AI時代のプログラミング

C/C++の歴史・現状・将来展望

C（1970年代・Bell Labs）

- システムプログラミング向けの移植性の高い高級言語として設計され、「ハードウェアに近い制御性」と「読みやすさ・保守性」のバランスを実現した。
- 構造化プログラミングを普及させた：
  - if/else、for/while、switch
  - 関数によるモジュール化
  - struct によるデータ構造の表現
- ポインタと明示的なメモリ管理により低レイヤの操作が可能
  - **高性能・高い制御性を得られる**
  - **資源解放や境界チェックは基本的にプログラマの責任となる**

アセンブリ言語
(MIPS)

```
swap:
        multi $2, $5,4
        add   $2, $4,$2
        lw    $15, 0($2)
        lw    $16, 4($2)
        sw    $16, 0($2)
        sw    $15, 4($2)
        jr    $31
```

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

C言語

C/C++の歴史・現状・将来展望

C++（1980年代・Bjarne Stroustrup）

- 「高性能」と「抽象化」を両立できる言語
- オブジェクト指向：クラス、カプセル化、継承、多態性。
- RAII（Resource Acquisition Is Initialization）：オブジェクトの寿命に資源管理（メモリ／ファイル／ロック等）を結び付け、例外発生時でも安全に後始末できる設計を促進
- ジェネリックプログラミング：テンプレート（templates）を導入し、STLによる「組み合わせ可能」な汎用ライブラリ生態系を形成
- 例外処理：try/catch とスタック巻き戻しが RAII と連携
- 演算子オーバーロード／名前空間／関数オーバーロード：表現力とライブラリ設計の自由度を向上

C/C++の歴史・現状・将来展望

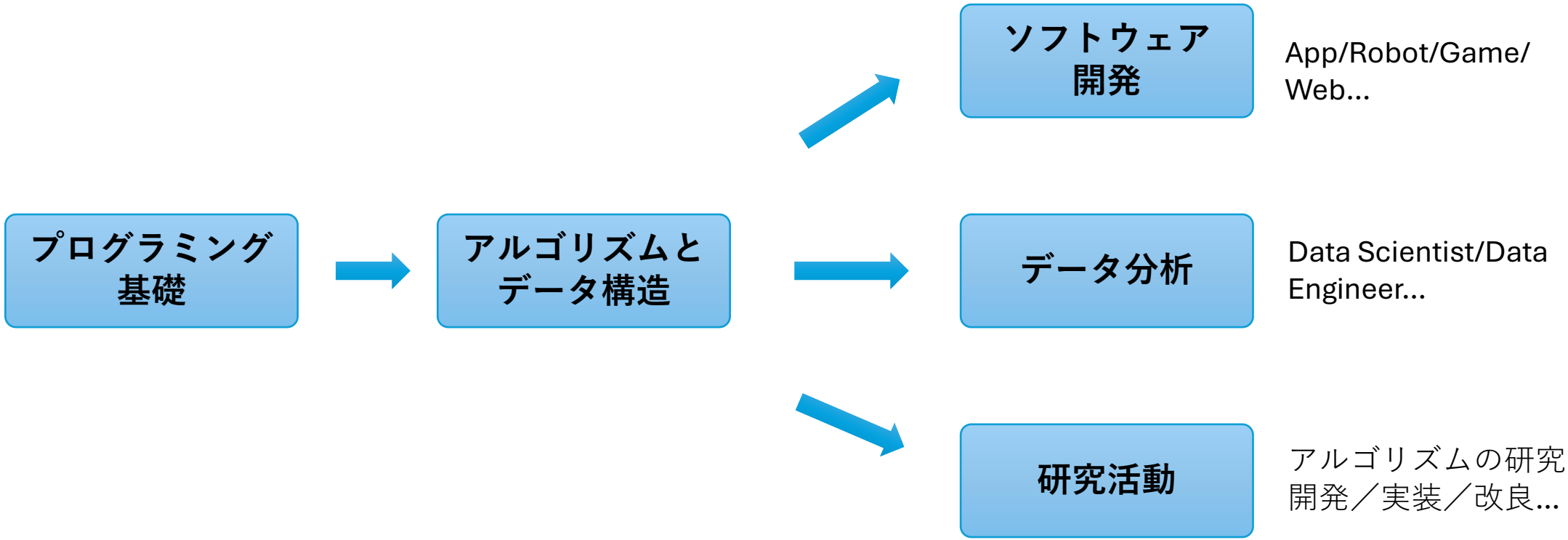
C++言語の課題

- メモリ管理の高リスク
  - メモリ管理が不適切だと、S0（Blocker）やS1（Critical）といった高い重大度の不具合を引き起こしやすい。

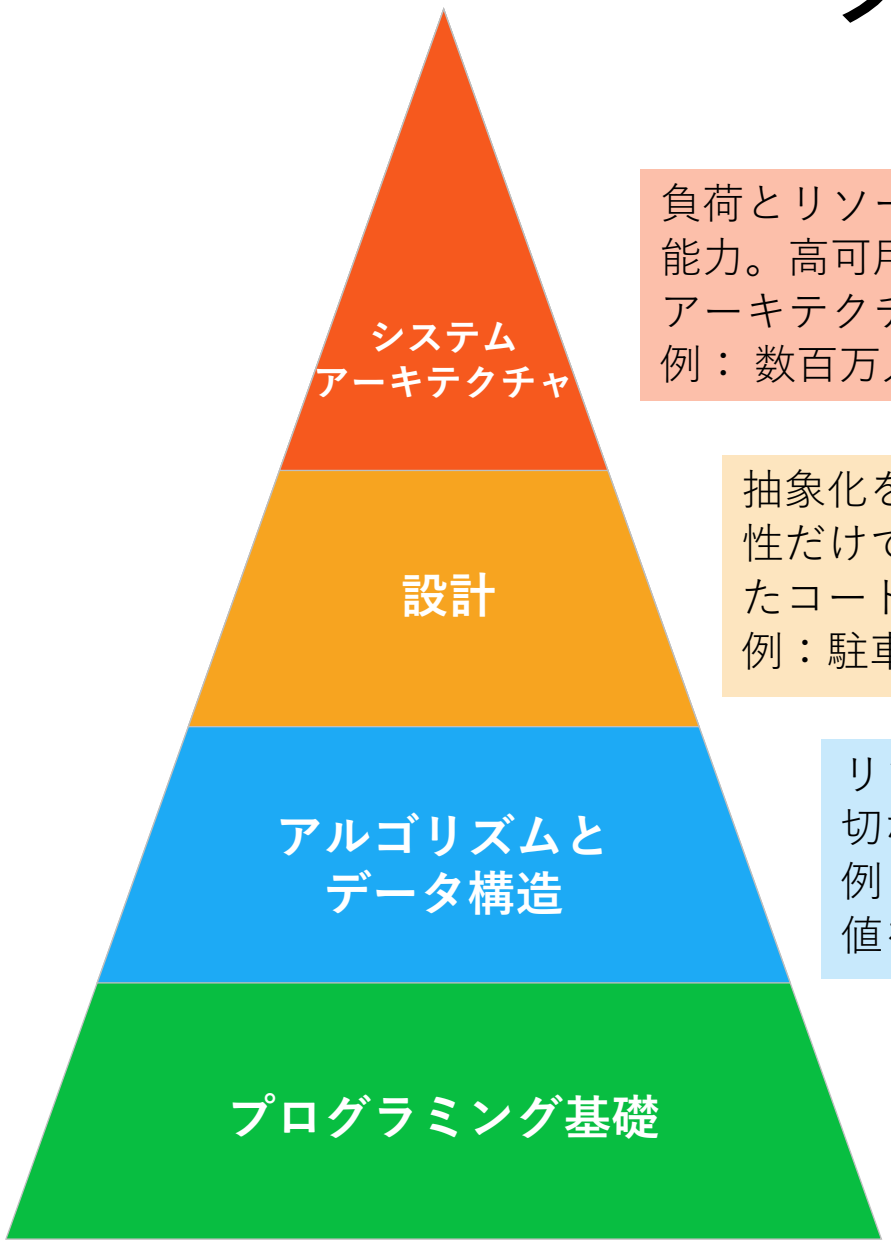
- **CISA（サイバーセキュリティ・社会基盤安全保障庁）とFBI**が、C/C++による新規開発の制限とメモリ安全な言語への移行を要請（2025年2月頃）。
- **ホワイトハウス（ONCD）**も、開発者に対してC/C++からRust、Java、C#などへの移行を強く推奨（2024年2月）。

- 開発効率（生産性）が下がりやすい
  - 実行時の性能は強い一方で、実装〜検証のループが長くなりやすい
  - 高い複雑性：C++は言語仕様や書き方の選択肢が多く、さらにテンプレートやオーバーロード解決（暗黙変換など）の要素が重なることで、コードの挙動が追いにくくなり、可読性・理解容易性が下がりやすい。
- 規格の継続的進化による学習・適用コスト
  - C++11以降、規格は概ね3年周期で更新され、言語仕様や推奨される書き方も継続的に変化してきました（C++11/14/17/20/23）。
  - その結果、同じ目的を達成する方法が複数存在しやすく（新旧の書き方が併存しやすい）、コードベース内でスタイルが混在すると、レビュー基準の統一や保守、チーム開発における合意形成のコストが増大します。

プログラミング基礎の次は？



ソフトウェア開発の層級



負荷とリソースを分析し、全体のシステム構造を設計する能力。高可用性・スケーラビリティ・耐障害性を考慮したアーキテクチャを構築する。
例： 数百万人規模のチケットシステム設計。

「大規模で信頼性の高いシステムを設計する」

抽象化を行い、インターフェースを設計する能力。機能性だけでなく、拡張性・保守性・テスト容易性を考慮したコードを書く。
例：駐車場システムをオブジェクト指向で設計せよ。

「きれいで長持ちするコードを書く」

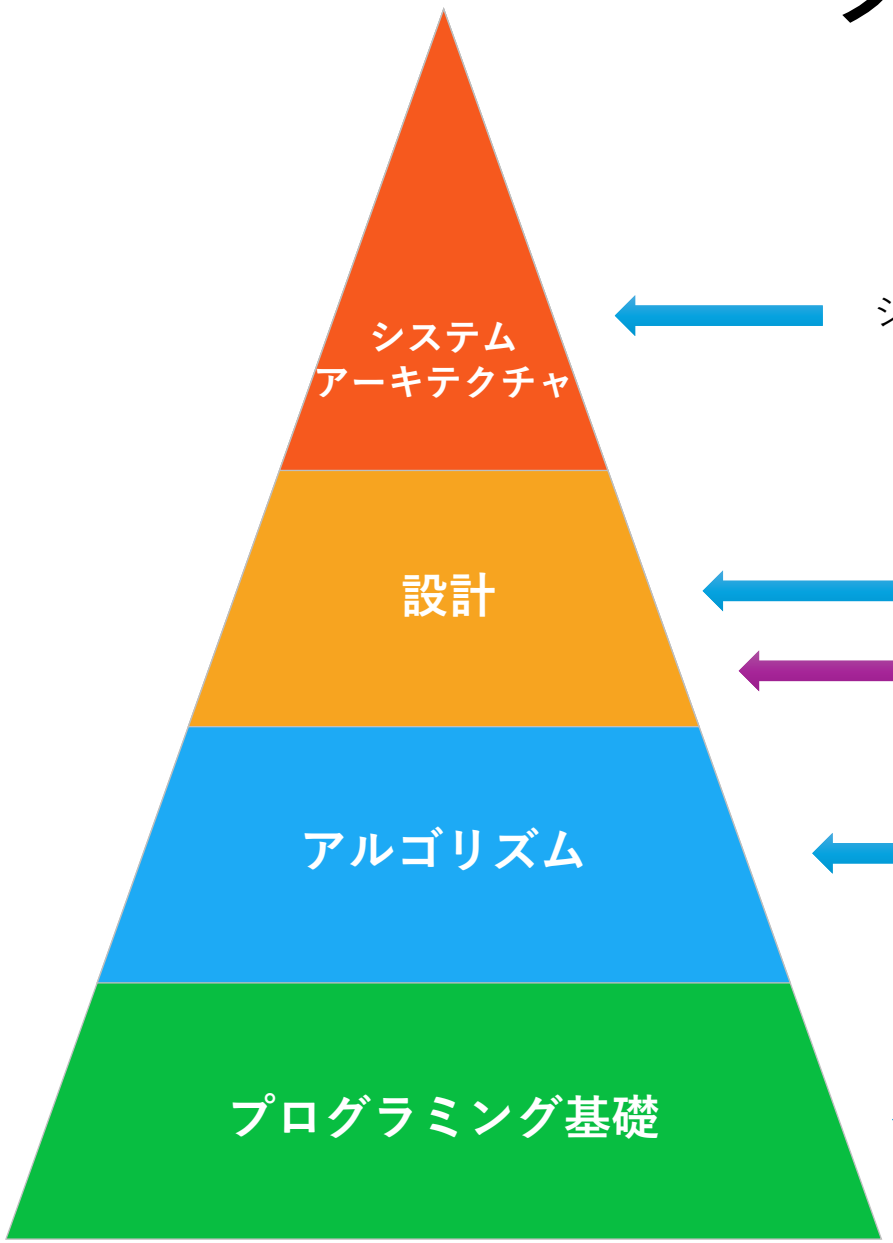
リソースの消費を意識し、計算量を分析できる能力。適切なデータ構造とアルゴリズムを選択する。
例：2つのソート済み配列が与えられたとき、全体の中央値を O(log(m+n)) で求める。

「効率的に書く」

文法・デバッグ・プロジェクト構成
例：文字列から重複する文字をすべて削除し、ユニークな文字だけ残す（順序は保持）

「コードを書いて動かす」

ソフトウェア開発の層級



シニアエンジニア

ベテランエンジニア

AI

新人エンジニア

授業

AI?

「大規模で信頼性の高いシステムを設計する」

「きれいで長持ちするコードを書く」

「効率的に書く」

「コードを書いて動かす」

Vibe コーディング

- 生成AIと人間が対話しながら、自然言語での「雰囲気」や「意図」を伝えることで、直感的にソフトウェアを開発する手法です。
- ソフトウェア開発の過程では、低レベルで反復的な作業の多くをAIに任せることで、生産性を大幅に向上させることができる。
- AI支援のプログラミングツールは今や非常に多く、しかも急速に進化しています。状況に応じて柔軟に活用してください。

図１　従来のプログラミングvs.バイブコーディング（Vibe Coding）

https://arpable.com/artificial-intelligence/agent/ai-agent-economy-vibe-coding/

AI コーディング：機会と脅威

機会：生産性向上と新たな役割の創出

- 経験豊富な開発者はAIツールを活用して生産性を大幅向上。
- プログラミングに関する知識の習得が、非常に簡単かつ迅速になった。
- 2025年のStack Overflow調査では、開発者の84%がAIツールを日常的に使用。

脅威：エントリーレベルの雇用が急減

- US ソフトウェア開発者（22〜25歳）の雇用は、2022年10月のピークを基準に、2022年末から2025年7月にかけて**約20%減少している**。同業の26歳以上の層では、雇用は同様の減少は見られない。

AI時代のプログラミング学習のアドバイス

- AIをフル活用して、爆速で学習し、早く「経験者」になる。 AIの力を借りて自分の生産性を最大化すること。
- 実際の問題から始めて、問題解決力を磨く。文法だけ覚えても意味がないのはその通り。結局、プログラミングの本質は「現実の問題を解決する」ことです。
- いろんなアイデアをどんどん試して、すぐに実装する。AI時代は試行錯誤のコストが極端に低いです。思いついたらすぐにプロトタイプを作ってみる。

AIを「助手」にして、決して「思考の代行者」にしない