What is a “Better” Program?

C++ Object Oriented Programming

Pei-yih Ting

NTOU CS

軟體的特性

- 軟體之所謂軟...因為沒有“硬性”不可變、不可挑戰的規則

軟體的特性

- 軟體之所謂軟…因為沒有“硬性”不可變、不可挑戰的規則
  - 好處: 彈性很大, 山不轉路轉, 沒有標準答案, 正確運作就好…

軟體的特性

- 軟體之所謂軟...因為沒有“硬性”不可變、不可挑戰的規則
  - 好處: 彈性很大, 山不轉路轉, 沒有標準答案, 正確運作就好...
  - 壞處: 很多小問題合在一起不斷放大, 到處藏污納垢, 沒有標準答案, 不知道到底對了沒有

軟體的特性

- 軟體之所謂軟...因為沒有“硬性”不可變、不可挑戰的規則
  - 好處: 彈性很大, 山不轉路轉, 沒有標準答案, 正確運作就好...
  - 壞處: 很多小問題合在一起不斷放大, 到處藏污納垢, 沒有標準答案, 不知道到底對了沒有
- 解決方法

軟體的特性

- ✧ 軟體之所謂軟...因為沒有“硬性”不可變、不可挑戰的規則
  - ★ 好處: 彈性很大, 山不轉路轉, 沒有標準答案, 正確運作就好...
  - ★ 壞處: 很多小問題合在一起不斷放大, 到處藏污納垢, 沒有標準答案, 不知道到底對了沒有
- ✧ 解決方法
  - ★ **Coding styles**

軟體的特性

- 軟體之所謂軟...因為沒有“硬性”不可變、不可挑戰的規則
  - 好處: 彈性很大, 山不轉路轉, 沒有標準答案, 正確運作就好...
  - 壞處: 很多小問題合在一起不斷放大, 到處藏污納垢, 沒有標準答案, 不知道到底對了沒有
- 解決方法
  - **Coding styles**
  - test-driven

軟體的特性

- 軟體之所謂軟...因為沒有“硬性”不可變、不可挑戰的規則
  - 好處: 彈性很大, 山不轉路轉, 沒有標準答案, 正確運作就好...
  - 壞處: 很多小問題合在一起不斷放大, 到處藏污納垢, 沒有標準答案, 不知道到底對了沒有
- 解決方法
  - **Coding styles**
  - test-driven
  - 元件化

軟體的特性

- 軟體之所謂軟…因為沒有“硬性”不可變、不可挑戰的規則
  - 好處: 彈性很大, 山不轉路轉, 沒有標準答案, 正確運作就好…
  - 壞處: 很多小問題合在一起不斷放大, 到處藏污納垢, 沒有標準答案, 不知道到底對了沒有
- 解決方法
  - **Coding styles**
  - test-driven
  - 元件化

    模型化 (資料結構, 演算法, 物件化, 設計樣版)

Source Code is the Primary Document

- Jack Reeves, C++ Journal, 1992, “What is Software Design?”

Source Code is the Primary Document

- Jack Reeves, C++ Journal, 1992, “What is Software Design?”

  “After reviewing the software development life cycle as I understood it, I concluded that the on **documentation** that actually seems to satisfy the criteria of an engineering design is the **source co**

”.sgnitsil ed

Source Code is the Primary Document

- Jack Reeves, C++ Journal, 1992, "What is Software Design?"

  "After reviewing the software development life cycle as I understood it, I concluded that the on **documentation** that actually seems to satisfy the criteria of an engineering design is the **source co**

- The **design of a software project** is an abstract concept:

Source Code is the Primary Document

- Jack Reeves, C++ Journal, 1992, "What is Software Design?"

  "After reviewing the software development life cycle as
  I understood it, I concluded that the on
  documentation that actually seems to satisfy the criteria
  of an engineering design is the **source co**

- The **design of a software project** is an abstract concept:
  - It has to do with the overall shape and structure of the program as well as the detailed shape and structure of each module, class, and method.

Source Code is the Primary Document

- Jack Reeves, C++ Journal, 1992, "What is Software Design?"

  "After reviewing the software development life cycle as I understood it, I concluded that the on **documentation** that actually seems to satisfy the criteria of an engineering design is the **source co**

- The **design of a software project** is an abstract concept:
  - It has to do with the overall shape and structure of the program as well as the detailed shape and structure of each module, class, and method.
  - It can be represented by many different diagrams and media, but its final embodiment is the source code.

Source Code is the Primary Document

- Jack Reeves, C++ Journal, 1992, “What is Software Design?”

  > “After reviewing the software development life cycle as
  > I understood it, I concluded that the on
  > **documentation** that actually seems to satisfy the criteria
  > of an engineering design is the **source co**

- The **design of a software project** is an abstract concept:
  - It has to do with the overall shape and structure of the program as well as the detailed shape and structure of each module, class, and method.
  - It can be represented by many different diagrams and media, but its final embodiment is the source code.

Source code is the design

Goals

- 透過一些編碼的潛規則，我們可以寫出一個“好”一點的 C 程式

Goals

- 透過一些編碼的潛規則，我們可以寫出一個“好”一點的C程式
- 除了正確性之外，程式短一點?? 執行快一點???

Goals

- 透過一些編碼的潛規則，我們可以寫出一個“好”一點的 C 程式
- 除了正確性之外，程式短一點?? 執行快一點???
- “好” ? (in terms of test, debug, review, and extension)

Goals

- ✧ 透過一些編碼的潛規則，我們可以寫出一個“好”一點的 C 程式
- ✧ 除了正確性之外，程式短一點?? 執行快一點???
- ✧ “好” ?　　(in terms of test, debug, review, and extension)
  1. 容易了解，沒有邏輯上不緊密結合的資料變數或是敘述

Goals

- 透過一些編碼的潛規則，我們可以寫出一個“好”一點的 C 程式
- 除了正確性之外，程式短一點?? 執行快一點???
- “好” ?　　(in terms of test, debug, review, and extension)

  1. 容易了解，沒有邏輯上不緊密結合的資料變數或是敘述
  2. Self-explaining … 我的程式碼會說話

Goals

- ✧ 透過一些編碼的潛規則，我們可以寫出一個“好”一點的 C 程式

- ✧ 除了正確性之外，程式短一點?? 執行快一點???

- ✧ “好” ?   (in terms of test, debug, review, and extension)

  1. 容易了解，沒有邏輯上不緊密結合的資料變數或是敘述
  2. Self-explaining … 我的程式碼會說話
  3. 和觀念上的運作模型一致

Goals

- ✧ 透過一些編碼的潛規則，我們可以寫出一個“好”一點的 C 程式
- ✧ 除了正確性之外，程式短一點?? 執行快一點???
- ✧ “好”？ (in terms of test, debug, review, and extension)

1. 容易了解，沒有邏輯上不緊密結合的資料變數或是敘述
2. Self-explaining … 我的程式碼會說話
3. 和觀念上的運作模型一致
4. 容易修改，不容易改錯

Goals

- ✧ 透過一些編碼的潛規則，我們可以寫出一個“好”一點的 C 程式
- ✧ 除了正確性之外，程式短一點?? 執行快一點???
- ✧ “好”？ (in terms of test, debug, review, and extension)

  1. 容易了解，沒有邏輯上不緊密結合的資料變數或是敘述
  2. Self-explaining … 我的程式碼會說話
  3. 和觀念上的運作模型一致
  4. 容易修改，不容易改錯
  5. 沒有容易錯誤的語法

Goals

- 透過一些編碼的潛規則，我們可以寫出一個“好”一點的 C 程式
- 除了正確性之外，程式短一點?? 執行快一點???
- “好” ?   (in terms of test, debug, review, and extension)

1. 容易了解，沒有邏輯上不緊密結合的資料變數或是敘述
2. Self-explaining … 我的程式碼會說話
3. 和觀念上的運作模型一致
4. 容易修改，不容易改錯
5. 沒有容易錯誤的語法

正確性無關：接下來是一個很簡單的例子，共有七個版本，執行結果都是正確的

Version 1

```
#include <stdio.h>

void main()
{
    int d[] = {12, 3, 37, 8, 24, 15, 5, 33};
    int n = 8;
    int *d1, *d2;
    int *p;
    int *e;

    d1 = d;
    d2 = d+n;
    while (d1<d2)
    {
        p = d1;
        e = d1 + 1;
        while (e<d2)
        {
            if (*e<*p) p = e;
            e++;
        }
        n = *p;
        *p = *d1;
        *d1 = n;
        d1++;
    }
    printf("Sorted data:\n");
    d1 = d;
    while (d1<d2)
        printf(" %d", *d1++);
    printf("\n");
}
```

Execution Results

```
Sorted data:
    3  5  8  12  15  24  33  37
```

由小至大按順序排列

What is this program doing?

Initial view

What is this program doing?

Initial view

- Input array initialized with unordered integers

What is this program doing?

Initial view

- Input array initialized with unordered integers
- Two layers of while loops

What is this program doing?

Initial view

- Input array initialized with unordered integers
- Two layers of while loops
- Some pointers to the elements of the array

What is this program doing?

Initial view

- ✧ Input array initialized with unordered integers
- ✧ Two layers of while loops
- ✧ Some pointers to the elements of the array
- ✧ Another while loop for output the results

What is this program doing?

Initial view

- Input array initialized with unordered integers
- Two layers of while loops
- Some pointers to the elements of the array
- Another while loop for output the results

Don’t like it!!??

What is this program doing?

Initial view

- Input array initialized with unordered integers
- Two layers of while loops
- Some pointers to the elements of the array
- Another while loop for output the results

Don’t like it!!??

- Pointers

What is this program doing?

Initial view

- Input array initialized with unordered integers
- Two layers of while loops
- Some pointers to the elements of the array
- Another while loop for output the results

Don’t like it!!??

- Pointers
- Generic while loops

What is this program doing?

Initial view

- ✧ Input array initialized with unordered integers
- ✧ Two layers of while loops
- ✧ Some pointers to the elements of the array
- ✧ Another while loop for output the results

Don’t like it!!??

- ✧ Pointers
- ✧ Generic while loops
- ✧ Mysterious variable names (identifier means nothing)

What is this program doing?

Initial view

- Input array initialized with unordered integers
- Two layers of while loops
- Some pointers to the elements of the array
- Another while loop for output the results

Don’t like it!!??

- Pointers
- Generic while loops
- Mysterious variable names (identifier means nothing)
- Deep control structures

What is this program doing?

Initial view

- ✧ Input array initialized with unordered integers
- ✧ Two layers of while loops
- ✧ Some pointers to the elements of the array
- ✧ Another while loop for output the results

Don’t like it!!??

- ✧ Pointers
- ✧ Generic while loops
- ✧ Mysterious variable names (identifier means nothing)
- ✧ Deep control structures

Looks like a snippet of low level assembly instructions

Remove Unnecessary Pointers

- Pointers are sophisticated and sometimes inevitable, but not always.

Remove Unnecessary Pointers

- Pointers are sophisticated and sometimes inevitable, but not always.
- In the case of **accessing memory blocks**, pointers are error prone, use array whenever possible.

Remove Unnecessary Pointers

- ✧ Pointers are sophisticated and sometimes inevitable, but not always.
- ✧ In the case of **accessing memory blocks**, pointers are error prone, use array whenever possible.

  Array syntax has much better semantic meaning than the generic **pointer dereferencing and arithmetics**.

Remove Unnecessary Pointers

- ✧ Pointers are sophisticated and sometimes inevitable, but not always.
- ✧ In the case of **accessing memory blocks**, pointers are error prone, use array whenever possible.

  Array syntax has much better semantic meaning than the generic **pointer dereferencing and arithmetics**.

```
int array[100];
int *ptr=array;
int i, sum = 0;
…
for (i=0; i<100; i++)
    sum += *ptr++;
```

Remove Unnecessary Pointers

- Pointers are sophisticated and sometimes inevitable, but not always.
- In the case of **accessing memory blocks**, pointers are error prone, use array whenever possible.

  Array syntax has much better semantic meaning than the generic **pointer dereferencing and arithmetics**.

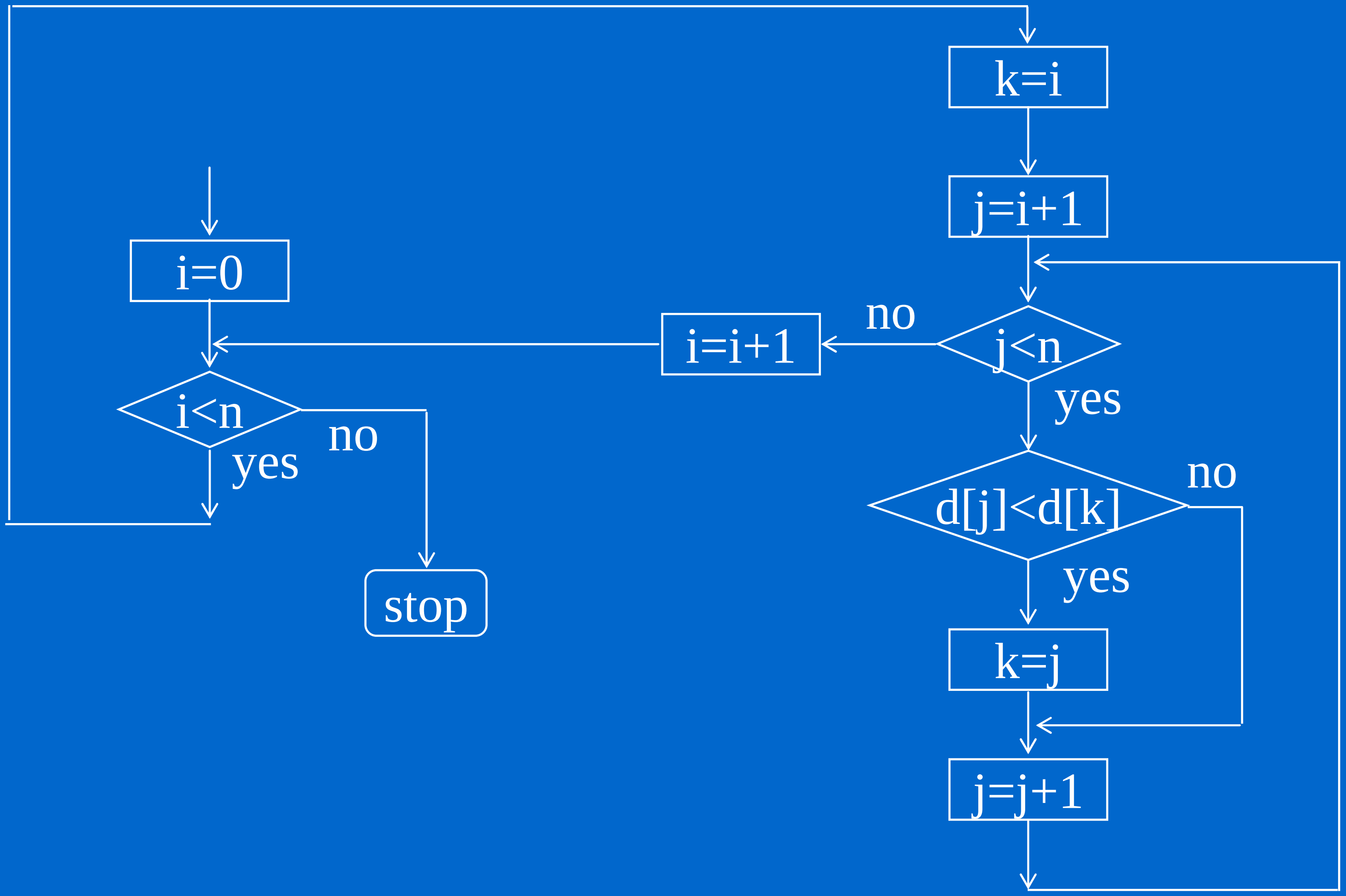
```
int array[100];
int *ptr=array;
int i, sum = 0;
…
for (i=0; i<100; i++)
   sum += *ptr++;
```

```
int array[100];
int i;
int sum = 0;
…
for (i=0; i<100; i++)
   sum += array[i];
```

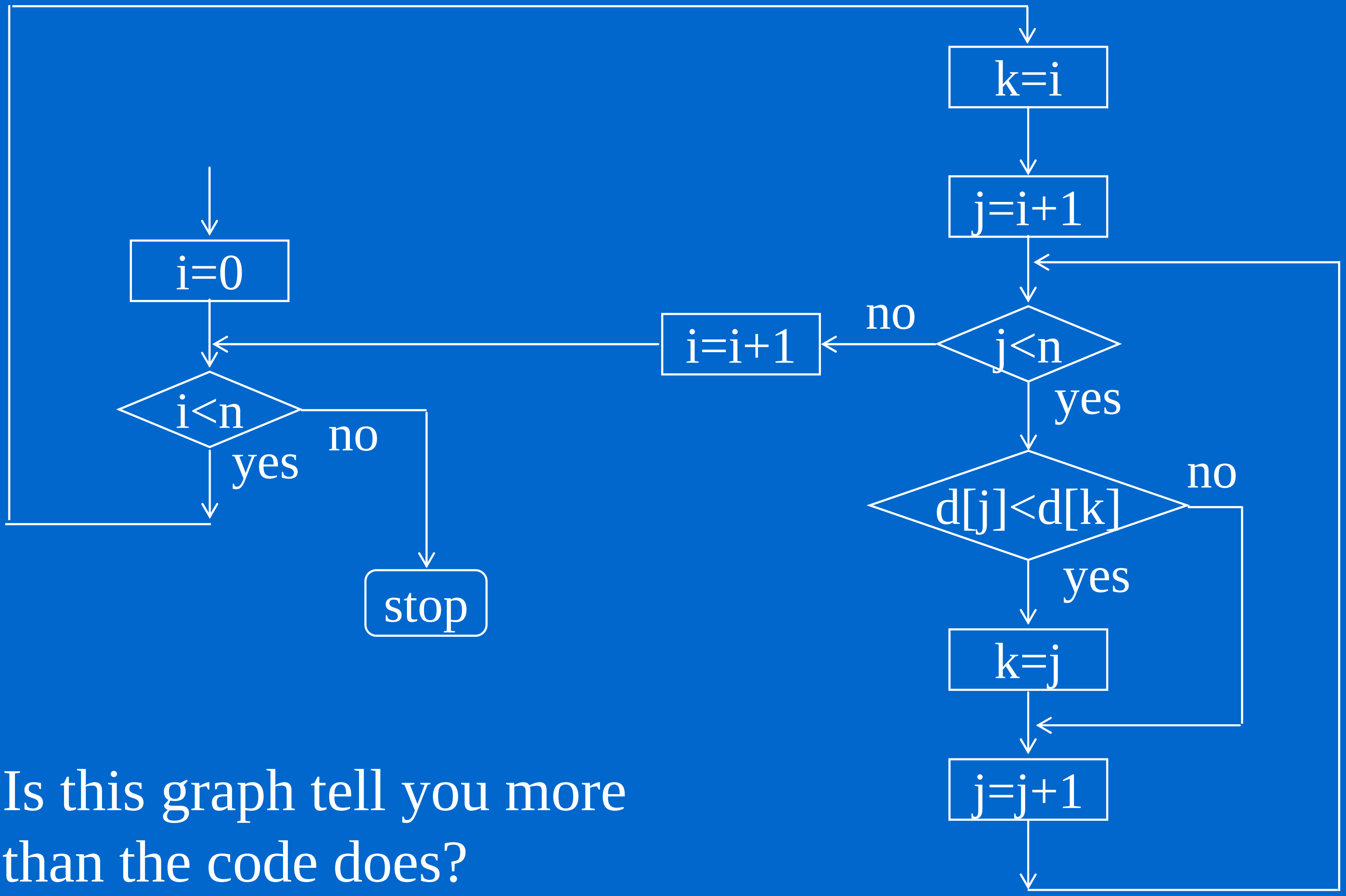
Version 2

```
#include <stdio.h>

void main()
{
    int d[] = {12, 3, 37, 8, 24, 15, 5, 33};
    int n = 8;
    int i, j, k;

    i = 0;
    while (i<n)
    {
        k = i;
        j = i + 1;
        while (j<n)
        {
            if (d[j]<d[k]) k = j;
            j = j + 1;
        }
        j = d[k];
        d[k] = d[i];
        d[i] = j;
        i = i + 1;
    }
    printf("Sorted data:\n");
    i = 0;
    while (i<n)
    {
        printf(" %d", d[i]);
        i = i + 1;
    }
    printf("\n");
}
```

Flowchart of the Program



Flowchart of the Program



Is this graph tell you more than the code does?

Meaningful Identifiers

- ✧ A program is composed with a language. Just like any language in your daily life, language itself should **tell good stories** when it is used properly.

Meaningful Identifiers

- A program is composed with a language. Just like any language in your daily life, language itself should **tell good stories** when it is used properly.

- Why does the version 1 or version 2 program look like gibberish to well trained programmers?

Meaningful Identifiers

- ✧ A program is composed with a language. Just like any language in your daily life, language itself should **tell good stories** when it is used properly.

- ✧ Why does the version 1 or version 2 program look like gibberish to well trained programmers?

Are the **identifiers** used meaningful??

e.g.

Hw ds Jhn lk th stk?

or

How does John like the steak?

Version 3

```
#include <stdio.h>

void main()
{
    int data[] = {12, 3, 37, 8, 24, 15, 5, 33};
    int ndata = sizeof(data) / sizeof(int);
    int i, j;
    int min;
    int swapTmp;

    i = 0;
    while (i<ndata)
    {
        min = i;
        j = i + 1;
        while (j<ndata)
        {
            if (data[j]<data[min]) min = j;
            j = j + 1;
        }
        swapTmp = data[min];
        data[min] = data[i];
        data[i] = swapTmp;
        i = i + 1;
    }

    printf("Sorted data:\n");
    i = 0;
    while (i<ndata)
    {
        printf(" %d", data[i]);
        i = i + 1;
    }
    printf("\n");
}
```

Version 3

```c
#include <stdio.h>

void main()
{
    int data[] = {12, 3, 37, 8, 24, 15, 5, 33};
    int ndata = sizeof(data) / sizeof(int);
    int i, j;
    int min;
    int swapTmp;

    i = 0;
    while (i<ndata)
    {
        min = i;
        j = i + 1;
        while (j<ndata)
        {
            if (data[j]<data[min]) min = j;
            j = j + 1;
        }
        swapTmp = data[min];
        data[min] = data[i];
        data[i] = swapTmp;
        i = i + 1;
    }

    printf("Sorted data:\n");
    i = 0;
    while (i<ndata)
    {
        printf(" %d", data[i]);
        i = i + 1;
    }
    printf("\n");
}
```

avoid magic constants

Advanced View of the Codes

Initial view

- ✧ Input array initialized with unordered integers
- ✧ Two layers of while loops
- ✧ Some pointers to the elements of the array
- ✧ Another while loop for output the results

Advanced View of the Codes

Initial view

- ✧ Input array initialized with unordered integers
- ✧ Two layers of while loops
- ✧ Some pointers to the elements of the array
- ✧ Another while loop for output the results

Is it changing?

Advanced View of the Codes

Initial view

- ✧ Input array initialized with unordered integers
- ✧ Two layers of while loops
- ✧ Some pointers to the elements of the array
- ✧ Another while loop for output the results

Is it changing?

- ✧ Input array initialized with unordered integers

- ✧ Another while loop for output the results

Advanced View of the Codes

Initial view

- Input array initialized with unordered integers
- Two layers of while loops
- Some pointers to the elements of the array
- Another while loop for output the results

Is it changing?

- Input array initialized with unordered integers
- Two layers of while loops, the outer one prepares ndata sub-arrays, the inner one goes through each sub-array to find something minimal.
- Another while loop for output the results

Advanced View of the Codes

Initial view

- Input array initialized with unordered integers
- Two layers of while loops
- Some pointers to the elements of the array
- Another while loop for output the results

Is it changing?

- Input array initialized with unordered integers
- Two layers of while loops, the outer one prepares ndata sub-arrays, the inner one goes through each sub-array to find something minimal.
- A snippet of memory swapping code
- Another while loop for output the results

More Meaningful Language Construct

- While loop is the most generic repetition construct in C language

More Meaningful Language Construct

- While loop is the most generic repetition construct in C language

Forget about ***goto*** please!!!

More Meaningful Language Construct

- While loop is the most generic repetition construct in C language

```
initialize the loop condition
while (condition)
{
   …
}
```

More Meaningful Language Construct

✧ While loop is the most generic repetition construct in C language

```
initialize the loop condition
while (condition)
{
    …
}
```

the condition is likely to change inside the loop

More Meaningful Language Construct

- While loop is the most generic repetition construct in C language

```
initialize the loop condition
while (condition)
{
    …
}
```

the condition is likely to change inside the loop

- When you see this construct in a program, you expect some sort of job repetition, could be an easy one or a complex one.

More Meaningful Language Construct

- While loop is the most generic repetition construct in C language

```
initialize the loop condition
while (condition)
{
    …
}
```

the condition is likely to change inside the loop

- When you see this construct in a program, you expect some sort of job repetition, could be an easy one or a complex one.

  For loop is a more semantically specific repetition construct in C language --- **repeat for a predetermined number of times**

More Meaningful Language Construct

- While loop is the most generic repetition construct in C language

```
initialize the loop condition
while (condition)
{
    …
}
```

the condition is likely to change inside the loop

- When you see this construct in a program, you expect some sort of job repetition, could be an easy one or a complex one.

  For loop is a more semantically specific repetition construct in C language --- **repeat for a predetermined number of times**

```
for (i=0; i<count; i++)
{
    …
}
```

Version 4

```
#include <stdio.h>

void main()
{
    int data[] = {12, 3, 37, 8, 24, 15, 5, 33};
    int ndata = sizeof(data) / sizeof(int);
    int i, j;
    int min;
    int swapTmp;

    for (i=0; i<ndata; i++)
    {
        min = i;
        for (j=i+1; j<ndata; j++)
        {
            if (data[j]<data[min]) min = j;
        }
        swapTmp = data[min];
        data[min] = data[i];
        data[i] = swapTmp;
    }

    printf("Sorted data:\n");
    for (i=0; i<ndata; i++)
        printf(" %d", data[i]);
    printf("\n");
}
```

Code That Further Illustrates Itself

- **Function** is a powerful construct to **abstract** ideas, to **hide** distracting details, not just a utility for saving your typing time and removing redundancy.

  --- Version 5

Code That Further Illustrates Itself

- ✧ **Function** is a powerful construct to **abstract** ideas, to **hide** distracting details, not just a utility for saving your typing time and removing redundancy.

  --- Version 5

- ✧ A construct like **“loop inside a loop”** is somehow beyond the concrete control of common human mind. A single layer of “loop” is better for most people to visualize in mind.

  --- Version 6

Version 5

```
#include <stdio.h>

void swap(int *, int *);
void printArrayContents(int [], int);

void main()
{
    int data[] = {12, 3, 37, 8, 24, 15, 5, 33};
    int ndata = sizeof(data) / sizeof(int);
    int i, j;
    int min;

    for (i=0; i<ndata; i++)
    {
        min = i;
        for (j=i+1; j<ndata; j++)
        {
            if (data[j]<data[min]) min = j;
        }
        swap(&data[i], &data[min]);
    }

    printArrayContents(data, ndata);
}

void swap(int *x, int *y)
{
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

Version 5 (cont’d)

```

void printArrayContents(int data[], int ndata)
{
    int i;
    printf("Sorted data:\n");
    for (i=0; i<ndata; i++)
        printf(" %d", data[i]);
    printf("\n");
}
```

Version 6

```
#include <stdio.h>

void selectionSort(int[], int);
void findMinimumOfAnArray(int[], int);
void swap(int*, int*);
void printArrayContents(int[], int);

void main()
{
    int data[] = {12, 3, 37, 8, 24, 15, 5, 33};
    int ndata = sizeof(data) / sizeof(int);

    selectionSort(data, ndata);
    printArrayContents(data, ndata);
}
```

Version 6 (cont’d)

suitable level of details

```
void selectionSort(int data[], int ndata)
{
    int i;
    for (i=0; i<ndata; i++)
        putMinimalElementInPlace(&data[i], ndata-i);
}

void putMinimalElementInPlace(int data[], int ndata)
{
    int i, min;

    min = 0;
    for (i=1; i<ndata; i++)
    {
        if (data[i]<data[min]) min = i;
    }
    swap(&data[0], &data[min]);
}

```

```
void swap(int *x, int *y)
{
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}

```

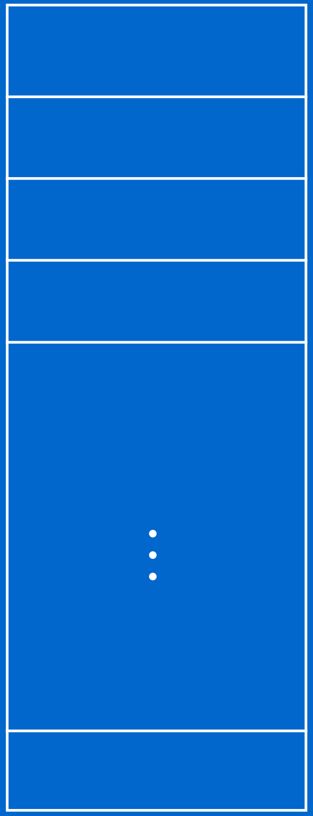
```
void printArrayContents(int data[], int ndata)
{
    int i;
    printf("Sorted data:\n");
    for (i=0; i<ndata; i++)
        printf(" %d", data[i]);
    printf("\n");
}
```

Codes with a Conceptual Model

- ✧ Flowchart is no longer needed but definitely requires a conceptual **model** for the codes to work with.

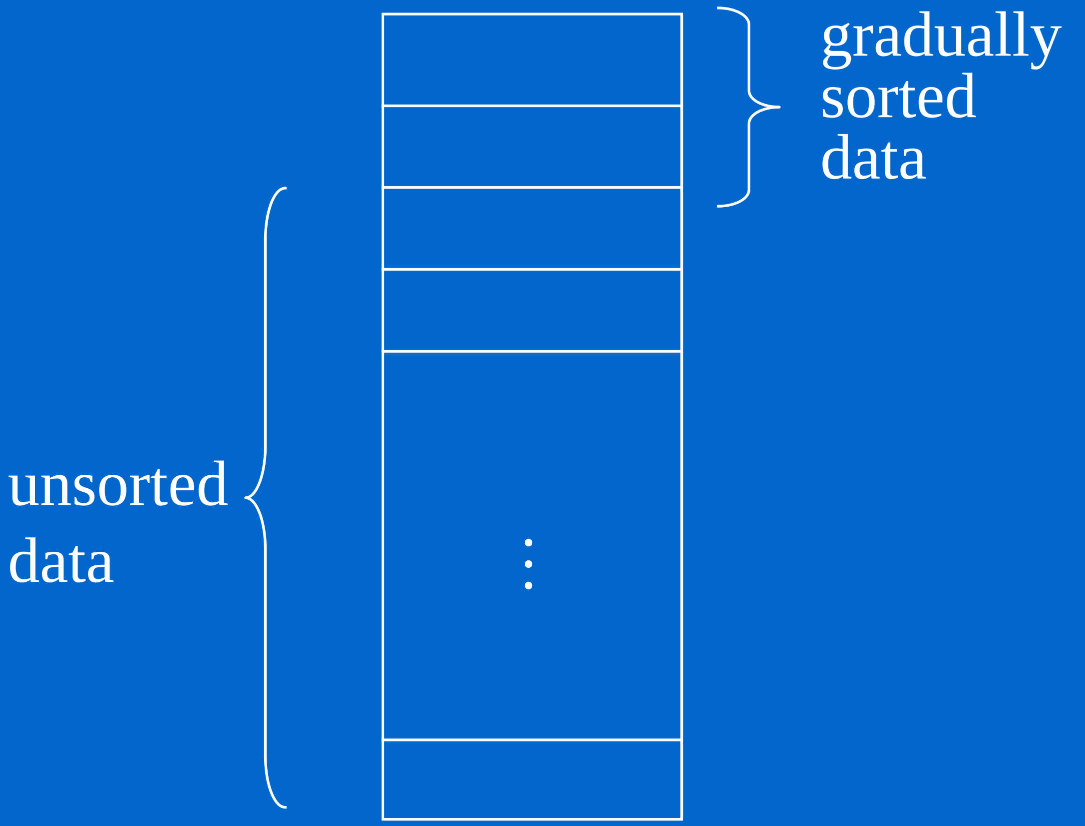
Codes with a Conceptual Model

- Flowchart is no longer needed but definitely requires a conceptual **model** for the codes to work with.



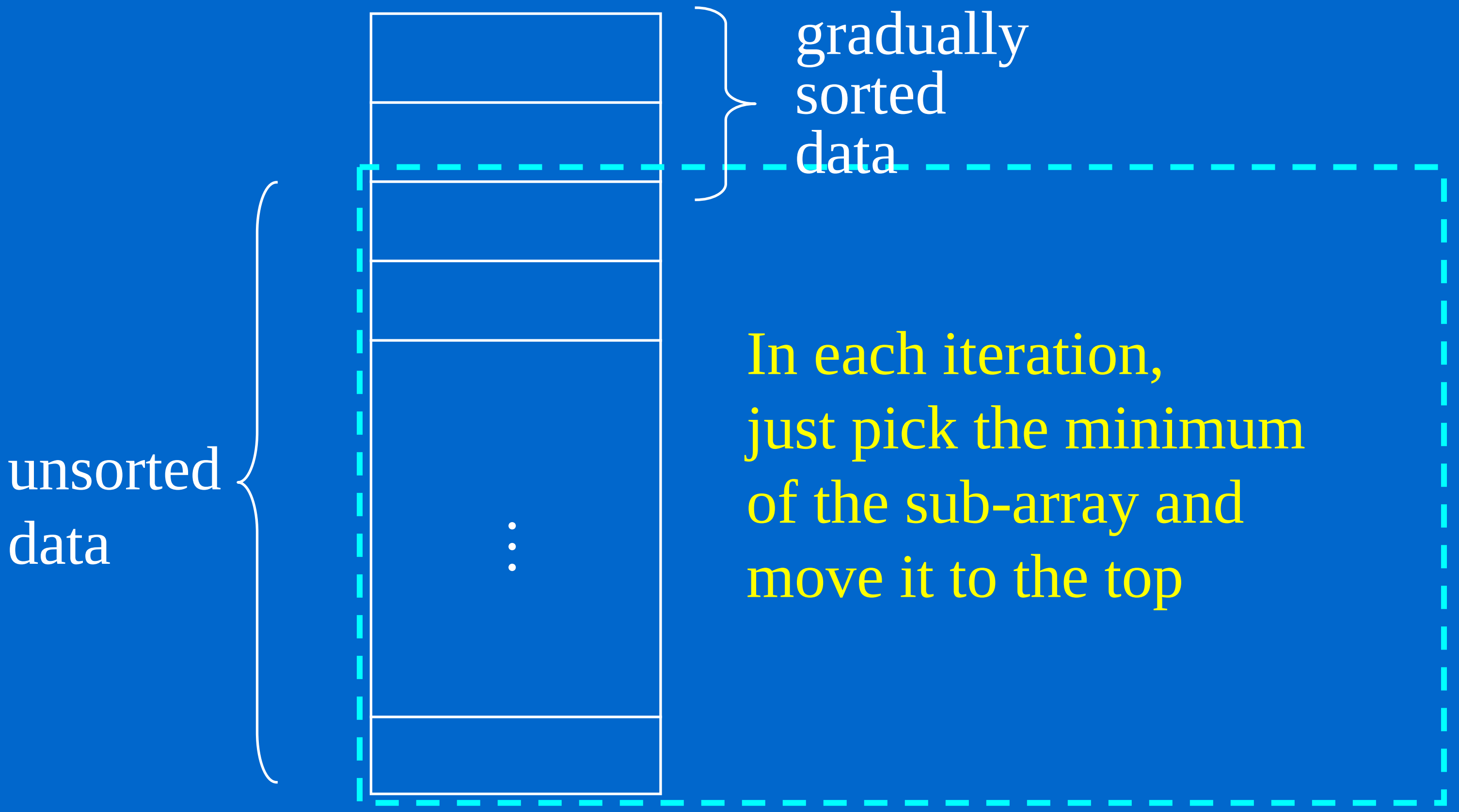
Codes with a Conceptual Model

✧ Flowchart is no longer needed but definitely requires a conceptual **model** for the codes to work with.



Codes with a Conceptual Model

- ✧ Flowchart is no longer needed but definitely requires a conceptual **model** for the codes to work with.



In each iteration, just pick the minimum of the sub-array and move it to the top

Who is responsible of this task?

Who is responsible of this task?

- ✧ The programmer or the program reader?

Who is responsible of this task?

- The programmer or the program reader?
- When we read the version 1 of this program, there were little clues in the codes that told us directly what the program is doing.

Who is responsible of this task?

- The programmer or the program reader?
- When we read the version 1 of this program, there were little clues in the codes that told us directly what the program is doing.
- Although we figure out that this is a piece of code that implements the selection sort algorithm at last, it should not take the original programmer too much effort to produce a code snippet like version 6 and its corresponding conceptual model which tell directly the story of what the program is doing.

Who is responsible of this task?

- The programmer or the program reader?
- When we read the version 1 of this program, there were little clues in the codes that told us directly what the program is doing.
- Although we figure out that this is a piece of code that implements the selection sort algorithm at last, it should not take the original programmer too much effort to produce a code snippet like version 6 and its corresponding conceptual model which tell directly the story of what the program is doing.

A piece of code is to implement some engineering design, **simplicity** is the best engineering principle. Try your best to think and express ideas in an intuitive way.

Recursive Version

- ✧ Recursive version is often the most expressive form of the underlying algorithm.

Recursive Version

- ✧ Recursive version is often the most expressive form of the underlying algorithm.

```
void selectionSort(int data[], int ndata)
{
    putMinimalElementInPlace(data, ndata);
    if (ndata>2)
        selectionSort(&data[1], ndata-1);
}
```

Efficiency Issues

- Using expressive name for all identifiers makes the program much lengthier, easier to have typos, slow in composing the program!!!   Really??

Efficiency Issues

- ✧ Using expressive name for all identifiers makes the program much lengthier, easier to have typos, slow in composing the program!!!   Really??
  - ✶ Harddisk is cheap. Not necessary to think of space.

Efficiency Issues

- Using expressive name for all identifiers makes the program much lengthier, easier to have typos, slow in composing the program!!!   Really??
  - Harddisk is cheap. Not necessary to think of space.
  - It is easier for compiler to detect typo than using x, y, z.

Efficiency Issues

- ✧ Using expressive name for all identifiers makes the program much lengthier, easier to have typos, slow in composing the program!!!   Really??
  - ✶ Harddisk is cheap. Not necessary to think of space.
  - ✶ It is easier for compiler to detect typo than using x, y, z.
  - ✶ Typing should not be the bottleneck.

Efficiency Issues

- Using expressive name for all identifiers makes the program much lengthier, easier to have typos, slow in composing the program!!!   Really??
  - Harddisk is cheap. Not necessary to think of space.
  - It is easier for compiler to detect typo than using x, y, z.
  - Typing should not be the bottleneck.
  - Expressive programs are easier to compose, maintain, and extend.

Efficiency Issues

- ✧ Using expressive name for all identifiers makes the program much lengthier, easier to have typos, slow in composing the program!!! Really??
  - ✶ Harddisk is cheap. Not necessary to think of space.
  - ✶ It is easier for compiler to detect typo than using x, y, z.
  - ✶ Typing should not be the bottleneck.
  - ✶ Expressive programs are easier to compose, maintain, and extend.
- ✧ Excessive function calls take CPU time to transfer arguments and to branch the control.

Efficiency Issues

- ✧ Using expressive name for all identifiers makes the program much lengthier, easier to have typos, slow in composing the program!!!   Really??
  - ✶ Harddisk is cheap. Not necessary to think of space.
  - ✶ It is easier for compiler to detect typo than using x, y, z.
  - ✶ Typing should not be the bottleneck.
  - ✶ Expressive programs are easier to compose, maintain, and extend.
- ✧ Excessive function calls take CPU time to transfer arguments and to branch the control.
  - ✶ Let the compiler worry about it --- use inline function.

Efficiency Issues

- ✧ Using expressive name for all identifiers makes the program much lengthier, easier to have typos, slow in composing the program!!!   Really??
  - ✶ Harddisk is cheap. Not necessary to think of space.
  - ✶ It is easier for compiler to detect typo than using x, y, z.
  - ✶ Typing should not be the bottleneck.
  - ✶ Expressive programs are easier to compose, maintain, and extend.
- ✧ Excessive function calls take CPU time to transfer arguments and to branch the control.
  - ✶ Let the compiler worry about it --- use inline function.
- ✧ Using dedicated variables for independent tasks looks like abusing memories.

Efficiency Issues

- ✧ Using expressive name for all identifiers makes the program much lengthier, easier to have typos, slow in composing the program!!! Really??
  - ✶ Harddisk is cheap. Not necessary to think of space.
  - ✶ It is easier for compiler to detect typo than using x, y, z.
  - ✶ Typing should not be the bottleneck.
  - ✶ Expressive programs are easier to compose, maintain, and extend.
- ✧ Excessive function calls take CPU time to transfer arguments and to branch the control.
  - ✶ Let the compiler worry about it --- use inline function.
- ✧ Using dedicated variables for independent tasks looks like abusing memories.
  - ✶ Let the compiler worry about it.

Efficiency Issues

- Using expressive name for all identifiers makes the program much lengthier, easier to have typos, slow in composing the program!!!   Really??
  - Harddisk is cheap. Not necessary to think of space.
  - It is easier for compiler to detect typo than using x, y, z.
  - Typing should not be the bottleneck.
  - Expressive programs are easier to compose, maintain, and extend.
- Excessive function calls take CPU time to transfer arguments and to branch the control.
  - Let the compiler worry about it --- use inline function.
- Using dedicated variables for independent tasks looks like abusing memories.
  - Let the compiler worry about it.

Assignments

- ✧ Bubble Sort
- ✧ Quick Sort
- ✧ Minimum Spanning Tree
- ✧ Tree Traversal
- ✧ …