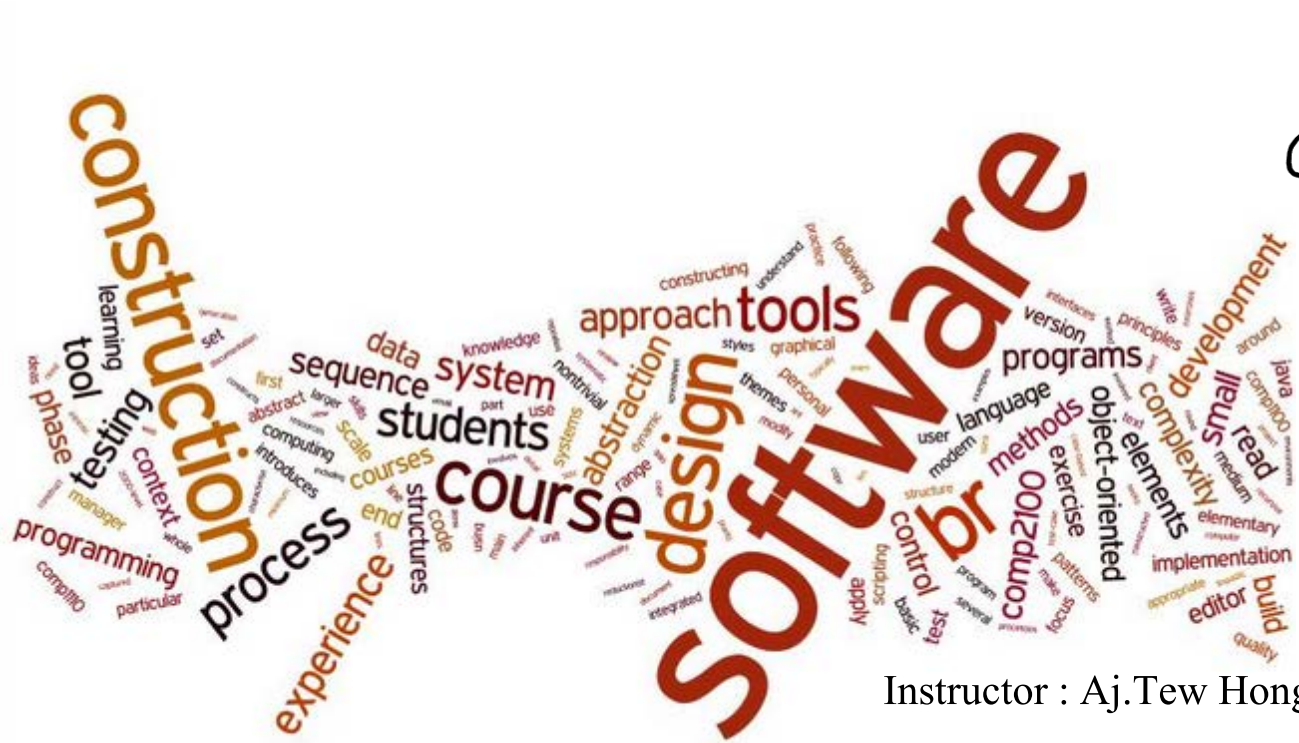
SOFTWARE CONSTRUCTION AND EVOLUTION

General Issues in Using Variables



Instructor : Aj.Tew Hongthong

School of IT, Mae Fah Luang University (MFU)

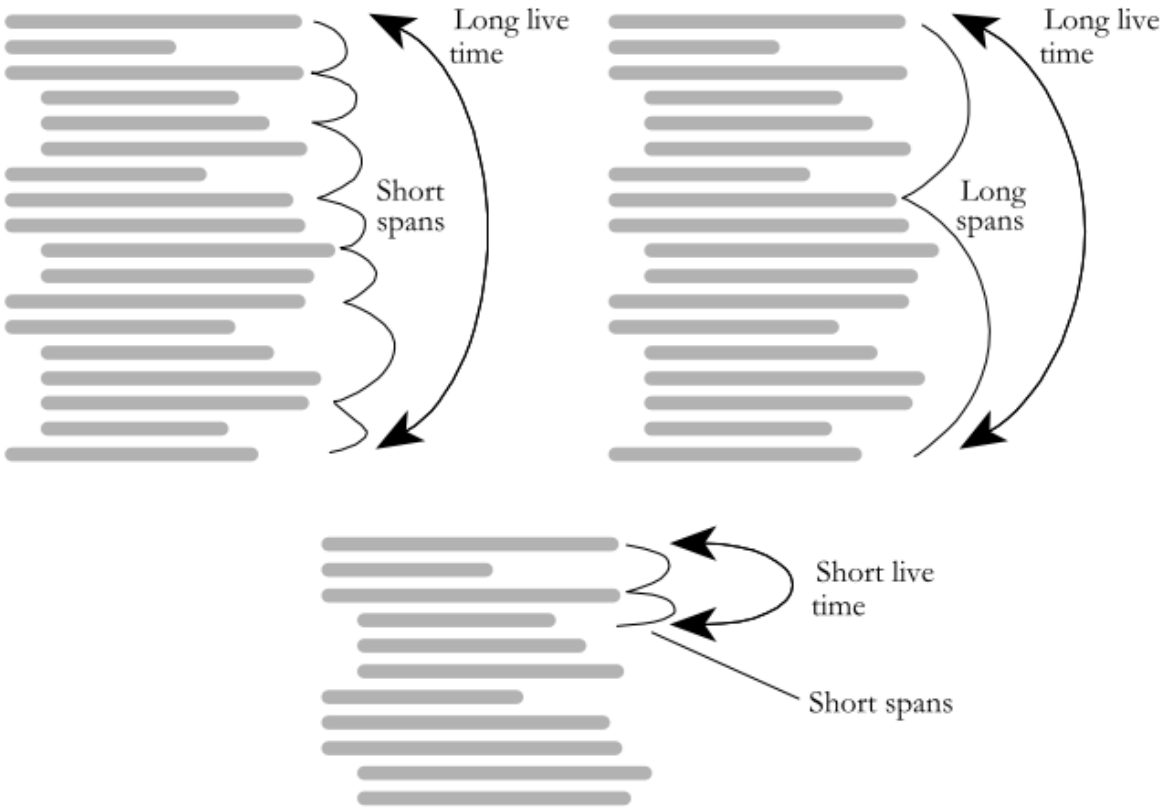
CONTENTS

- Scope of variable
- Good Variable Names
- Number Data Type In General
- Integer
- Floating Point

Scope of Variables

- “Scope” is a way of thinking about a variable’s celebrity status (How famous it is?)
  - A loop index “i” is used in only one small loop
  - A table of “employee” information is used throughout a program
- “span” how close together the references to a variable used.
- Keep variable live for as short a time as possible
- A variable’s life begins at the first statement in which it’s referenced; its life ends at the last statement in which it’s referenced.

Keep Variables Live for As Short a Time As Possible



- "Long live time" a variable is alive over the course of many statements.
- "short live time" it's alive for only a few statement.
- span refers to how close together the references to a variable are.

Advantage of keeping a live time short

- Give you an accurate picture of your code.
  - If a variable is assigned a value in line 10 and not used again until line 45, implies that the variable is used between lines 10 and 45.
  - If a variable is assigned a value in line 44, 45 only, we can concentrate on a smaller section of code.
- A short live time reduces the chance of initialization errors.
  - Keeping the initialization code and the loop closer together, can reduce the chance that modifications will introduce initiation errors.

Advantage of keeping a live time short

- Make your code more readable
  - the fewer lines of code a reader has to keep in mind at once, the easier your code is to understand.
  - the shorter the live time, the less code you have to keep on your screen when you want to see all the references to a variable during editing and debugging.

Java Example of Variables with Bad Long Live Times

```
1   // initialize all variables
2   recordIndex = 0;
3   total = 0;
4   done = false;
    ...
26  while ( recordIndex < recordCount ) {
27  ...
28     recordIndex = recordIndex + 1;
       ...

64  while ( !done ) {
       ...
69     if ( total > projectedTotal ) {
70        done = true;
```

recordIndex (line28-line2+1) = 27

total (line69-line3+1) = 67

done (line70-line4+1) = 67

Average Live Time (27+67+67)/3 = 54

Java Example of Variables with Good Long Live Times

```
    ...
25  recordIndex = 0;
26  while ( recordIndex < recordCount ) {
27  ...
28     recordIndex = recordIndex + 1;
       ...
62  total = 0;
63  done = false;
64  while ( !done ) {
       ...
69     if ( total > projectedTotal ) {
70        done = true;
```

recordIndex (line28-line25+1) = 4

total (line69-line62+1) = 8

done (line70-line63+1) = 8

Average Live Time (4+8+8)/3 = 7

General Guidelines for Minimizing Scope

- Initialize variables used in a loop immediately before the loop
- Don’t assign a value to a variable until just before the value is used

C++ Example of Good Variable Declarations and Initializations

```
int receiptIndex = 0;
float dailyReceipts = TodaysReceipts();
double totalReceipts = TotalReceipts( dailyReceipts );
```

General Guidelines for Minimizing Scope

- Group related statements

C++ Example of Using Two Sets of Variables in a Confusing Way

```
void SummarizeData (...) {
   ...
   GetOldData( oldData, &numOldData );
   GetNewData( newData, &numNewData );
   totalOldData = Sum( oldData, numOldData );
   totalNewData = Sum( newData, numNewData );
   PrintOldDataSummary( oldData, totalOldData, numOldData );
   PrintNewDataSummary( newData, totalNewData, numNewData );
   SaveOldDataSummary( totalOldData, numOldData );
   SaveNewDataSummary( totalNewData, numNewData );
   ...
}
```

General Guidelines for Minimizing Scope

- Group related statements

C++ Example of Using Two Sets of Variables More Understandably

```
void SummarizeDaily( ... ) {
   GetOldData( oldData, &numOldData );
   totalOldData = Sum( oldData, numOldData );
   PrintOldDataSummary( oldData, totalOldData, numOldData );
   SaveOldDataSummary( totalOldData, numOldData );
   ...
   GetNewData( newData, &numNewData );
   totalNewData = Sum( newData, numNewData );
   PrintNewDataSummary( newData, totalNewData, numNewData );
   SaveNewDataSummary( totalNewData, numNewData );
   ...
}
```

Using Each Variable for Exactly One Purpose

C++ Example of Using One Variable for Two Purposes—Bad Practice

```
// Compute roots of a quadratic equation.
// This code assumes that (b*b-4*a*c) is positive.
temp = Sqrt( b*b - 4*a*c );
root[0] = ( -b + temp ) / ( 2 * a );
root[1] = ( -b - temp ) / ( 2 * a );
...
// swap the roots
temp = root[0];
root[0] = root[1];
root[1] = temp;
```

Question: What is the relationship between temp in the first lines and temp in the last few?

Answer: The two temp have no relationship. Using the same varible in both instances makes it seems as though they're related.

Using Each Variable for Exactly One Purpose

C++ Example of Using Two Variables for Two Purposes—Good Practice

```
// Compute roots of a quadratic equation.
// This code assumes that (b*b-4*a*c) is positive.
discriminant = Sqrt( b*b - 4*a*c );
root[0] = ( -b + discriminant ) / ( 2 * a );
root[1] = ( -b - discriminant ) / ( 2 * a );
...
// swap the roots
oldRoot = root[0];
root[0] = root[1];
root[1] = oldRoot;
```

Creating unique variables for each purpose makes your code more readable.

Check List: General Considerations In Using Data

- Does the code declare variables close to where they're first used?
- Does the code initialize variables close to where they're first used, if it isn't possible to declare and initialize them at the same time?
- Do all variables have the smallest scope possible?
- Are all the declared variables being used?
- Does each variable have one and only one purpose?

Considerations in Choosing Good Variable Names

Java Example of Poor Variable Names

```
x = x - xx;
xxx = aretha + SalesTax( aretha );
x = x + LateFee( x1, x ) + xxx;
x = x + Interest( x1, x );
```

- What do *x1, xx,* and *xxx* mean?
- What do *aretha* mean?

Java Example of Good Variable Names

```
balance = balance - lastPayment;
monthlyTotal = NewPurchases + SalesTax( newPurchases );
balance = balance + LateFee( customerID, balance ) + monthlyTotal;
balance = balance + Interest( customerID, balance );
```

- a good variable name is readable, memorable, and appropriate.

Considerations in Choosing Good Variable Names

Example:

- For a variable that represents the number of people on the U.S. Olympic team, you would create the name *numberOfPeopleOnTheUsOlympicTeam*.
- A variable that represents the number of seats in a stadium would be *numberOfSeatsInTheStadium*.
- A variable that represents the maximum number of points scored by a country's team in any modern Olympics would be *maximumNumberOfPointsInModernOlympics*.
- A variable that contains the current interest rate is better named *rate* or *interestRate* than *r* or *x*.

Example of Good and Bad Variable

Purpose of Variable	Good Names, Good Descriptors	Bad Names, Poor Descriptors
Running total of checks written to date	*runningTotal, checkTotal*	*written, ct, checks, CHKTTL, x, x1, x2*
Velocity of a bullet train	*velocity, trainVelocity, velocityInTrain*	*velt, v, tv, x, x1, x2, train*
Current date	*currentDate, todaysDate*	*cd, current, c, x, x1, x2, date*
Lines per page	*linesPerPage*	*lpp, lines, l, x, x1, x2*

Optimum Name Length

- On Average; 8 to 20 characters were almost as easy to debug
- Variable Names That Are Too Long, Too Short, or Just Right

Too long:	*numberOfPeopleOnTheUsOlympicTeam*
	numberOfSeatsInTheStadium
	maximumNumberOfPointsInModernOlympics
Too short:	*n*, *np*, *ntm*
	n, *ns*, *nsisd*
	m, *mp*, *max*, *points*
Just right:	*numTeamMembers*, *teamMemberCount*

Naming Specific Type of Data

Naming Loop Indexes

Java Example of a Simple Loop Variable Name

```
for ( i = firstItem; i < lastItem; i++ ) {
   data[ i ] = 0;
}
```

If a variable is to be used outside the loop, it should be given a name more meaningful than *i, j,* or *k*.

Java Example of a Good Descriptive Loop Variable Name

```
recordCount = 0;
while ( moreScores() ) {
   score[ recordCount ] = GetNextScore();
   recordCount++;
}

// lines using recordCount
...
```

Naming Specific Type of Data

Naming Temporary Variable

- **C++ Example of an Uninformative "Temporary" Variable Name**

```
// Compute roots of a quadratic equation.
// This assumes that (b^2-4*a*c) is positive.
  temp = sqrt( b^2 - 4*a*c );
  root[0] = ( -b + temp ) / ( 2 * a );
  root[1] = ( -b - temp ) / ( 2 * a );
```

- **C++ Example with a "Temporary" Variable Name Replaced with a Real Variable**

```
// Compute roots of a quadratic equation.
// This assumes that (b^2-4*a*c) is positive.
    discriminant = sqrt( b^2 - 4*a*c );
    root[0] = ( -b + discriminant ) / ( 2 * a );
    root[1] = ( -b - discriminant ) / ( 2 * a );
```

Naming Specific Type of Data

Naming Boolean Variable

- Give Boolean variables names that imply TRUE or FALSE
  - *Done (true, false)*
  - *Error (true, false)*
  - *Found (true, false)*
  - *Success (true, false)*
  - *OK (true, false)*
- Use positive Boolean variable Names
  - *notFound, notdone, notSuccessful* are difficult to read when they are negated: If not *notFound*

When you should have a Naming Convention

- When multiple programmers are working on a project
- When you plan to turn a program over to another programmer for modifications and maintenance (which is nearly always)
- When your programs are reviewed by other programmers in your organization
- When your program is so large that you can't hold the whole thing in your brain at once and must think about it in pieces
- When the program will be long-lived enough that you might put it aside for a few weeks or months before working on it again
- When you have a lot of unusual terms that are common on a project and want to have standard terms or abbreviations to use in coding

Number Data Type In General

A magic number is a direct usage of a number in the code

```
public class Foo {
    public void setPassword(String password) {
         // don't do this
         if (password.length() > 7) {
              throw new InvalidArgumentException("password");
         }
    }
}
```

This should be changed to:

```
public class Foo {
    public static final int MAX_PASSWORD_SIZE = 7;

    public void setPassword(String password) {
         if (password.length() > MAX_PASSWORD_SIZE) {
              throw new InvalidArgumentException("password");
         }
    }
}
```

Number Data Type In General

Use hard-coded 0s and 1s if you need to

```
for i = 0 to CONSTANT do ...
```

is OK, and the 1 in

```
total = total + 1
```

is OK. A good rule of thumb is that the only literals that should occur in the body of a program are *0* and *1*. Any other literals should be replaced with something more descriptive.

Number Data Type In General

- Anticipate divide-by-zero errors
- Make type conversions obvious
  - y = x+(float) i
- Avoid mixed-type comparisons
  - x:float, i:interger => if (i=x) …..
- Heed your compiler’s warnings
  - Pay attention about complier’s warning, it helps you to prevent the error

Things to consider before using Integer

- Check for integer division
  - 7/10 = 0
  - 10*(7/10) = 0, (10*7)/10 = 7
- Check for integer overflow

Integer Type	Range
Signed 8-bit	-128 through 127
Unsigned 8-bit	0 through 255
Signed 16-bit	-32,768 through 32,767
Unsigned 16-bit	0 through 65,535
Signed 32-bit	-2,147,483,648 through 2,147,483,647
Unsigned 32-bit	0 through 4,294,967,295
Signed 64-bit	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807
Unsigned 64-bit	0 through 18,446,744,073,709,551,615

Things to consider before using Floating

Avoid additions and subtractions on numbers that have greatly different magnitudes

1,000,000.00 + 0.1 probably produces an answer of 1,000,000.00 because 32 bits don't give you enough significant digits to encompass the range between 1,000,000 and 0.1.

Things to consider before using Floating

Avoid equality comparisons

Java Example of a Bad Comparison of Floating-Point Numbers

```
double nominal = 1.0;
double sum = 0.0;

for ( int i = 0; i < 10; i++ ) {
   sum += 0.1;
}

if ( nominal == sum ) {
   System.out.println( "Numbers are the same." );
}
else {
   System.out.println( "Numbers are different." ) ;
}
```

```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
```

Things to consider before using Floating

Java Example of a Routine to Compare Floating-Point Numbers

```
double const ACCEPTABLE_DELTA = 0.00001;
boolean Equals( double Term1, double Term2 ) {
   if ( Math.abs( Term1 - Term2 ) < ACCEPTABLE_DELTA ) {
      return true;
   }
   else {
      return false;
   }
}
```

Kinds of names to avoid:

- **Avoid misleading names or abbreviations.**

As FALSE as abb for "Fig and Almond Season."

- **Avoid names with similar meanings.**

As fileNumber and fileIndex.

- **Avoid variables with different meanings but similar names.**

As clientRecs and clientReps.

- **Avoid names that sound similar.**

As wrap and rap.

- **Avoid numerals in names.**

As file1, file2.

Kinds of names to avoid:

- **Avoid words that are commonly misspelled in English.**

As occassionally, accumulate, and ascend.

- **Don't differentiate variable names solely by capitalization.**

As count and Count.

- **Avoid multiple natural languages.**

As "color" or "colour".

Do details like these really matter?

Programmers over the lifetime of a system spend more time reading code than writing code.

Example for Low Quality Coding!! (C++)

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec, double
   & estimRevenue, double ytdRevenue, int screenX, int screenY, COLOR_TYPE &
   newColor, COLOR_TYPE & prevColor, StatusType & status, int expenseType )
{
int i;
for ( i = 0; i < 100; i++ ) {
   inputRec.revenue[i] = 0;
   inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
   }
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
if ( expenseType == 1 ) {
     for ( i = 0; i < 12; i++ )
           profit[i] = revenue[i] - expense.type1[i];
     }
else if ( expenseType == 2 )  {
          profit[i] = revenue[i] - expense.type2[i];
          }
else if ( expenseType == 3 )
          profit[i] = revenue[i] - expense.type3[i];
          }
```

Find at least 10 different problems with it!

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec, double
   & estimRevenue, double ytdRevenue, int screenX, int screenY, COLOR_TYPE &
   newColor, COLOR_TYPE & prevColor, StatusType & status, int expenseType )
{
int i;
for ( i = 0; i < 100; i++ ) {
   inputRec.revenue[i] = 0;
   inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
   }
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
if ( expenseType == 1 ) {
     for ( i = 0; i < 12; i++ )
          profit[i] = revenue[i] - expense.type1[i];
     }
else if ( expenseType == 2 )  {
          profit[i] = revenue[i] - expense.type2[i];
          }
else if ( expenseType == 3 )
          profit[i] = revenue[i] - expense.type3[i];
          }
```

7. Only CORP_DATA is used!!

1. Bad Name!!

5. Doesn't defend itself against BAD data!! (crntQtr = 0! Error!)

2. Input variable is changed!

3.Read&Write Global Variable (corpExpense, profit)

4. Not Single purpose

- Write to DB, Calculate Xxx

6. Magic Number!! 100, 4.0, 12, 123?