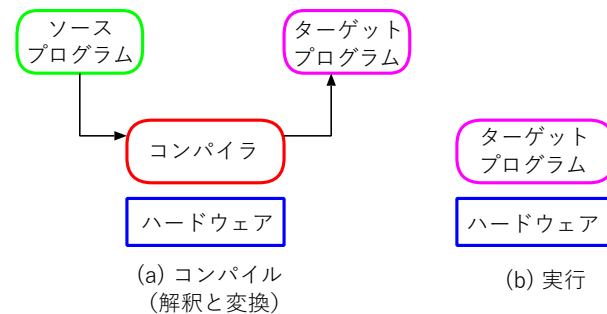


C++プログラミングI

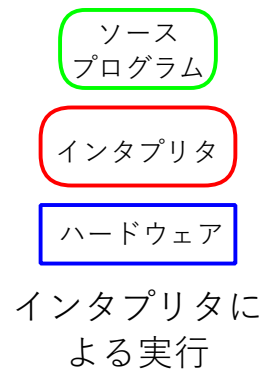
- 第1回：基本データ型と入出力
- 担当：二瓶芙巳雄

コンパイラとインタプリタ

- コンパイラ:
 - プログラムを結果が同一な別プログラムに変換する
 - 機械語プログラムに変換すればハードウェアで直接実行できる
- インタプリタ:
 - プログラムを解釈すると同時に実行も行う
 - 間接的な処理で余分な負荷が伴う



コンパイルと実行



インタプリタによる実行

様々なプログラミング言語

- C / C++ : コンパイラを想定
- Python : インタプリタを想定
- Java : コンパイラを想定, 実行環境がインタプリタ

言語	デスクトップ	Web	携帯機器	組み込み機器
C/C++	✓		✓	✓
Python	✓	✓		✓
Java	✓	✓	✓	

C++の標準規格

- C++98 : 情報科学科のこれまでの授業
- C++11
- C++14 : 最近のコンパイラのデフォルト
- C++17 : この授業の対象
- C++20 : 最新の規格

基本構造

```
1  // Hello! を出力するプログラム
2  #include <iostream>
3
4  int main() {
5      std::cout << "Hello\n";
6      return 0;
7  }
```

- コメント: 覚え書きを書く
- ヘッダファイル: 必要なライブラリに応じて追加
- `main()` (メイン関数): 主たる処理を指定する場所
 - `std::cout`: standard character output(標準の文字出力)
 - 行末のセミコロンので、処理を区切る
- ※このスライド上でのソースコードの見方について
 - わかりやすさのため、行頭に行番号を表示しています (ソースコードに行番号は入力しないこと)
 - わかりやすさのため、カラフルに着色しています (ソースコードに色が付くかどうかは環境次第)

コンパイルと実行の仕方

```
% g++ -std=c++17 hello.cpp # g++で実行ファイル a.out を作成
% ./a.out
Hello!
% g++ -std=c++17 hello.cpp -o hello # g++で実行ファイル hello を作成
% ./hello
Hello!
% g++ -std=c++17 hello.cpp -o hello.out # 拡張子が付いていてもOK
% ./hello.out
Hello!
```

- `g++` : ソースファイルから実行ファイルへの変換を行うプログラム
 - `-std=c++17` で `g++` が `C++17` の規格に沿った動作. `a.out` デフォルトの実行ファイルの名前.
- `hello.cpp` : コンパイル対象のソースファイル (※1ページ前のプログラムが `hello.cpp`)
- `./a.out` 現在のディレクトリにあるa.outを指定
- 行頭の `%` (あるいは `$`, `#`) : プロンプト (prompt: 促す). 「コマンドを入力できる状態です」の目印
 - 実験においては, プロンプトより後ろを入力すればOK

変数宣言・初期化・値の変更

- 変数とは値を入れる容器（型が決まっている）
- 変数は使う前には宣言が必要（型と名前を事前に指定）
- 初期値は省略できるがとても重要
- 値を変更するには「代入」または「入力」

```
1  #include <iostream>
2
3  int main() {
4      int x {1};      // 変数宣言と初期値指定
5      int y {x + 3};  // 他変数による初期化
6      int z {};       // 0で初期化（省略形）
7
8      x = 2;          // 値の代入
9      y = x;          // 変数値の代入
10     std::cin >> z;  // 入力
11     std::cout << z; // 出力
12 }
```

基本データ型

```
bool    // 論理値trueとfalse
char    // 英数記号の1文字(0から127の整数)
int     // 整数
double  // 実数(浮動小数点数)
```

- 基本データ型とはコンピュータの基本機能に結びつくデータ型
- 型は変数の種類を決めるので、用途別を選ぶ
- その他の型指定
 - `short int`, `long int`, `long long int`
 - `unsinged char`, `unsinged int`, `unsinged long int` ...
 - `float`, `long double`

```
1  #include <iostream>
2
3  int main() {
4      int x {1};
5      double y {3.14};
6      // ↓特別な意図がなければやらない
7      // int z {3.14};
8
9      char c {'a'};
10
11     bool b1 {true};
12     bool b2 {false};
13 }
```


リテラル

boolリテラル: `true` `false`

文字リテラル: `'a'` `'Z'` `' '` `'$'` `'%'`

整数リテラル: `123` `1` `0` `0x15` `034`

浮動小数点数リテラル: `3.1415` `1.0` `0.0` `1.0e-5`

文字列リテラル: `"Hello!"`

- リテラル：文字どおりの
- データ型ごとの値を直接指定する方法
- エスケープシーケンス：制御文字などの特殊文字, 一部の記号用
 - `'\n'` (改行), `'\t'` (タブ文字), `'\0'` (ヌル文字), `'\''`, `'\"'`, `'\\'`
 - `\` は, windows環境では ¥ として表示される

- 文字列リテラルは末尾にヌル文字 `\0` が付加される

`"Hello!"`

H	e	l	l	o	!	\0
---	---	---	---	---	---	----

`""`

\0

pythonとの比較：プログラムの書き方

```
1  // cppのプログラム
2  #include <iostream>
3
4  int main() {
5      std::cout << "Hello\n";
6
7      int x {5};
8      double y {3.14};
9      std::cout << x << " " << y << "\n";
10
11     string s;
12     std::cout << "input text:";
13     std::cin >> s;
14     std::cout << "\n";
15
16     std::cout << "text is: " << s << "\n";
17
18     return 0;
19 }
```

- ※ `\n` の `\` は ¥ と同じ

```
1  # pythonのプログラム
2
3
4
5  print( "Hello" )
6
7  x = 5
8  y = 3.14
9  print( x, " ", y )
10
11
12
13
14  s = input( "input text:" )
15
16  print( "text is: ", s )
```

- 行番号で対応を取っています
- どのように違うのか観察してみましょう

pythonとの比較：実行方法

- 1ページ前のソースコードの名前がそれぞれ, `hello.cpp`, `hello.py` の場合

```
1  % # cppで書かれたプログラムの実行方法
2  % g++ -std=c++17 hello.cpp -o a.out # コンパイル
3  % ./a.out                          # 実行
4  Hello!
5  5 3.14
6  input text: cpp # cpp の文字は手入力したもの
7  text is: cpp
8  %
```

```
1  % # pythonで書かれたプログラムの実行方法
2  %
3  % python hello.py # 実行
4  Hello!
5  5 3.14
6  input text: python # python の文字は手入力したもの
7  text is: python
8  %
```

- 行番号で対応を取っています
- どのように違うのか観察してみましょう

n進数の整数リテラル

```
1234          // 10進数
0x4d2         // 16進数
02322         // 8進数
0b010011010010 // 2進数, C++14
```

- 16進数は `0x` または `0X` で始まる数字 (hexadecimal)
 - a~fまたはA~Fを10~15として使用
- 8進数は `0` で始まる数字(octal)
- 2進数は `0b` または `0B` で始まる数字 (binary)
- 位取りに単一引用符を用いてよい
 - `1'234'567`

```
1  #include <iostream>
2
3  int main() {
4      int hex {0x4d2};
5      int oct {02322};
6      int bin {0b010011010010};
7      int large_val {1'000'000};
8
9      cout << "hex: " << hex
10         << ", oct: " << oct
11         << ", bin: " << bin
12         << ", large_val: " << large_val;
13 }
```

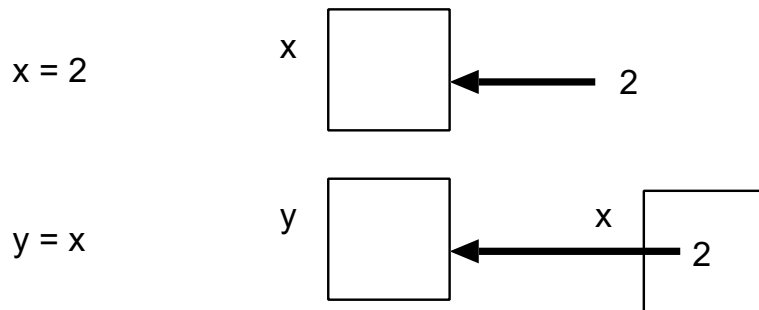
```
% ./a.out
hex: 1234, oct: 1234, bin: 1234, large_val: 1000000
```

※ `a.out` についてはp6参照

代入

- 変数の値を変更する
 - `=` だが「等しい」という意味ではない

```
1  int main() {  
2      int x {1};      // 変数宣言と初期値指定  
3      int y {x + 3};  // 他変数による初期化  
4      int z {};       // 0で初期化（省略形）  
5  
6      x = 2;          // 値の代入  
7      y = x;          // 変数値の代入  
8  }
```

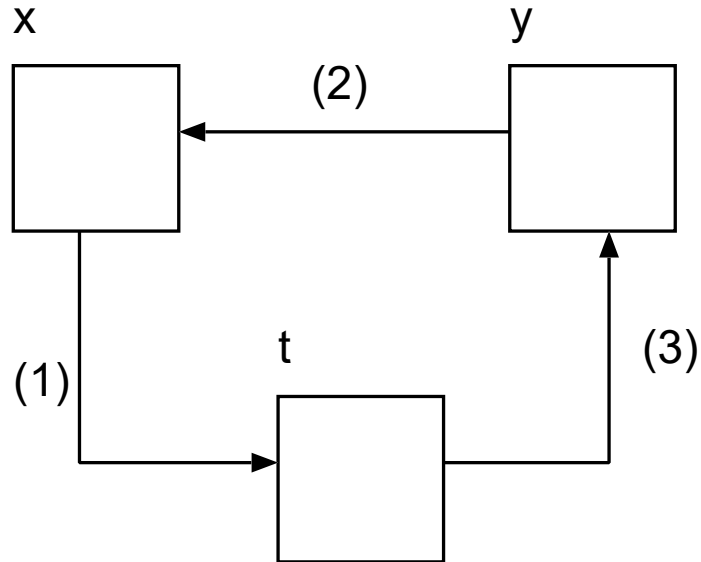


- lvalue: 左辺値、location（場所） value
 - 変数は左右どちらも指定可
- rvalue: 右辺値、read（値の読み出し） value
 - リテラルは右辺値
- `=` 演算子は右結合
 - `a = b = 1` → `a = (b = 1)`

値の入れ替え

```
1  int main() {  
2      int t {x};  
3      x = y;  
4      y = t;  
5  }
```

- 入れ替えには第三の変数が必要
 - 変数 **t** (temporal: 一時的な) が慣例的に使われる
- 代入の順序を確認する



変更できない変数

- 変数の中には値を変えない方が良くものもある
- `const` を付けて変更を防ぐ
 - `constant`の略語
 - `const int x {0};`
 - `const double pi {3.141592};`
- 宣言時の初期値が必須
- `const` 変数に代入しようとするとコンパイルエラー
- `const` 指定はプログラムの結果に影響を与えないが、実行を速くする場合も

```
1  #include <iostream>
2
3  int main() {
4      const int total_count {349};
5      std::cout << total_count << "\n";
6
7      total_count = 10; // エラー！
8      return 0;
9  }
```

```
% g++ -std=c++17 test.cpp
test.cpp: In function 'int main()':
test.cpp:7:15: error: assignment of read-only
variable 'total_count'
    7 |     total_count = 10;
      |
```

cin / cout の変換処理

```
1  #include <iostream>
2  int main() {
3      int x{};
4      std::cin >> x;    // 10進数文字列から変換
5      std::cout << "x = " << x << ", "; // 10進数
6      std::cout << std::hex << x << ", "; // 16進数
7      std::cout << std::oct << x << ", "; // 8進数
8      std::cout << std::dec << x << "\n"; // 10進数
9  }
```

- **cin** : 文字列を値に変換して変数に設定. **cout** : 値を文字列に変換して出力 (画面に表示).
- ※変数が保存している値が変わるのではなく, 画面に表示した際の見え方が変わるだけ

```
% ./a.out
31
x = 31, 1f, 37, 31
```


算術演算子

- 単項演算子と二項演算子
 - `+3`, `-3` は単項演算
 - `1+3`, `1-3` は二項演算
- これらの二項演算子は左結合
 - `a+b+c` → `(a+b)+c`
- 演算子の優先度
 - `x = a + b * c; // * + =` の順に優先
- 累乗の演算子はない

```
1  int main() {  
2      int x;  
3      x = -3;      // x は -3  
4      x = 1 - 3;   // x は -2  
5      x = 1 - - 3; // x は 4  
6  }
```

演算子 操作

`+` 加算

`-` 減算

`*` 乗算

`/` 除算 (整数どうしの場合は切り捨て)

`%` 剰余算 (割ったあまり)

int と double の組み合わせ

```
1  int main() {  
2      int      x{10}, y{3};  
3      double dx{x}, dy{y};  
4      std::cout << x/y << " " << dx/y << " "      // 3, 3.3333  
5                  << x/dy << " " << dx/dy << "\n"; // 3.3333, 3.3333  
6  }
```

- int どうしの割り算は切り捨て
- double が混ざればすべて double として計算
- int 変数を double に変換しても問題は少ない
 - 有効桁数次第
 - 暗黙の変換をさせない方が良い
 - 明示的な変換: `double dx{ static_cast<double>(x) };`
 - 問題が起きた場合に見つけやすい

代入演算子

- 変数の更新時に変数名を二回書かなくてもよい

```
1  #include <iostream>
2  int main() {
3      int point{3}, x{2}, y{5};
4
5      point *= 2;    // point=point*2
6      x *= y + 1;    // y+1倍, x=x*(y+1)
7      std::cout << "point=" << point << "\n";
8      std::cout << "x=" << x << "\n";
9  }
```

```
% ./a.out
point=6
x=12
```

演算子 操作

`+=` 加算代入

`-=` 減算代入

`*=` 乗算代入

`/=` 除算代入

`%=` 剰余算代入

増分 / 減分演算子

```
1  #include <iostream>
2  int main() {
3      int a{}, b{};
4      a++;    // a += 1
5      b--;    // b -= 1
6
7      int x{}, y{}, z{};
8      z = x++;    // z=x; x+=1;
9      z = ++y;    // y+=1; z=y;
10
11     int v1{5}, v2{10};
12     std::cout << v1 << " " << v1++ << " " << v1 << "\n"; // 画面に表示される値は？
13     std::cout << v2 << " " << --v2 << " " << v2 << "\n"; // 画面に表示される値は？
14 }
```

- `+1` や `-1` を目立たせる
- 前置きと後ろ置きに注意

数学用の定数と関数

良く使われる定数	意味	値
<code>M_E</code>	ネイピア数 e	2.7182818284590452354
<code>M_PI</code>	円周率 π	3.14159265358979323846
<code>M_SQRT2</code>	$\sqrt{2}$	1.41421356237309504880

■ `<cmath>` ヘッダファイルをインクルード

```
1  #include <iostream>
2  #include <cmath>
3
4  int main() {
5      std::cout << M_PI << ", "
6              << std::sin( 0 ) << ", "
7              << std::sin( M_PI/2 );
8      // 3.14159, 0, 0.707107
9  }
```

良く使われる関数

意味

`std::abs(arg)`

`arg` の絶対値を返す

`std::ceil(arg)`

天井関数： `arg` よりも小さくなくもっとも近い整数を返す

`std::floor(arg)`

床関数： `arg` よりも大きくなくもっとも近い整数を返す

`std::round(arg)`

`arg` にもっとも近い整数を返す

`std::exp(arg)`

自然対数の底 e の `arg` 乗(e^{arg})を返す

`std::log(arg)`

`arg` の自然対数を返す

`std::pow(arg1, arg2)`

`arg1` の `arg2` 乗($arg1^{arg2}$)を返す

`std::sqrt(arg)`

`arg` の平方根のうち負でない方を返す

`std::sin(arg)`

`arg` (ラジアン) の正弦 (サイン) の値を返す

`std::cos(arg)`

`arg` (ラジアン) の余弦 (コサイン) の値を返す

using宣言

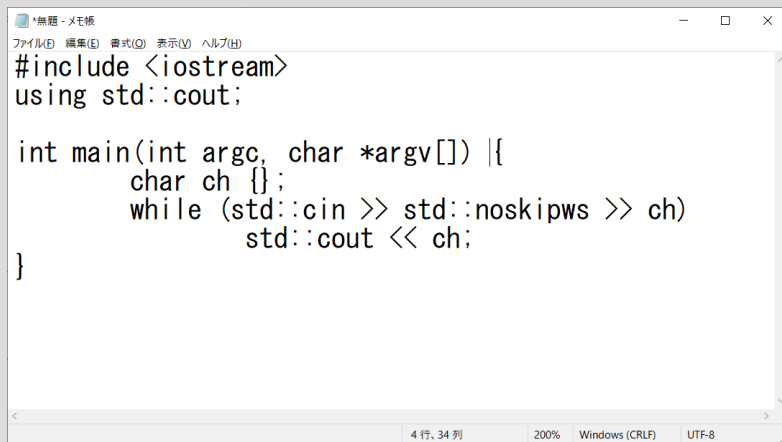
- `std::` を省略して指定する方法

```
1  #include <iostream>
2  using std::cin;  // using 宣言
3  using std::cout; // using 宣言
4
5  int main() {
6      int x{0}, y{0};
7      cout << "input: ";
8      cin >> x >> y;
9      cout << x << "*" << y << " = " << x*y << "\n";
10     return 0;
11 }
```

```
% ./a.out
input: 10 20
10*20 = 200
```

余談：テキストエディタ

- テキストエディタによって、制御文字が見える/見えない、シンタックスハイライトあり/なし がある
 - 制御文字が見えるほうが、プログラミング的には便利
 - ※好きなテキストエディタを使ってもらってOKです
- メモ帳

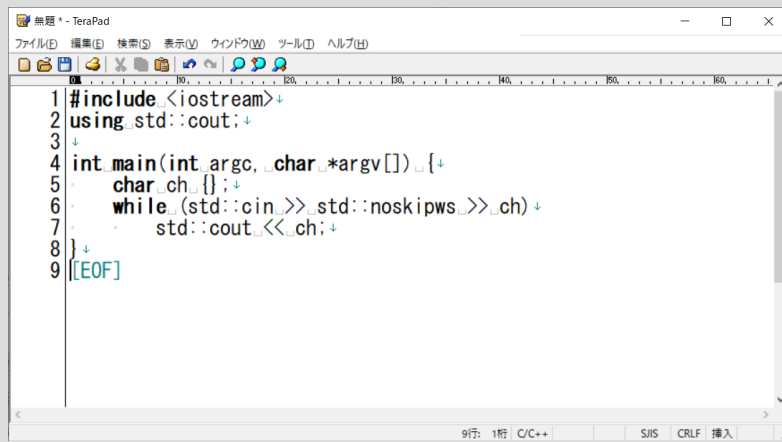


```
#include <iostream>
using std::cout;

int main(int argc, char *argv[]) {
    char ch {};
    while (std::cin >> std::noskipws >> ch)
        std::cout << ch;
}
```

- メモ帳は早めに卒業するとよいかもしれません

- terpad (この授業で推奨)



```
1 #include <iostream>+
2 using std::cout;+
3 ↓
4 int main(int argc, char *argv[]) {+
5     char ch {} ;+
6     while (std::cin >> std::noskipws >> ch) +
7         std::cout << ch;+
8 }+
9 [EOF]
```

- ↓: 改行 (\n), > : タブ文字 (\t), □: 半角スペース
- ホワイトスペースが別の記号として表現される