SWEN 6301 Software Construction

Module 6: Variables

Ahmed Tamrawi

Outline

- Data Literacy
- Variable Initialization
- Scope
- One Variable = One Purpose

Data Literacy

- The first step in creating **effective data** is knowing which **kind of data to create**.
- A **good repertoire of data types** is a key part of a programmer's toolbox.

Variable Initialization

- Improper data initialization is one of the most fertile sources of error in computer programming.
- Developing effective techniques for avoiding initialization problems can save a lot of debugging time.

Variable Initialization

- The problems with improper initialization stem from a variable's containing an initial value that you do not expect it to contain.
- This can happen for any of several reasons:
  - **The variable has never been assigned a value**. Its value is whatever bits happened to be in its area of memory when the program started.
  - **The value in the variable is outdated**. The variable was assigned a value at some point, but the value is no longer valid.
  - **Part of the variable has been assigned a value and part has not**. The case of classes instances.

Variable Initialization

Initialize each variable as it's declared

- Initializing variables as they're declared is an ***inexpensive form of defensive programming***. It's a good insurance policy against initialization errors.
- The example below ensures that *studentGrades* will be reinitialized each time you call the routine that contains it.

C++ Example of Initialization at Declaration Time

```
float studentGrades[ MAX_STUDENTS ] = { 0.0 };
```

Variable Initialization

Initialize each variable close to where it 's first used

- Some languages, including Visual Basic, don't support initializing variables as they're declared.
- That can lead to coding styles like the following one, in which declarations are grouped together and then initializations are grouped together all far from the first actual of the variables.

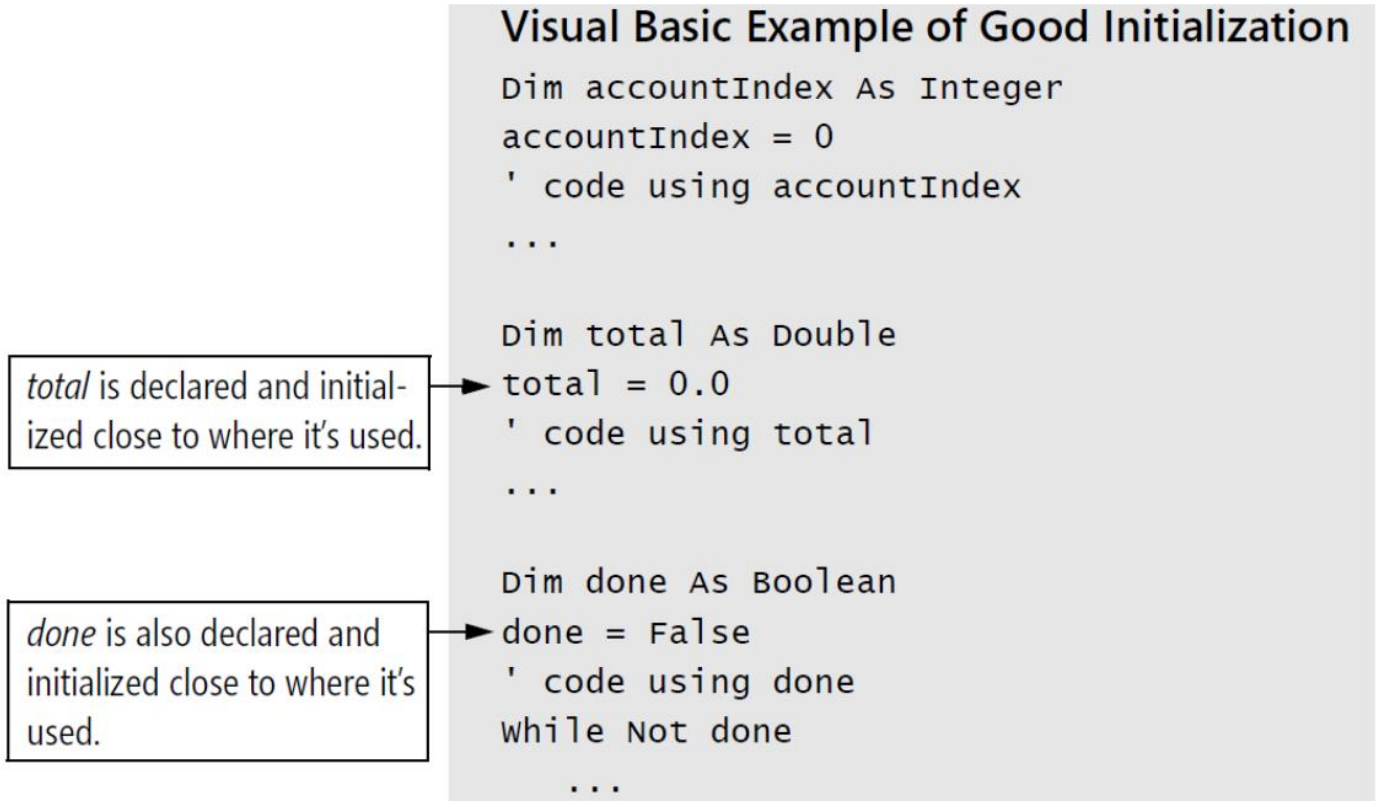
Visual Basic Example of Bad Initialization

```
' declare all variables
Dim accountIndex As Integer
Dim total As Double
Dim done As Boolean

' initialize all variables
accountIndex = 0
total = 0.0
done = False
...

' code using accountIndex
...

' code using total
...

' code using done
While Not done
   ...
```

Variable Initialization

Initialize each variable close to where it 's first used

- A better practice is to initialize variables as close as possible to where they're first used:



Variable Initialization

Initialize each variable close to where it 's first used

- By the time execution of the first example gets to the code that uses *done*, *done* could have been modified. If that's not the case when you first write the program, later modifications might make it so.
- Throwing all the initializations together creates the impression that all the variables are used throughout the whole routine—when in fact *done* is used only at the end.

Visual Basic Example of Bad Initialization

```
' declare all variables
Dim accountIndex As Integer
Dim total As Double
Dim done As Boolean

' initialize all variables
accountIndex = 0
total = 0.0
done = False
...

' code using accountIndex
...

' code using total
...

' code using done
While Not done
   ...
```

Variable Initialization

Ideally, declare and define each variable close to where it 's first used

- A declaration establishes a variable's type.
- A definition assigns the variable a specific value.
- In languages that support it, such as C++ and Java, variables should be declared and defined close to where they are first used.
- Ideally, each variable should be defined at the same time it's declared, as shown next:

Java Example of Good Initialization

```
int accountIndex = 0;
// code using accountIndex
...
```

Variable Initialization

Use `final` or `const` when possible

- By declaring a variable to be *final* in Java or *const* in C++, you can prevent the variable from being assigned a value after it's initialized.
- The *final* and *const* keywords are useful for *defining class constants*, *input-only parameters*, and *any local variables whose values are intended to remain unchanged* after initialization.

```
/*
 * Final variables can be declared using final keyword.
 * Once created and initialized, its value can not be changed.
 */
final int hoursInDay=24;

//This statement will not compile. Value can't be changed.
//hoursInDay=12;
```

Variable Initialization

Pay special attention to counters and accumulators

- The variables *i*, *j*, *k*, *sum*, and *total* are often counters or accumulators.
- A common error is **forgetting to reset a counter or an accumulator before the next time it's used**.

Variable Initialization

Initialize a class's member data in its constructor

- *Just as a routine's variables should be initialized within each routine, a class's data should be initialized within its constructor*.
- If memory is allocated in the constructor, it should be freed in the destructor.

```
// Fig. 3.10: GradeBook.java
// GradeBook class with a constructor to initialize the course name.

public class GradeBook
{
   private String courseName; // course name for this GradeBook

   // constructor initializes courseName with String argument
   public GradeBook( String name ) // constructor name is class name
   {
      courseName = name; // initializes courseName
   } // end constructor

} // end class GradeBook
```

Variable Initialization

Check the need for reinitialization

- Ask yourself whether the variable will ever need to be reinitialized, either because a loop in the routine uses the variable many times or because the variable retains its value between calls to the routine and needs to be reset between calls.
- *If it needs to be reinitialized, make sure that the initialization statement is inside the part of the code that's repeated*.

Variable Initialization

Check input parameters for validity

- Another valuable form of initialization is checking input parameters for validity. Before you assign input values to anything, make sure the values are reasonable.
- Remind you what? Assertion, Exception

Scope

- Scope is a way of thinking about a variable's celebrity status: how famous is it?
- A variable with limited or small scope is known in only a small area of a program.

```
// Counter-controlled repetition with the for repetition statement

public class ForCounter
{
   public static void main( String[] args )
   {
      // for statement header includes initialization,
      // loop-continuation condition and increment
      for ( int counter = 1; counter <= 10; counter++ )
         System.out.printf( "%d  ", counter );

      System.out.println(); // output a newline
   } // end main
} // end class ForCounter
```

Scope

- Different languages handle scope in different ways.
- In C++ and similar languages, a variable can be visible to a block (a section of code enclosed in curly brackets), a routine, a class (and possibly its derived classes), or the whole program (or a package).

Scope: *Guidelines*

Localize References to Variables

- The code between references to a variable is a "**window of vulnerability**."
- In the window, new code might be added, inadvertently altering the variable, or someone reading the code might forget the value the variable is supposed to contain.
- It's always a good idea to localize references to variables by keeping them close together.

Scope: *Guidelines*

Localize References to Variables

- One method of measuring how close together the references to a variable are is to compute the "span" of a variable. Here's an example:

Java Example of Variable Span

```
a = 0;
b = 0;
c = 0;
a = b + c;
```

- Two lines come between the first reference to *a* and the second, so *a* has a span of 2. One line comes between the two references to *b*, so *b* has a span of 1, and *c* has a span of 0.

Scope: *Guidelines*

Localize References to Variables

- The average span is computed by averaging the individual spans. For *b*, (1+0)/2 equals an average span of 0.5.

Java Example of Spans of One and Zero

```
a = 0;
b = 0;
c = 0;
b = a + 1;
b = b / c;
```

- When you keep references to variables close together, you enable the person reading your code to focus on one section at a time.
- If the references are far apart, you force the reader to jump around in the program. *Thus the main advantage of keeping references to variables together is that it improves program readability*.

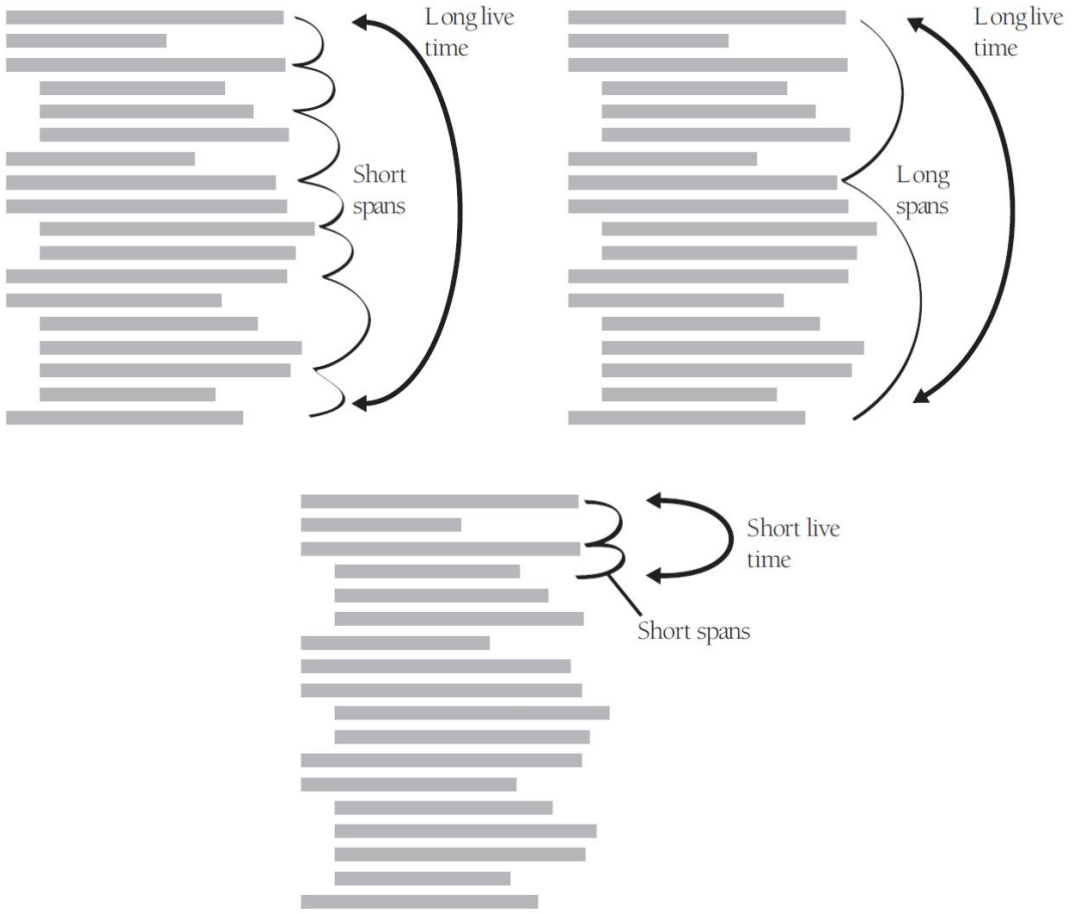
Scope: *Guidelines*

Keep Variables "Live" for as Short a Time as Possible

- A concept that's related to variable span is variable "live time", ***the total number of statements over which a variable is live***.
- **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.**
- Unlike span, live time isn't affected by how many times the variable is used between the first and last times it's referenced. If the variable is first referenced on line 1 and last referenced on line 25, it has a live time of 25 statements.

Scope: *Guidelines*

Keep Variables "Live" for as Short a Time as Possible



Scope: *Guidelines*

Keep Variables "Live" for as Short a Time as Possible

- As with span, the goal with respect to live time is to keep the number low, to keep a variable live for as short a time as possible.
- And as with span, the basic advantage of maintaining *a low number is that it reduces the* ***window of vulnerability***.
- You reduce the chance of incorrectly altering a variable between the places in which you intend to alter it.

Scope: *Guidelines*

Keep Variables "Live" for as Short a Time as Possible

- Keeping the live time short is that it gives you an accurate picture of your code.
- If a variable is assigned a value in line 10 and not used again until line 45, the very space between the two references implies that the variable is used between lines 10 and 45.
- If the variable is assigned a value in line 44 and used in line 45, no other uses of the variable are implied, and you can concentrate on a smaller section of code when you're thinking about that variable.

Scope: *Guidelines*

Keep Variables "Live" for as Short a Time as Possible

- A short live time makes 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.
- Likewise, 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.
- Short live times are useful when splitting a large routine into smaller routines.
- If references to variables are kept close together, it's easier to refactor related sections of code into routines of their own.

Scope: *Guidelines*

Measuring the Live Time of a Variable

- You can formalize the concept of live time by **counting the number of lines between the first and last references to a variable** (including both the first and last lines).

Java Example of Variables with Excessively 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;
```

Last reference to *recordIndex*. → line 28

Last reference to *total*. → line 69

Last reference to *done*. → line 70

recordIndex	(line 28 - line 2 + 1) = 27
total	(line 69 - line 3 + 1) = 67
done	(line 70 - line 4 + 1) = 67
Average Live Time	(27 + 67 + 67) / 3 ≈ 54

Scope: *Guidelines*

Measuring the Live Time of a Variable

Initialization of *recordIndex* is moved down from line 3.

Initialization of *total* and *done* are moved down from lines 4 and 5.

Java Example of Variables with Good, Short 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	(line 28 - line 25 + 1) = 4
total	(line 69 - line 62 + 1) = 8
done	(line 70 - line 63 + 1) = 8
Average Live Time	(4 + 8 + 8) / 3 ≈7

Scope: *Guidelines*

Measuring the Live Time of a Variable

- *Intuitively*, the second example seems better than the first because the initializations for the variables are performed closer to where the variables are used.
- **Avoid global variables!**

Java Example of Variables with Excessively 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;
```

Last reference to *recordIndex*.

Last reference to *total*.

Last reference to *done*.

Initialization of *recordIndex* is moved down from line 3.

Initialization of *total* and *done* are moved down from lines 4 and 5.

Java Example of Variables with Good, Short 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;
```

Scope: *Guidelines*

Initialize variables used in a loop immediately before the loop

- Doing this improves the chance that when you modify the loop, you'll **remember to make corresponding modifications to the loop initialization**.
- Later, when **you modify the program and put another loop around the initial loop**, the initialization will work on each pass through the new loop rather than on only the first pass.

Scope: *Guidelines*

Don't assign a value to a variable until just before the value is used

- You might have experienced the frustration of trying to figure out where a variable was assigned its value. The more you can do to clarify where a variable receives its value, the better.
- Languages like C++ and Java support variable initializations like these:

C++ Example of Good Variable Declarations and Initializations

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

Scope: *Guidelines*

Group related statements

- The following examples show a routine for summarizing daily receipts and illustrate how to put references to variables together so that they're easier to locate.

Statements using two sets of variables.

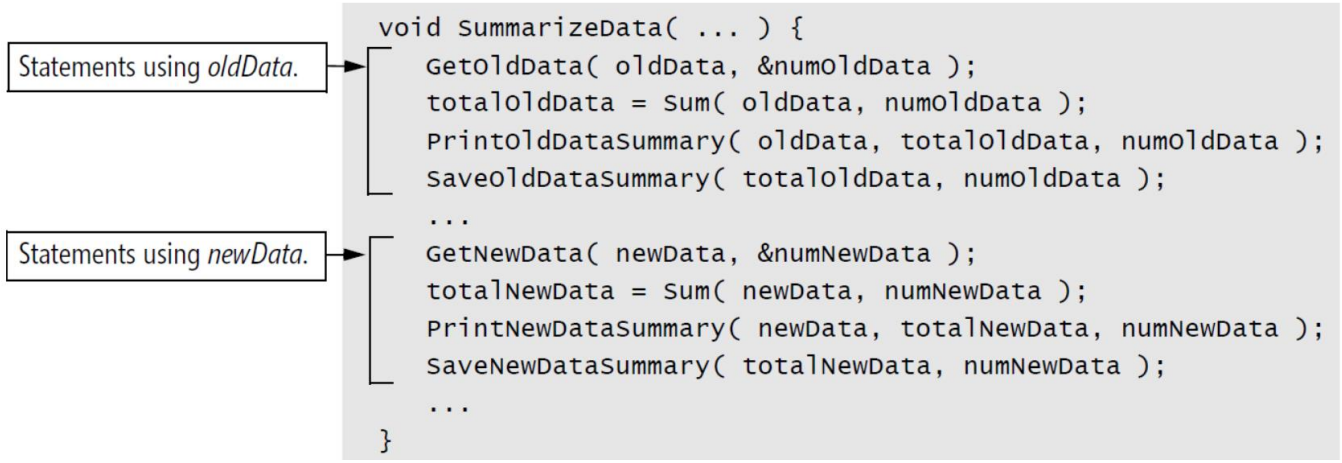
```
void SummarizeData(...) {
   ...
   GetOldData( oldData, &numOldData );
   GetNewData( newData, &numNewData );
   totalOldData = Sum( oldData, numOldData );
   totalNewData = Sum( newData, numNewData );
   PrintOldDataSummary( oldData, totalOldData, numOldData );
   PrintNewDataSummary( newData,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

Good or bad?

Scope: *Guidelines*

Group related statements

- Good or bad? **Good**: *more understandable, with groups*



- When the code is broken up, the two blocks are each shorter than the original block and individually contain fewer variables.
- They're easier to understand, and if you need to break this code out into separate routines, the shorter blocks with fewer variables will promote better-defined routines.

Scope: *Guidelines*

Break groups of related statements into separate routines

- All other things being equal, a variable in a shorter routine will tend to have smaller span and live time than a variable in a longer routine. By breaking related statements into separate, smaller routines, you reduce the scope that the variable can have - *Remember cohesion!*

Scope: *Guidelines*

Begin with most restricted visibility, and expand the variable's scope only if necessary

- Part of minimizing the scope of a variable is keeping **it as local as possible**.
- It is much more difficult to reduce the scope of a variable that has had a large scope than to expand the scope of a variable that has had a small scope!
- It's harder to turn a **protected data member into a private data member** than vice versa.
- For that reason, when in doubt, favor the smallest possible scope for a variable: local to a specific loop, local to an individual routine, then private to a class, then protected, then package (if your programming language supports that), and global only as a last resort.

One Variable = One Purpose

Use each variable for one purpose only!

- It's sometimes tempting to use one variable in two different places for two different activities.
- Usually, the variable is named inappropriately for one of its uses or a "temporary" variable is used in both cases

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;
```

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

One Variable = One Purpose

Use each variable for one purpose only!

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;
```

One Variable = One Purpose

Avoid variables with hidden meanings - Hybrid Coupling in variables

- Another way in which a variable can be used for more than one purpose is to have different values for the variable mean different things.
- For example:
  - The value in the variable *pageCount* might represent the number of pages printed, unless it equals *-1*, in which case it indicates that an error has occurred.
  - What are types of these two parameters?
  - *pageCount* normally indicates the number of pages; it's an integer.
  - When *pageCount* is *-1*, however, it indicates that an error has occurred; the integer is moonlighting as a boolean!

Variable Names

Outline

- Good Names
- Naming Specific Types of Data
- Naming Conventions
- Standardized Prefixes
- Creating Short Readable Names
- Names to Avoid

Good Names

A variable and its name are essentially the same thing

- Consequently, the goodness or badness of a variable is largely determined by its name. It should be: **Readable**, **Memorable**, and **Appropriate**!

Java Example of Poor Variable Names

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

Java Example of Good Variable Names

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

Good Names

The most important consideration in naming a variable is that the name fully and accurately describe the entity the variable represents

- **An effective technique for coming up with a good name is to state in words what the variable represents!**
- Often that statement itself is the best variable name.
  - It's easy to read because it doesn't contain cryptic abbreviations, and it's unambiguous.
  - It's a full description of the entity, so it won't be confused with something else.
  - It's easy to remember because the name is similar to the concept.

Good Names

- 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`.
- What do you think about these variable names?

Good Names

- Characteristics of these names?
  - Easy to decipher. Simply read them.
  - A bit long to be practical!

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, velocityInMph*	*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*

Good Names

A good name generally speaks to the problem rather than the solution.

- A good name tends to express the *what* more than the *how*.
- In general, if a name refers to some aspect of computing rather than to the problem, *it's a how rather than a what*.
- A record of employee data could be called `inputRec` or `employeeData`?
  - `inputRec` is a computer term—input and record.
  - `employeeData` refers to the problem domain rather than the computing universe.
- `bitFlag` *vs* `printerReady`?
  - Similarly, for a bit field indicating printer status, `bitFlag` is a more computer-ish name than `printerReady`. In an accounting application, `calcVal` is more computer-ish than `sum`.

Good Names: *Length*

- The optimum length for a name seems to be somewhere between the lengths of `x` and `maximumNumberOfPointsInModernOlympics`.
- Names that are too short don't convey enough meaning. The problem with names like *x1* and *x2* is that even if you can discover what *x* is, you won't know anything about the relationship between *x1* and *x2*.
- Names that are too long are hard to type and can harm the visual structure of a program – Auto Completion?

Good Names: *Length*

Variable names that are too long, too short and just right

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

Good Names: *Length*

- Are short variable names always bad?
  - No, not always.
- When you give a variable a short name like *i*, the length itself says something about the variable—namely, that the variable is a scratch value with a limited scope of operation.
- A programmer reading such a variable should be able to assume that its value isn't used outside a few lines of code.
- When you name a variable *i*, you're saying, "This variable is a loop counter or an array index and doesn't have any significance outside these few lines of code."

Good Names: *Length*

- Longer names are better for rarely used variables or global variables
- Shorter names are better for local variables or loop variables
- Short names are subject to many problems, however, and some careful programmers avoid them altogether as a matter of ***defensive-programming*** policy.

Good Names: *Qualifiers*

- Many programs have variables that contain computed values: totals, averages, maximums, and so on.
- If you modify a name with a qualifier like *Total*, *Sum*, *Average*, *Max*, *Min*, *Record*, *String*, or *Pointer*, put the modifier at the end of the name.
- This practice offers several advantages
  - The most significant part of the variable name, the part that gives the variable most of its meaning, is at the front, so it's most prominent and gets read first.
  - Avoid the confusion you might create if you were to use both *totalRevenue* and *revenueTotal* in the same program.
  - Set of names like *revenueTotal*, *expenseTotal*, *revenueAverage*, and *expenseAverage* has a pleasing symmetry.
  - The consistency improves readability and eases maintenance!

Good Names: *Qualifiers*

- `NumCustomers` vs `CustomerNum`?
- Exception: *Num*
  - Placed at the beginning of a variable name, *Num* refers to a total: *numCustomers* is the total number of customers.
  - Placed at the end of the variable name, *Num* refers to an index: *customerNum* is the number of the current customer.
  - The *s* at the end of *numCustomers* is another tip-off about the difference in meaning.
  - But, because using *Num* so often creates confusion, it's probably best to sidestep the whole issue by using *Count* or *Total* to refer to a total number of customers and *Index* to refer to a specific customer.
  - Thus, `customerCount` is the total number of customers and `customerIndex` refers to a specific customer.

Good Names: *Opposites*

Use opposites precisely.

- Using naming conventions for opposites helps consistency, which helps readability.
- Pairs like *begin/end* are easy to understand and remember.
- Pairs that depart from common-language opposites tend to be hard to remember and are therefore confusing.

- begin/end

- locked/unlocked
- min/max
- next/previous
- old/new
- opened/closed
- visible/invisible
- source/target
- source/destination
- up/down

Naming Specific Types of Data

- In addition to the general considerations in naming data, special considerations come up in the naming of specific kinds of data.
  - loop variables
  - status variables
  - temporary variables
  - boolean variables
  - enumerated types
  - constants

Naming Loop Indexes

- Guidelines for naming variables in loops have arisen because loops are such a common feature of computer programming. The names *i*, *j*, and *k* are customary:

Java Example of a Simple Loop Variable Name

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

Naming Loop Indexes

- 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 Loop Indexes

- If the loop is longer than a few lines, it's easy to forget what *i* is supposed to stand for and you're better off giving the loop index a more meaningful name.
- Because code is so often changed, expanded, and copied into other programs, many experienced programmers avoid names like *i* altogether.

Naming Loop Indexes

- One common reason loops grow longer is that they're nested.
- If you have several nested loops, assign longer names to the loop variables to improve readability.

Java Example of Good Loop Names in a Nested Loop

```
for ( teamIndex = 0; teamIndex < teamCount; teamIndex++ ) {
   for ( eventIndex = 0; eventIndex < eventCount[ teamIndex ]; eventIndex++ ) {
      score[ teamIndex ][ eventIndex ] = 0;
   }
}
```

Naming Loop Indexes

- Carefully chosen names for loop-index variables avoid the common problem of index cross-talk: saying *i* when you mean *j* and *j* when you mean *i*.
- They also make array accesses clearer:
  - `score[teamIndex][eventIndex]` is more informative than `score[i][j]`
- If you have to use *i*, *j*, and *k*, don't use them for anything other than loop indexes for simple loops—the convention is too well established, and breaking it to use them in other ways is confusing.
- The simplest way to avoid such problems is simply to *think of more descriptive names* than *i*, *j*, and *k*.

Naming Status Variables

Think of a better name than* flag *for status variables

- Status variables describe the state of your program.
- It's better to think of flags as status variables.
- A flag should never have *flag* in its name because that doesn't give you any clue about what the flag does.
- For clarity, flags should be assigned values and their values should be tested with enumerated types, named constants, or global variables that act as named constants.

Naming Status Variables

- Flags with good or bad names?

```
if ( flag ) ...
if ( statusFlag & 0x0F ) ...
if ( printFlag == 16 ) ...
if ( computeFlag == 0 ) ...

flag = 0x1;
statusFlag = 0x80;
printFlag = 16;
computeFlag = 0;
```

Naming Status Variables

- Statements like *statusFlag = 0x80* give you no clue about what the code does unless you wrote the code or have documentation that tells you both what *statusFlag* is and what *0x80* represents

```
if ( flag ) ...
if ( statusFlag & 0x0F ) ...
if ( printFlag == 16 ) ...
if ( computeFlag == 0 ) ...

flag = 0x1;
statusFlag = 0x80;
printFlag = 16;
computeFlag = 0;
```

C++ Examples of Better Use of Status Variables

```
if ( dataReady ) ...
if ( characterType & PRINTABLE_CHAR ) ...
if ( reportType == ReportType_Annual ) ...
if ( recalcNeeded = false ) ...

dataReady = true;
characterType = CONTROL_CHARACTER;
reportType = ReportType_Annual;
recalcNeeded = false;
```

Naming Status Variables

- Named constants and enumerated types to set up the values used in the example

Declaring Status Variables in C++

```
// values for CharacterType
const int LETTER = 0x01;
const int DIGIT = 0x02;
const int PUNCTUATION = 0x04;
const int LINE_DRAW = 0x08;
const int PRINTABLE_CHAR = ( LETTER | DIGIT | PUNCTUATION | LINE_DRAW );

const int CONTROL_CHARACTER = 0x80;

// values for ReportType
enum ReportType {
   ReportType_Daily,
   ReportType_Monthly,
   ReportType_Quarterly,
   ReportType_Annual,
   ReportType_All
};
```

Naming Enumerated Types

- When you use an enumerated type, you can ensure that it's clear that members of the type all belong to the same group by using a group prefix, such as *Color_*, *Planet_*, or *Month_*.
- Any comments?

```
Public Enum Color
    Color_Red
    Color_Green
    Color_Blue
End Enum

Public Enum Planet
    Planet_Earth
    Planet_Mars
    Planet_Venus
End Enum

Public Enum Month
    Month_January
    Month_February
    ...
    Month_December
End Enum
```

Naming Temporary Variables

- Temporary variables are used to hold intermediate results of calculations, as temporary placeholders, and to hold housekeeping values.
- They're usually called *temp*, *x*, or some other vague and non-descriptive name.
- In general, temporary variables are a sign that the programmer does not yet fully understand the problem.
- Moreover, because the variables are officially given a "temporary" status, programmers tend to treat them more casually than other variables, *increasing the chance of errors*.

Naming Temporary Variables

Be cautious of "temporary" variables

- It's often necessary to preserve values temporarily.
- But in one way or another, most of the variables in your program are temporary. Calling a few of them temporary may indicate that you aren't sure of their real purposes. Consider the following example:

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

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

Naming Temporary Variables

Be cautious of "temporary" variables

- It's fine to store the value of the expression *sqrt(b^2 - 4 * a * c)* in a variable, especially since it's used in two places later. But the name *temp* doesn't tell you anything about what the variable does.
- A better approach is shown in this example:

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

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

Naming Boolean Variables

Keep typical boolean names in mind

- done
- Use done to indicate whether something is done.
- The variable can indicate whether a loop is done or some other operation is done. Set done to false before something is done, and set it to true when something is completed.

Naming Boolean Variables

Keep typical boolean names in mind

- error
- Use error to indicate that an error has occurred.
- Set the variable to false when no error has occurred and to true when an error has occurred.

Naming Boolean Variables

Keep typical boolean names in mind

- found
- Use found to indicate whether a value has been found.
- Set found to false when the value has not been found and to true once the value has been found. Use found when searching an array for a value, a file for an employee ID, a list of paychecks for a certain paycheck amount, and so on.

Naming Boolean Variables

Keep typical boolean names in mind

- success or ok
- Use success or ok to indicate whether an operation has been successful.
- Set the variable to false when an operation has failed and to true when an operation has succeeded. If you can, replace success with a more specific name that describes precisely what it means to be successful. If the program is successful when processing is complete, you might use `processingComplete` instead. *If the program is successful when a value is found, you might use found instead*.

Naming Boolean Variables

Give boolean variables names that imply true or false

- Names like *done* and *success* are good boolean names because the state is either *true* or *false*; something is done or it isn't; it's a success or it isn't.
- Names like *status* and *sourceFile*, on the other hand, are poor boolean names because they're not obviously *true* or *false*.
- For better results, replace *status* with a name like *error* or *statusOK*, and replace *sourceFile* with *sourceFileAvailable* or *sourceFileFound*, or whatever the variable represents.

Naming Boolean Variables

Give boolean variables names that imply true or false

- Some programmers like to put *is* in front of their boolean names.
- Then the variable name becomes a question: *isdone*? *isError*? *isFound*? *isProcessingComplete*? Answering the question with *true* or *false* provides the value of the variable.
- It won't work with vague names: *isStatus*? makes no sense at all.

Naming Boolean Variables

Use positive boolean variable names

- Negative names like *notFound*, *notdone*, and *notSuccessful* are difficult to read when they are negated—for example,

```
if not notFound
```

- Such a name should be replaced by *found*, *done*, or *processingComplete* and then negated with an operator as appropriate.
- If what you're looking for is found, you have *found* instead of *not notFound*.

Naming Constants

- When naming constants, name the abstract entity the constant represents rather than the number the constant refers to.
- *FIVE* is a bad name for a constant (regardless of whether the value it represents is *5.0*). *CYCLES_NEEDED* is a good name.
- *CYCLES_NEEDED* can equal *5.0* or *6.0*.
- *FIVE = 6.0* would be ridiculous.

Naming Conventions

- The key is that any convention is often better than no convention.
- The convention may be arbitrary. The power of naming conventions doesn't come from the specific convention chosen but from the fact that a convention exists, adding structure to the code and giving you fewer things to worry about.

Naming Conventions

Why Have Conventions?

- *They let you take more for granted*.
- By making one global decision rather than many local ones, you can concentrate on the more important characteristics of the code.

Naming Conventions

Why Have Conventions?

- *They help you transfer knowledge across projects*.
- Similarities in names give you an easier and more confident understanding of what unfamiliar variables are supposed to do.

Naming Conventions

Why Have Conventions?

- *They help you learn code more quickly on a new project*.
- Rather than learning that Anita's code looks like this, Julia's like that, and Kristin's like something else, you can work with a more consistent set of code.

Naming Conventions

Why Have Conventions?

- Without naming conventions, you can easily call the same thing by two different names.
- ***For example***, you might call total points both *pointTotal* and *totalPoints*. This might not be confusing to you when you write the code, but it can be enormously confusing to a new programmer who reads it later.

Naming Conventions

Why Have Conventions?

- *They compensate for language weaknesses*.
- You can use conventions to emulate named constants and enumerated types. The conventions can differentiate among local, class, and global data and can incorporate type information for types that aren't supported by the compiler.

Naming Conventions

Why Have Conventions?

- *They emphasize relationships among related items*. If you use object data, the compiler takes care of this automatically. If your language doesn't support objects, you can supplement it with a naming convention.
- Names like *address*, *phone*, and *name* don't indicate that the variables are related. But suppose you decide that all employee-data variables should begin with an *Employee* prefix. *employeeAddress*, *employeePhone*, and *employeeName* leave no doubt that the variables are related.
- Programming conventions can make up for the weakness of the language you're using.

Naming Conventions

Degrees of Formality

- Different conventions have different degrees of formality. An informal convention might be as simple as "Use meaningful names."
- In general, the degree of formality you need is dependent on the number of people working on a program, the size of the program, and the program's expected life span.
- On tiny projects, a strict convention might be unnecessary overhead.
- On larger projects in which several people are involved, formal conventions are extremely practical for readability

Naming Conventions

Differentiate between variable names and routine names

- The convention the course book uses is to begin variable and object names with lower case and routine names with upper case: *variableName* vs. *RoutineName()*.

Naming Conventions (Informal)

Differentiate between classes and objects

- The correspondence between class names and object names—or between types and variables of those types—can get tricky.
- Several standard options exist, as shown in the following examples:

Option 1: Differentiating Types and Variables via Initial Capitalization

```
Widget widget;
LongerWidget longerWidget;
```

Option 2: Differentiating Types and Variables via All Caps

```
WIDGET widget;
LONGERWIDGET longerWidget
```

Option 3: Differentiating Types and Variables via the “t_” Prefix for Types

```
t_Widget Widget;
t_LongerWidget LongerWidget;
```

Option 4: Differentiating Types and Variables via the “a” Prefix for Variables

```
Widget aWidget;
LongerWidget aLongerWidget;
```

Option 5: Differentiating Types and Variables via Using More Specific Names for the Variables

```
Widget employeeWidget;
LongerWidget fullEmployeeWidget;
```

Naming Conventions (Informal)

Identify global variables

- *One common programming problem is misuse of global variables*.
- If you give all global variable names a *g_* prefix, for example, a programmer seeing the variable *g_RunningTotal* will know it's a global variable and treat it as such.

Naming Conventions (Informal)

Identify member variables

- Identify a class's member data.
- Make it clear that the variable isn't a local variable and that it isn't a global variable either.
- For example, you can identify class member variables with an *m_* prefix to indicate that it is member data.

Naming Conventions (Informal)

Identify named constants

- Named constants need to be identified so that you can tell whether you're assigning a variable a value from another variable (whose value might change) or from a named constant.
- One approach to naming constants is to use a prefix like *c_* for constant names.
- That would give you names like *c_RecsMax* or *c_LinesPerPageMax*. *In C++ and Java, the convention is to use all uppercase letters*, possibly with underscores to separate words, *RECSMAX* or *RECS_ MAX* and *LINESPERPAGEMAX* or *LINES_PER_PAGE_ MAX*.

Naming Conventions (Informal)

Identify elements of enumerated types

- Elements of enumerated types need to be identified for the same reasons that named constants do—to make it easy to tell that the name is for an enumerated type as opposed to a variable, named constant, or function.
- The standard approach applies: you can use all caps or an *e_* or *E_* prefix for the name of the type itself and use a prefix based on the specific type like *Color_* or *Planet_* for the members of the type.

Naming Conventions (Informal)

Format names to enhance readability

- Two common techniques for increasing readability are using capitalization and spacing characters to separate words.
- For example, *GYMNASTICSPOINTTOTAL* is less readable than *gymnasticsPointTotal* or *gymnastics_point_total*.
- C++, Java, Visual Basic, and other languages allow for mixed uppercase and lowercase characters. C++, Java, Visual Basic, and other languages also allow the use of the underscore (_) separator.

Table 11-3 Sample Naming Conventions for C++ and Java

Entity	Description
ClassName	Class names are in mixed uppercase and lowercase with an initial capital letter.
TypeName	Type definitions, including enumerated types and typedefs, use mixed uppercase and lowercase with an initial capital letter.
EnumeratedTypes	In addition to the rule above, enumerated types are always stated in the plural form.
localVariable	Local variables are in mixed uppercase and lowercase with an initial lowercase letter. The name should be independent of the underlying data type and should refer to whatever the variable represents.
routineParameter	Routine parameters are formatted the same as local variables.
RoutineName()	Routines are in mixed uppercase and lowercase. (Good routine names are discussed in Section 7.3.)
m_ClassVariable	Member variables that are available to multiple routines within a class, but only within a class, are prefixed with an *m_*.
g_GlobalVariable	Global variables are prefixed with a *g_*.
CONSTANT	Named constants are in *ALL_CAPS*.
MACRO	Macros are in *ALL_CAPS*.
Base_EnumeratedType	Enumerated types are prefixed with a mnemonic for their base type stated in the singular—for example, *Color_Red*, *Color_Blue*.

Creating Short Readable Names

- Use standard abbreviations (the ones in common use, which are listed in a dictionary).
- Remove all nonleading vowels. (computer becomes cmptr, and screen becomes scrn, and integer becomes intgr.)
- Remove articles: and, or, the, and so on.
- Use the first letter or first few letters of each word.
- Truncate consistently after the first, second, or third (whichever is appropriate) letter of each word.
- Keep the first and last letters of each word.
- Use every significant word in the name, up to a maximum of three words.
- Remove useless suffixes—ing, ed, and so on.
- Keep the most noticeable sound in each syllable.
- Be sure not to change the meaning of the variable.
- Iterate through these techniques until you abbreviate each variable name to between 8 to 20 characters or the number of characters to which your language limits variable names.

Creating Short Readable Names

Phonetic Abbreviations

- Some people advocate creating abbreviations based on the sound of the words rather than their spelling.
- Thus *skating* becomes *sk8ing*, *highlight* becomes *hilite*, *before* becomes *b4*, *execute* becomes *xqt*, and so on.
- This seems too much like asking people to figure out. Preferable avoid them.

Creating Short Readable Names

Don’t abbreviate by removing one character from a word

- Typing one character is little extra work, and the one-character savings hardly justifies the loss in readability.
- It’s like the calendars that have “Jun” and “Jul.” You have to be in a big hurry to spell June as “Jun.” With most one-letter deletions, it’s hard to remember whether you removed the character. Either remove more than one character or spell out the word.

Creating Short Readable Names

Create names that you can pronounce

- Use *xPos* rather than *xPstn* and *needsComp* rather than *ndsCmptg*.
- Apply the telephone test—if you can't read your code to someone over the phone, rename your variables to be more distinctive (Kernighan and Plauger 1978).

Creating Short Readable Names

Avoid combinations that result in misreading or mispronunciation

- To refer to the end of *B*, favor *ENDB* over *BEND*.
- If you use a good separation technique, you won't need this guideline since *B-END*, *BEnd*, or *b_end* won't be mispronounced.

Creating Short Readable Names

Use a thesaurus to resolve naming collisions

- One problem in creating short names is naming collisions—names that abbreviate to the same thing.
- **For example**, if you're limited to three characters and you need to use *fired* and *full revenue disbursal* in the same area of a program, you might inadvertently abbreviate both to *frd*.
- One easy way to avoid naming collisions is to use a different word with the same meaning, so a thesaurus is handy.
- *Complete revenue disbursal* might be substituted for *full revenue disbursal*.

Creating Short Readable Names

Document extremely short names with translation tables in the code

- In languages that allow only very short names, include a translation table to provide a reminder of the mnemonic content of the variables. Include the table as comments at the beginning of a block of code. Here's an example:

Fortran Example of a Good Translation Table

```
C ***********************************************************************
C    Translation Table
C
C    Variable    Meaning
C    --------    -------
C    XPOS        x-Coordinate Position (in meters)
C    YPOS        Y-Coordinate Position (in meters)
C    NDSCMP      Needs Computing (=0 if no computation is needed;
C                                 =1 if computation is needed)
C    PTGTTL      Point Grand Total
C    PTVLMX      Point Value Maximum
C    PSCRMX      Possible Score Maximum
C ***********************************************************************
```

Creating Short Readable Names

Document all abbreviations in a project-level "Standard Abbreviations" document

- Abbreviations in code create two general risks:
  - A reader of the code might not understand the abbreviation.
  - Other programmers might use multiple abbreviations to refer to the same word, which creates needless confusion.
- To address both these potential problems, you can create a "Standard Abbreviations" document that captures all the coding abbreviations used on your project. The document can be a word processor document or a spreadsheet. On a very large project, it could be a database.

Creating Short Readable Names

Remember that names matter more to the reader of the code than to the writer

- Read code of your own that you haven’t seen for at least six months and notice where you have to work to understand what the names mean. Resolve to change the practices that cause such confusion.

Names to Avoid

Avoid misleading names or abbreviations

- Make sure that a name is clear.
- For example, *FALSE* is usually the opposite of *TRUE* and would be a bad abbreviation for “Fig and Almond Season.”

Names to Avoid

Avoid names with similar meanings

- If you can switch the names of two variables without hurting the program, you need to rename both variables.
- For example, *input* and *inputValue*, *recordNum* and *numRecords*, and *fileNumber* and *fileIndex* are so semantically similar that if you use them in the same piece of code you'll easily confuse them and install some subtle, hard-to-find errors.

Names to Avoid

Avoid variables with different meanings but similar names

- If you have two variables with similar names and different meanings, try to rename one of them or change your abbreviations.
- Avoid names like *clientRecs* and *clientReps*. They're only one letter different from each other, and the letter is hard to notice. *Have at least two-letter differences between names, or put the differences at the beginning or at the end*.
- *clientRecords* and *clientReports* are better than the original names.

Names to Avoid

Avoid names that sound similar, such as wrap and rap yet different

- You end up having conversations like this:

I was just speaking with the Goal Donor–

Did you say "Gold Owner" or "Goal Donor"?

I said "Goal Donor."

What?

GOAL - - - DONOR!

OK, Goal Donor. You don't have to yell, Goll' Darn it.

Did you say "Gold Donut?"

Names to Avoid

Avoid numerals in names

- If the numerals in a name are really significant, use an array instead of separate variables. If an array is inappropriate, numerals are even more inappropriate.
- *For example*, avoid *file1* and *file2*, or *total1* and *total2*.
- You can almost always think of a better way to differentiate between two variables than by tacking a *1* or a *2* onto the end of the name.
- Some real-world entities (such as Route 66 or Interstate 405) have numerals embedded in them. But consider whether there are better alternatives before you create a name that includes numerals.

Names to Avoid

Avoid misspelled words in names

- It’s hard enough to remember how words are supposed to be spelled.
- To require people to remember “correct” misspellings is simply too much to ask.
- ***For example***, misspelling *highlight* as *hilite* to save three characters makes it devilishly difficult for a reader to remember how *highlight* was misspelled.
- Was it *highlite*? *hilite*? *hilight*? *hilit*? *jai-a-lai-t*? Who knows?

Names to Avoid

Avoid words that are commonly misspelled in English

- *Absense*, *acummulate*, *acsend*, *calender*, *concieve*, *defferred*, *definate*, *independance*, *occassionally*, *prefered*, *reciept*, *superseed*, and many others are common misspellings in English.

Names to Avoid

Don’t differentiate variable names solely by capitalization

- If you’re programming in a case-sensitive language such as C++, you may be tempted to use *frd* for *fired*, *FRD* for *final review duty*, and *Frd* for *full revenue disbursal*. Avoid this practice.
- Although the names are unique, the association of each with a particular meaning is arbitrary and confusing. *Frd* could just as easily be associated with *final review duty* and *FRD* with *full revenue disbursal*, and no logical rule will help you or anyone else to remember which is which.

Names to Avoid

Avoid multiple natural languages

- In multinational projects, enforce use of a single natural language for all code, including class names, variable names, and so on.
- Reading another programmer's code can be a challenge; reading another programmer's code in Southeast Martian is impossible.
- *A more subtle problem occurs in variations of English*. If a project is conducted in multiple English-speaking countries, standardize on one version of English so that you're not constantly wondering whether the code should say "color" or "colour," "check" or "cheque," and so on.

Names to Avoid

Don’t use names that are totally unrelated to what the variables represent

- Sprinkling names such as *margaret* and *pookie* throughout your program virtually guarantees that no one else will be able to understand it.
- *Avoid using personal, special and unrelated names*.

Names to Avoid

Avoid names containing hard-to-read characters

- Be aware that some characters look so similar that it's hard to tell them apart. If the only difference between two names is one of these characters, you might have a hard time telling the names apart.

eyeChartl	eyeChartI	eyeChartl
TTLCONFUSION	TTLCONFUSION	TTLC0NFUSION
hard2Read	hardZRead	hard2Read
GRANDTOTAL	GRANDTOTAL	6RANDTOTAL
ttl5	ttlS	ttlS

つづく