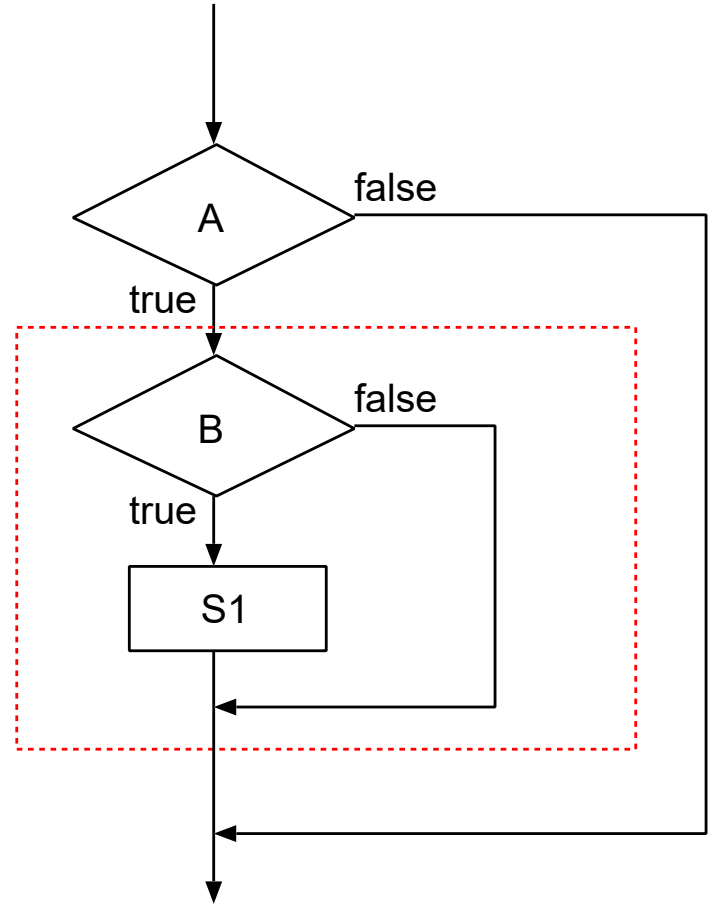
```python
# pythonのプログラム



x = float( input() )

if x < 0.0: x = -x

if x < 1.0: x += 10.0
else:       x *= 10.0

if x <= 10.0:
   print( "input error" )
   x = 10.0
else:
   print( "Result is", x )
```

- `if ...:`
- インデント（行頭の連続するスペース）で処理を制御

入れ子のif文

- `if`文の中に別の`if`文がある
- 条件`A`と`B`の組み合わせは論理積と同じ

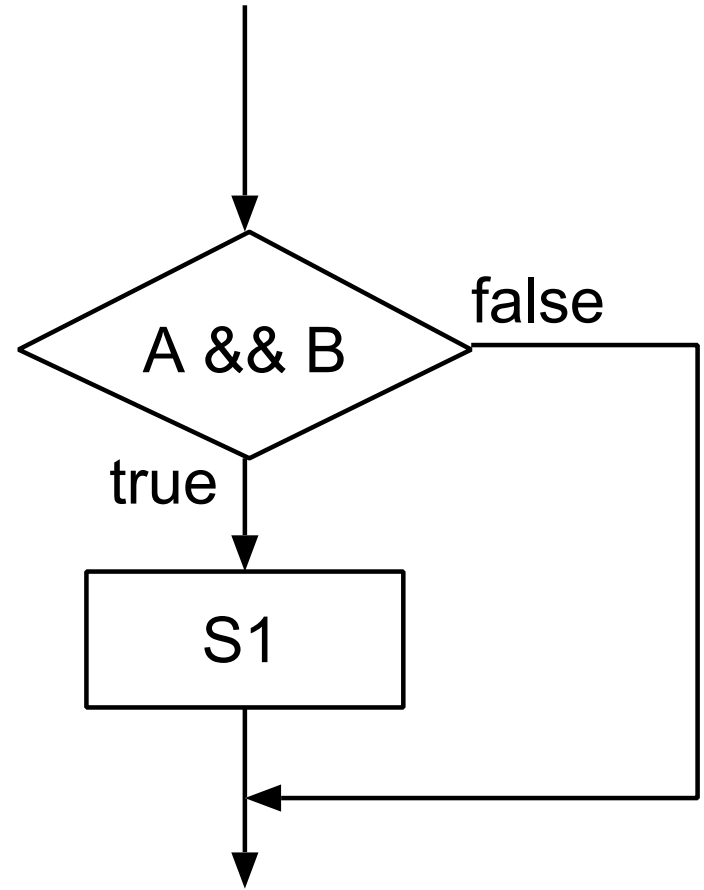
```
1  if (条件A) {
2      if (条件B) {
3          文S1
4      }
5  }
```



論理積の利用

- 入れ子の `if` 文は `&&` 演算子の短絡評価とマッチする
- 簡潔に記述できる

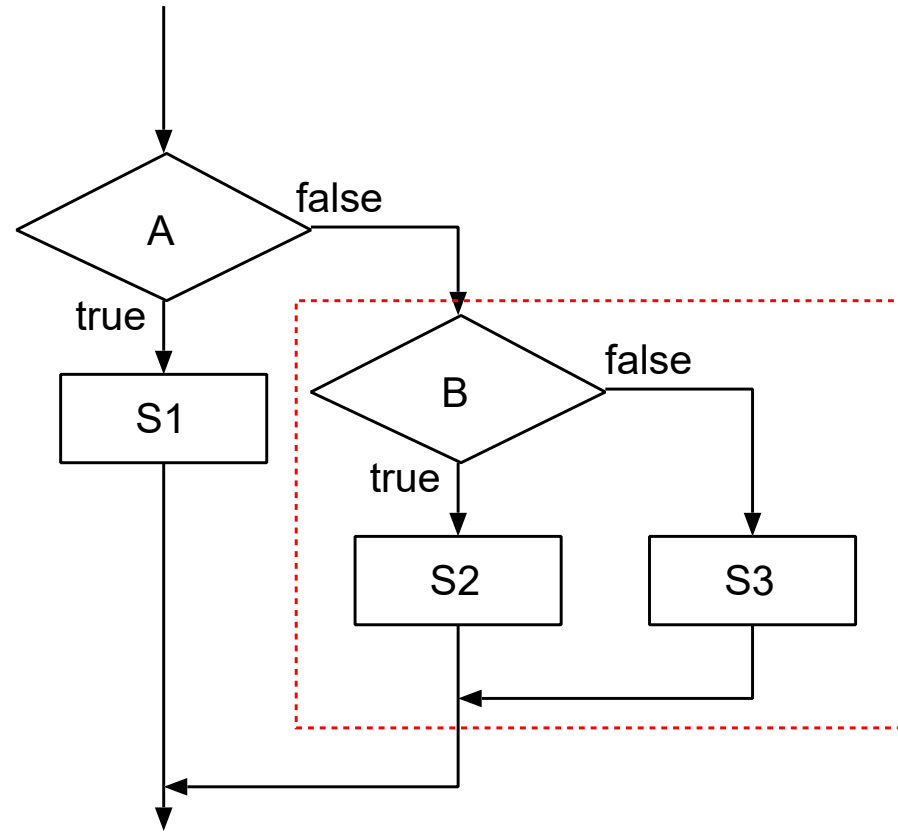
```
1  if (条件A && 条件B) {
2      文S1
3  }
```



else部のif-else文

- インデントが深くなる

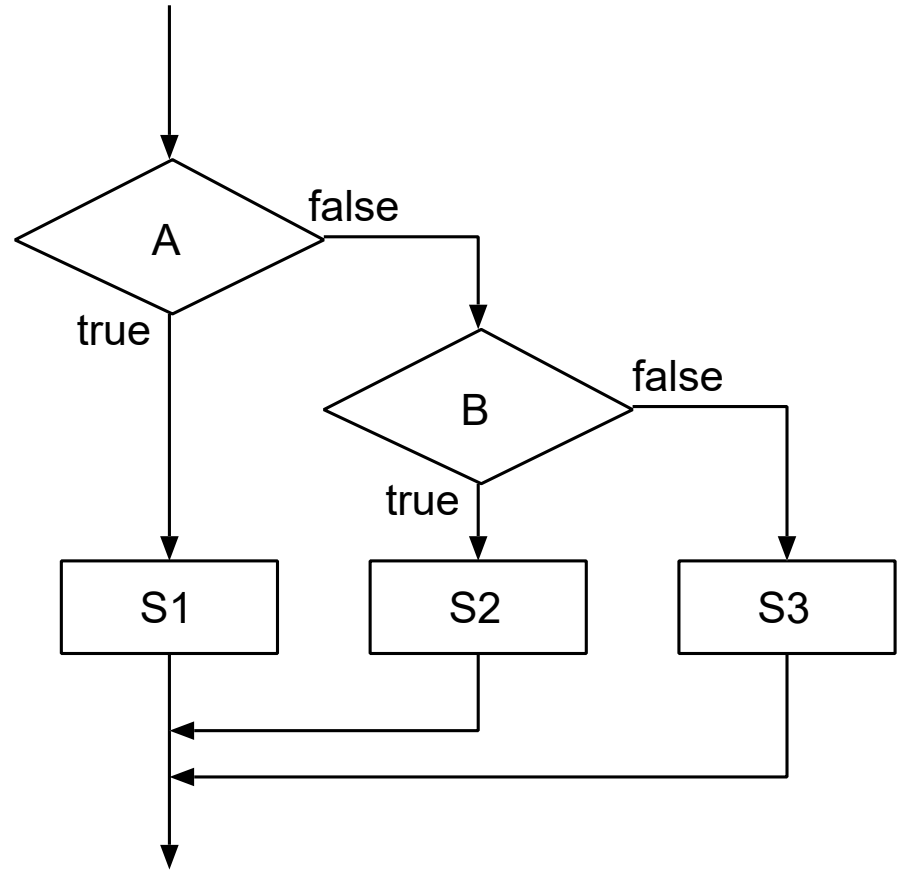
```
1  if (条件A) {
2    文S1
3  } else {
4    if (条件B) {
5        文S2
6    } else {
7        文S3
8    }
9  }
```



3個以上の実行文の選択

- 条件に応じた3個以上の実行文の選択
- `else`部の中括弧を書かずに`if-else`を続ける
- `else-if`文と呼ぶ人もいる
  - 非公式な呼び名なので注意する

```
1  if (条件A) {
2      文S1
3  } else if (条件B) {
4      文S2
5  } else {
6      文S3
7  }
```



その他

bool型の定数比較

- `bool`型の変数を`true` / `false`と比較しない
- 変数そのものが`true` / `false`の値である

```cpp
#include <iostream>

int main(){
  bool x{true};

  if (x == true) // 冗長
    std::cout << "redundant";

  if (x)         // 簡潔
    std::cout << "simple";

  if (!x)        // x == false の時の，簡潔な表現
    std::cout << "x is false";
}
```

ぶら下がりelse問題

- `else`部はもっとも近い`if`に対応する
  - インデントで混乱しないように気をつける
- 不安ならば中括弧を使い複文とする（ただし行数が増えることがある）.

```
1  int main(){ // 下の (a)と(b)は同じ処理？違う処理？
2    // (a) --------------------------------
3    if (x <= 10)
4      if (x == 10) std::cout << "Equal to 10";
5    else
6      std::cout << "Larger than 10?";
7
8    // (b) --------------------------------
9    if (x <= 10) {
10     if (x == 10) std::cout << "Equal to 10\n";
11   } else
12     std::cout << "Larger than 10\n";
13 }
```

三項演算子

```
条件A ? 式1 : 式2
```

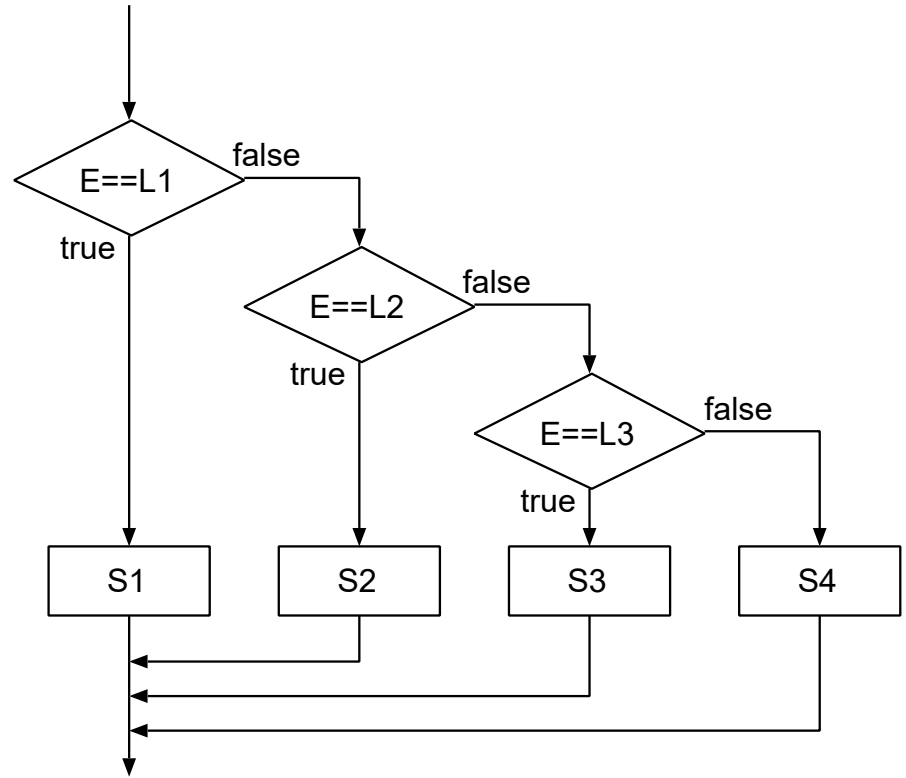
- ※ `?` と `:` が複合して登場したら三項演算子
- 条件に応じて二択の値を選ぶ
  - 条件Aが `true` → 式1
  - 条件Aが `false` → 式2
- 式1と式2の結果は同じ型

```
1  int main() {
2    int a = {10}; // 適当な値で初期化しておく
3
4    // 三項演算子で初期化
5    int x { (a < 0) ? (-a * 10) : (a * 20) };
6
7    // ↑と↓の三行は同じ処理
8    // int x;
9    // if ( a < 0 ) x = (-a * 10);
10   // else         x = (a * 20);
11
12   // 代入に使ってもOK
13   x = (a < 0) ? (-a * 10) : (a * 20);
14
15   // 3個以上の選択肢にも有用
16   char ch;
17   ch = (a == 1) ? 'a' : (a == 2) ? 'b' : (a == 3) ? 'c' : (a == 4) ? 'd' : 'z';
18
19   // 改行でわかりやすく
20   ch = (a == 1) ? 'a':
21        (a == 2) ? 'b':
22        (a == 3) ? 'c':
23        (a == 4) ? 'd': 'z';
24 }
```

switch文

- 整数値に応じて処理を選ぶ

```
switch (条件式E) {
    case ラベルL1:
        文S1;
        break;
    case ラベルL2:
        文S2;
        break;
    case ラベルL3:
        文S3;
        break;
    default:
        文S4;
}
```



switch文の例

- `case` ラベルにはコンパイル時の定数を使う
- `default` ラベルは他のラベルにマッチしない場合
- `break` を忘れないようにする
  - 忘れると**フォールスルー**（次の `case` ブロックが実行される）が起きる

```cpp
#include <iostream>
int main() {
  char ch;
  std::cin >> ch;

  switch (ch) {
    case 'a':
      std::cout << "apple\n";
      break;
    case 'b':
      std::cout << "banana\n";
      break;
    case 'c':
      std::cout << "candy\n";
      break;
    default:
      std::cout << "unknown\n";
  }
}
```

```
% ./a.out
a
apple
% ./a.out
z
unknown
```

フォールスルー

- `break` の有無で，実行結果が変わる
- 意図してフォールスルーさせるときもある

```cpp
#include <iostream>
int main() {
  ch = 'a';

  switch (ch) {
    case 'a':
      std::cout << "apple "; break;
    case 'b':
      std::cout << "banana "; break;
    case 'c':
      std::cout << "candy "; break;
    default:
      std::cout << "unknown ";
  }
}
```

```
% ./a.out
apple
```

```cpp
#include <iostream>
  int main() {
  ch = 'a';

  switch (ch) {
    case 'a':
      std::cout << "apple ";
    case 'b':
      std::cout << "banana ";
    case 'c':
      std::cout << "candy "; break;
    default:
      std::cout << "unknown ";
  }
}
```

```
% ./a.out
apple banana candy
```

`break` をあえて書かない例

- 条件となる値を並べるだけ
- 長い変数名も一度だけの指定で良い
- `break` 文と `case` ラベルの関係
  - `switch` 文を終わらせる役割
  - `case` ラベルは実行文を開始する場所を示すだけ

```cpp
#include <iostream>
int main() {
  int longnamevariable {};
  std::cin >> longnamevariable;

  switch (longnamevariable) {
    case 1: case 2: case 3:
    case 5: case 7: case 11:
      std::cout << " あたりです\n";
      break;
    default:
      std::cout << " はずれです\n";
  }
}
```

switch文の`default`

- `default`ラベルが最後ならば`break`は不要
- 逆に、`break`を付けるならば途中に指定しても良い

```cpp
#include <iostream>
int main() {
  int n {};
  std::cin >> n;

  switch (n) {
    case  1:
      std::cout << "1を検出\n"; break;
    default:
      std::cout << " 不正な値\n"; break;
    case -1:
      std::cout << "-1を検出\n"; break;
  }
}
```

switch文と変数宣言

- `case` ラベルの後で変数宣言が必要ならば複文とする

```
#include <iostream>
int main() {
  switch (a+b) {
    case 3:
    {  // 複文にする
      int x {};  // 新たな変数宣言
      ....       // xを使った処理
      break;
    }
    case 4:
      ...
    ...
  }
}
```

インデント

- インデント：行頭の連続するスペース（or タブ文字`\t`）で字下げすること
  - インデントは制御に影響しない．読みやすいコードは正しいインデントから．

```cpp
#include <iostream>

int main() {
  std::cout << "Hello\n";

  int x{5};
  if( x >= 0 ) {
    std::cout << "x is positive value\n";
  }

  std::cout << "if statement is finished\n";

  if( x >= 100 )
    std::cout << "x is large value\n";
}
```

- ↑良い例

```cpp
#include <iostream>

int main() {
std::cout << "Hello\n";

int x{5};
if( x >= 0 ) {
std::cout << "x is positive value\n";
}

std::cout << "if statement is finished\n";

if( x >= 100 )
std::cout << "x is large value\n";}
```

- ↑悪い例

よくある間違い

- `if` 文をセミコロン `;` で閉じてしまう

```
1  int main() {
2    int x{10};
3    if ( x >= 0 ); //間違い！
4      std::cout << "x is positive";
5  }
```

- 条件を満たすとき，何もしない処理をする
- この例の場合，`std::cout` は必ず実行される
  - インデントによる混乱

- `else` に条件を持たせてしまう
  - 条件を持つのは，`else if (...)`

```
1  int main() {
2    int x {0};
3
4    if ( x == 0 )
5      std::cout << "x is zero.";
6    else ( x != 0 ) //間違い！
7      std::cout << "x is not zero.";
8  }
```

- コンパイルエラーなので，間違いに気付けるはず