# 6-1. Data와 Code의 연계

창의적소프트웨어프로그래밍
2022년도 여름학기
Racin

# 오늘 내용

- Data와 Code의 연계
  - 프로그램을 구성하는 두 부분이 어떻게 서로 엮여 있는지 잠시 짚어 봅니다
    - 둘 중 한 쪽에 무게를 두고 다른 쪽이 어떤 느낌으로 옆에 붙게 되는지 생각해요
  - 이러한 관점에서, 여러 사람이 함께 프로그램을 만들 때 생길 수 있는 문제를 제기합니다
- Data & Code
  - 주요 Data와 이를 항상 유효하게 담아 두기 위한 Code를 한 데 담아 다루기
  - 주요 Code와 이를 목표에 맞게 **실행**하기 위한 Data를 한 데 담아 다루기
- 오늘은 처음으로 내가 직접 C++ 코드를 작성해 볼 예정이에요.
  - 맛만 볼 거임

# 이번 시간에는

- 예전에 나왔던 '프로그램 = Data + Code'를 살짝 들고 와서 새로 배운 내용들을 덧입혀 봅니다.
  - 역시나 짧은 복습이 될 것 같아요

- 바로 시작해 봅시다.

# Data와 Code

- 예전에 '프로그램 = Data + Code'라는 이야기를 들은 적이 있지요?

# Data와 Code

- 예전에 '프로그램 = Data + Code'라는 이야기를 들은 적이 있지요?

- 아직 이 둘의 경계선을 명확히 긋는 것은 쉽지 않을 거예요.
  - 명백히 Data라 말할 만한 'int **변수** number 자리에 담긴 **값**'이나
    명백히 Code라 말할 만한 break문 같은 것도 있지만
    '어떤 **함수 정의** 내용물에 대한 **위치 값**' 같이 모호한 친구도 존재한다 말할 수 있어요

# Data와 Code

- 예전에 '프로그램 = Data + Code'라는 이야기를 들은 적이 있지요?

- 아직 이 둘의 경계선을 명확히 긋는 것은 쉽지 않을 거예요.
  - 명백히 Data라 말할 만한 'int **변수** number 자리에 담긴 **값**'이나
    명백히 Code라 말할 만한 break문 같은 것도 있지만
    '어떤 **함수 정의** 내용물에 대한 **위치 값**' 같이 모호한 친구도 존재한다 말할 수 있어요

- 모호한 것이 당연해요. 컴퓨터는 오로지 숫자만 다룰 수 있기 때문에,
  이전에 디버그 모드에서 자주 구경했듯,
  Code들 또한 컴파일러에 의해 숫자가 된 채로 메모리에 오르기 때문이에요.
  - 그래서 하드웨어 분야에서는 종종 '**상수**면 Code'라고 치고 있기는 해요
    (**offset 상수 값**이나 '즉시**값**'같은 친구들은 대부분 명령어 안에 포함되어 있었음!)

# Data와 Code의 연계

- 뭐 그렇긴 하지만,
  Data와 Code가 서로 맞물려 프로그램 실행을 구성한다는 사실은 명백해요.
  - 이들 중 하나에 중점을 두었을 때, 다른 쪽에 해당하는 개념이 보조 역할을 수행하게 됨

# Data와 Code의 연계

- Data를 다루기 위한 Code

# Data와 Code의 연계

- Data를 다루기 위한 Code
  - 시험 등수 **값** 같은 것은 뭐 보편적인 Code만 가지고도 담거나 읽는 것이 가능해요
    - int **형식**의 표현 능력만으로도 순위 개념을 충분히 다룰 수 있어요
    - = **수식 계산** 한 번이면 내 시험 등수 **값**을 통째로 담을 수 있어요

  - 반면, Data가 복잡해질수록 이를 온건히 다루기 위해 더 복잡한 Code가 필요해요
    - 능력치 정보 1인분을 표현하려면 int 기준 여러 칸이 필요했어요.
      따라서 = **수식 계산** 한 번 만으로는 능력치 **값**을 통째로 담을 수 없었어요
      - 사실 **수식** 자체는 적을 수 있어요. 그렇기는 하지만 runtime에 mov를 여러 번 해야 하는 것은 맞아요

    - 반대로, '다음 랜덤 **값**'과 같이 **값**의 유효성 여부가 복잡하게 결정되는 경우에도
      적절한 Code(보통 rand()를 활용할 듯)의 **실행**이 필요해지게 될 거예요

# Data와 Code의 연계

- Code 실행에 필요한 Data

# Data와 Code의 연계

- Code 실행에 필요한 Data
  - 우리가 printf()를 호출하여 '검은 창에 출력' 목표에 접근하는 이유는 '검은 창'을 다루기 위해 반드시 필요한 Data를 우리는 모르기 때문이에요

# Data와 Code의 연계

- Code 실행에 필요한 Data
  - 우리가 printf()를 호출하여 '검은 창에 출력' 목표에 접근하는 이유는
    '검은 창'을 다루기 위해 반드시 필요한 Data를 우리는 모르기 때문이에요
    - 우리가 printf()를 직접 만드는 입장이라면, 우리는 그 Data를 반드시 알고 있어야 해요

  - rand() 또한, 매 번 호출받을 때마다 조건에 맞는 새 랜덤 **값**을 return해야 하므로,
    '무슨 **값**을 이번에 return해야 랜덤성이 유지되는지'에 대한 Data를 반드시 지녀야 해요

# Data와 Code의 연계

- 사실 '프로그램 = Data + Code'에서 가장 중요한 것은 '+'예요!
  - '칸의 일생' 파트에서도 보았듯,
    어떤 Code도 건드리지 않는 Data, 아무 Data도 건드리지 않는 Code는
    그 프로그램에서 의미를 가지지 않을 가능성이 높아요

  - 그러니,
    작은 **변수** 하나를 도입하더라도 '여기에 무엇을 언제 담고 언제 읽을 것인지'를,
    작은 **함수** 하나를 **정의**하더라도 '얘가 언제 호출되어 무엇을 받고 바꾸고 줄 것인지'를
    ...떠올려 보는 것은 나름 좋은 습관이 될 수 있을 거예요

# 문제점

- 여기서 문제가 생겨요.
  - 작은 프로그램 하나를 만든다 하더라도
    Data측면, Code측면을 생각하는 것도 모자라 둘의 연계점까지 고려해 주어야 해요

# 문제점

- 여기서 문제가 생겨요.
  - 작은 프로그램 하나를 만든다 하더라도
    Data측면, Code측면을 생각하는 것도 모자라 둘의 연계점까지 고려해 주어야 해요

  - 특히, 여러 사람이 하나의 프로그램을 같이 만들 때
    누가 필수 Code를, 필수 Data를 챙길 것인지에 대한 결정을 해 주어야 해요
    - 지난 시간 끝날 즈음 본 코드를 잠시 가져오면...

# 문제점

- 아래 코드를 구경해 봅시다:

```
struct Stat { int str, dex, con; };

struct Stat *New_Stat(
    int str, int dex, int con) {
    struct Stat *result
        = malloc(sizeof(struct Stat));
    result->str = str;
    result->dex = dex;
    result->con = con;
    return result;
}

void Delete_Stat(struct Stat *stat) {
    free(stat);
}
```

```
void func1()
{
    struct Stat *stat
        = malloc(sizeof(struct Stat));
    stat->str = 30;
    stat->dex = 50;
    stat->con = 20;
    free(stat);
}

void func2()
{
    struct Stat *stat = New_Stat(30, 50, 20);
    Delete_Stat(stat);
}
```

# 문제점

- 아래 코드를 구경해 봅시다:

```
struct Stat { int str, dex, con; };

struct Stat *New_Stat(
    int str, int dex, int con) {
    struct Stat *result
        = malloc(sizeof(struct Stat));
    result->str = str;
    result->dex = dex;
    result->con = con;
    return result;
}

void Delete_Stat(struct Stat *stat) {
    free(stat);
}
```

능력치 개념을 먼저 구성하는 사람이
적절한 Code를 미리 구성해 두면
나중에 능력치 개념을 활용하는 사람이
좀 더 간편하게 목표를 달성할 수 있어요.

```
void func2()
{
    struct Stat *stat = New_Stat(30, 50, 20);
    Delete_Stat(stat);
}
```

# 문제점

- 아래 코드를 구경해 봅시다:

```
struct Stat { int str, dex, con; };

struct Stat *New_Stat(
    int str, int dex, int con) {
    struct Stat *result
        = malloc(sizeof(struct Stat));
    result->str = str;
    result->dex = dex;
    result->con = con;
    return result;
}

void Delete_Stat(struct Stat *stat) {
    free(stat);
}
```

```
void func1()
{
    struct Stat *stat
        = malloc(sizeof(struct Stat));
    stat->str = 30;
    stat->dex = 50;
    stat->con = 20;
    free(stat);
}
```

그렇지 않은 케이스의 경우에는
이런 식으로
직접 적절한 Code를 적어 주어야 할 거예요.

# 문제점

- 아래 코드를 구경해 봅시다:

```
struct Stat { int str, dex, con; };

struct Stat *New_Stat(
    int str, int dex, int con) {
    struct Stat *result
        = malloc(sizeof(struct Stat));
    result->str = str;
    result->dex = dex;
    result->con = con;
    return result;
}

void Delete_Stat(struct Stat *stat) {
    free(stat);
}
```

```
void func1()
{
    struct Stat *stat
        = malloc(sizeof(struct Stat));
    stat->str = 30;
    stat->dex = 50;
    stat->con = 20;
    free(stat);
}
```

지금이야 뭐 그냥 값 세 개 담는 선에서 끝나지만,
'능력치 최대값 제한'이나 '능력치 합 제한'같은 조건이 붙는다면
실질적으로 요구되는 Code의 양이 매우 많아질 거예요.
'누가 적을 것이냐'가 나름 중요한 이슈가 됨!

# 문제점

- 일반적으로는, '이번 프로그램'을 위한 코드(예: `main()`)를 짜는 것은 '미래의 어떤 프로그램들'을 위한 코드(예: `printf()`)를 짜는 것에 비해 상대적으로 쉽다고 말할 수 있어요.

# 문제점

- 일반적으로는, '이번 프로그램'을 위한 코드(예: main())를 짜는 것은 '미래의 어떤 프로그램들'을 위한 코드(예: printf())를 짜는 것에 비해 상대적으로 쉽다고 말할 수 있어요.
  - 그러나, 프로그래밍의 보급에 따라,
    main() 짜는 사람과 printf() 짜는 사람의 실력차가 점점 심해지는 현상이 생겼어요

# 문제점

- 일반적으로는, '이번 프로그램'을 위한 코드(예: main())를 짜는 것은 '미래의 어떤 프로그램들'을 위한 코드(예: printf())를 짜는 것에 비해 상대적으로 쉽다고 말할 수 있어요.
  - 그러나, 프로그래밍의 보급에 따라,
    main() 짜는 사람과 printf() 짜는 사람의 실력차가 점점 심해지는 현상이 생겼어요
    - 70년대 이전까지는 (컴퓨터가 비싸기도 했고) 프로그래밍을 마음대로 할 수 있는 사람들은 대부분 컴퓨터 분야에 대한 사전 지식들을 상당히 많이 탑재한 상태였어요
    - 뭐 지금이야 '선코딩 후전공'이 매우 자연스럽게 느껴지지만, 80년대 시점에는 아직 '저 쉬운 main()도 오타를 내 놨네' 하면서 스트레스를 받는 라이브러리 제작자들이 이전에는 쉬쉬하던 새로운 프로그래밍 관점을 적극적으로 받아들이기 시작하게 되었어요

# 문제점

- 일반적으로는, '이번 프로그램'을 위한 코드(예: main())를 짜는 것은 '미래의 어떤 프로그램들'을 위한 코드(예: printf())를 짜는 것에 비해 상대적으로 쉽다고 말할 수 있어요.
  - 그러나, 프로그래밍의 보급에 따라, main() 짜는 사람과 printf() 짜는 사람의 실력차가 점점 심해지는 현상이 생겼어요
    - 70년대 이전까지는 (컴퓨터가 비싸기도 했고) 프로그래밍을 마음대로 할 수 있는 사람들은 대부분 컴퓨터 분야에 대한 사전 지식들을 상당히 많이 탑재한 상태였어요
    - 뭐 지금이야 '선코딩 후전공'이 매우 자연스럽게 느껴지지만, 80년대 시점에는 아직 '저 쉬운 main()도 오타를 내 놨네' 하면서 스트레스를 받는 라이브러리 제작자들이 이전에는 쉬쉬하던 새로운 프로그래밍 관점을 적극적으로 받아들이기 시작하게 되었어요
      - ➢ 먼저 코드를 작성하는 사람이 적극적으로 많은 것들을 미리 정해 두는 것을 추구하기 시작했어요

# 마무리

- 우리가 다룰 C++는, C의 많은 것들을 계승하고 있지만
  '여러 사람이 같이 코드를 작성하는 것'을 좀 더 중시하며
  '먼저 코드를 작성하는 프로그래머'가 보다 많은 자유도를 행사할 수 있도록
  몇 가지 핵심 관점들을 추가로 포함하고 있어요.
  - 사실 C도 자유로움 하면 어디 가서 빠지지 않는 프로그래밍 언어기는 해요.
    위 밑줄 친 설명은 사실,
    '나중에 코드를 작성하는 프로그래머'의 자유도를 깎는다고 보는게 더 정밀할지 몰라요

- 대충 짚어 보았으니
  잠시 쉬거나 쉬지 않고 바로 다음으로 넘어가 봅시다.