Проектиране и Тестиране на Софтуер
ТУ, кат. КС, летен семестър 2012

Лекция 1b

Тема:

Проектиране и Работа с Подпрограми (част 2)

Съдържание:

- Еволюция на концепцията ПП
- Проектиране и програмиране на ефикасни, сигурни, надеждни и качествени ПП

ПП

Еволюция на концепцията ПП

Развитие на ПП в C/C++

- Функции, връщащи псевдоним Functions returning reference
- Предефинирани функции Overloaded functions
- Функции с подразбиращи се стойности на аргументите Default-argument functions
- Вградени функции Inline functions
- Взаимоотношение (relation) функция-макрос
- Първични функции Generic functions

Функции, връщащи псевдоним

Наричат се псевдофункции. Позволено е да се ползват от двете страни на оператор за присвояване:

отляво:

```
setx() = <израз>
```

отдясно:

```
y = …setx()… - операнд в израз
```

отляво и отдясно:

```
setx() = …setx()…
```

Функции, връщащи псевдоним

```
 int x;
 int& setx();

void main() { setx() = 218; cout << x; }

 int& setx()
 {
   return x;
 }
```

Функции, връщащи псевдоним

```
#include <iostream>
using namespace std;
int x; int& setx();
void main()
{
  int y;
  y = setx();    cout << '\n' << y;
  setx() = 218; cout << '\n' << x;
  y = setx();    cout << '\n' << y;
  setx() = setx() + 12; cout<<‘\n’x<<y;
}
int& setx() {    return x; }
```

Предефинирани функции/методи

```
C++:    Yes

C#:     Yes

Java:   Yes
```

Предефинирани функции

```
void prch();
void prch(char);
void prch(int);
void prch(char, int);
```

Предефинирани функции

```
void prch()
{ cout<<‘\n’; for(int i=0;i<80;i++) cout<<‘*’;}

//-----------------------------------
```

Предефинирани функции

```
void prch()
{ cout<<'\n'; for(int i=0;i<80;i++) cout<<'*';}

//------------------------------------------
 void prch(char ch)
{ cout<<'\n'; for(int i=0;i<80;i++) cout<<ch;}

//------------------------------------------
```

Предефинирани функции

```
void prch()
{ cout<<‘\n’; for(int i=0;i<80;i++) cout<<‘*’;}

//--------------------------------------
 void prch(char ch)
{ cout<<‘\n’; for(int i=0;i<80;i++) cout<<ch;}

//--------------------------------------
 void prch(int n)
{ cout<<‘\n’; for(int i=0;i<n;i++) cout<<‘*’;}

//--------------------------------------
```

Предефинирани функции

```
void prch()
{ cout<<'\n'; for(int i=0;i<80;i++) cout<<'*';}

//-------------------------------------------
 void prch(char ch)
{ cout<<'\n'; for(int i=0;i<80;i++) cout<<ch;}

//-------------------------------------------
 void prch(int n)
{ cout<<'\n'; for(int i=0;i<n;i++) cout<<'*';}

//-------------------------------------------
 void prch(char ch, int n)
{ cout<<'\n'; for(int i=0;i<n;i++) cout<<ch;}
```

Нееднозначно активиране

Sometimes there may be two or more possible matches for an invocation of a method, but the compiler cannot determine the most specific match. This is referred to as *ambiguous invocation*. Ambiguous invocation is a compilation error.

Нееднозначно активиране

```
public class AmbiguousOverloading {
  public static void main(String[] args) {
    System.out.println(max(1, 2));
  }

  public static double max(int num1, double num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }

  public static double max(double num1, int num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
}
```

Learning About Ambiguity (continued)

- Overload methods
  - Correctly provide different argument lists for methods with same name
- Illegal methods
  - Methods with identical names that have identical argument lists but different return types
  - `int aMethod(int x)`
  - `void aMethod(int x)`

Функции/методи с подразбиращи се стойности на аргументите

C++:	Yes
C#:	Yes
Java:	No

Функции с подразбиращи се стойности на аргументите

C++, Fortran95, Ada, PHP: формалните параметри имат стойности по подразбиране

```
 float ave(int=20, int=30, int=40);
//--------------------------------------------------
main() {
   cout  <<  '\n'  <<     ave();
   cout  <<  '\n'  <<     ave(100);
   cout  <<  '\n'  <<     ave(100, 200);
   cout  <<  '\n'  <<     ave(5, 6, 8);
     }
//--------------------------------------------------
float ave(int x, int y, int z)
   {
      return (x+y+z)/3.;
   }
```

C# Функции/методи с подразбиращи се стойности на аргументите

```
class Program
{
 static double ave(int a=20, int b=30, int c=40)
  {
    return (a+b+c)/3.0;
  }

 static void Main(string[] args)
  {
    Console.WriteLine("ave={0}  {1}  {2} {3}",
   ave(),ave(100),ave(100,200),ave(100,200,300));
  }
}
```

Вградени функции

```
inline float area(float r) {return PI*r*r;}
```

Взаимоотношение функция-макрос

```
#define AREA(x)        PI*x*x


// по-надеждно
#define AREA(x)        PI*(x)*(x)


// най-надеждно
#define AREA(x)        (PI*(x)*(x))
```

Първични, родови функции

```
template <class Type>
Type max (Type a, Type b)
  {  return (a>b)? a: b; }


int x=20, y=30;          cout << max(x, y);
float p=3., q=5.;        cout << max(p,q);
```

ПП

Проектиране и програмиране на сигурни, надеждни и качествени ПП

Защо ПП?

Причини за работа с ПП

- **Намалява сложността**
  - Това е една от главните причини. Top-down design.
  - “Properly designed functions permit to ignore **how** a job’s done. Knowing **what** is done is sufficient.”
  - “A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. ”

B.Kernighan & D.Ritchie

- **Избягва дублиране на код**
  - ПП се създават, за да бъдат извиквани многократно.
  - Това е може би най-популярната причина.

Създава четлив и разбираем код

- Създава четлив и разбираем код

```
If (Node != NULL) then
   While (Node.Next!=NULL) do
     Node = Node.Next;
     LeafName = Node.Name;
  end while
else
 LeafName = “empty”;
Endif
==================================
Leafname = GetLeafName(Node);
```

Създава четлив и разбираем код

```
res=1;
for (int I=1;I<=N;I++)
  res = res * I;
=================================
res = fact(N);

++++++++++++++++++++++++++++++++++++++++++

int m, n, r;
while (  (r=m%n) != 0 )
  {
      m=n; n=r;
  }
=================================
res = gcd(m, n);
```

Още Причини за работа с ПП

- Опростява сложни логически изрази
  - Среща се дефиниция на функция, която се вика веднъж, само защото прави по-ясен сегмент от първичен код.

```
isalpha(c)
   c>='a' && c<='z' || c>='A' && c<='Z'

leapyear(y)
   y%4 == 0 && y%100 != 0 || y%400 == 0
```

- Скрива опериране с указатели
- Подобрява преносимостта
- Подобрява х-ки на продукта по памет

Х-ки на качествени ПП

- What is a high-quality routine?
- Design and implementation of efficient, secure and reliable routines
  - Criteria
  - Requirements

Steve McConnell

CODE COMPLETE

Chapter 7

2nd Edition, 2005

Качествени ПП

- What is a high-quality routine?
  - This is a harder question?
- It is easier to show what a high quality routine is not
  - See next slide

C++ пример на ПП с ниско качество

```
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[ crtQtr ][ 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];
    }
```

C++ пример на ПП с ниско качество

What’s wrong with this routine?

You are expected to try to find or register or formulate problems with it (3-5 min).

Once you’ve come up with your own list, look at the list with criticisms presented on next page.

C++ пример на ПП с ниско качество

- The routine has a bad name. HandleStuff tells you nothing about the routine purpose.
- The routine isn’t documented
- The routine has a bad layout (white spaces and indentation)
- The routine’s input variable *InputRec*, is changed. If it’s input variable, its value should not be modified and its name should not be like *InputRec*
- The routine reads and writes global variables. It read from *corpExpense* and writes to *profit*. It should communicate with other routines more directly than by reading/writing global data.
- The routine doesn’t have a single purpose. One routine, one task. It initializes some variables, writes to a data base, does some calculations – none of which seems to relate to each other in any way. A routine should have a single, clearly defined purpose.
- The routine doesn’t defend itself against bad data. Attention – division by zero error is possible
- The routine uses magic numbers (100, 12, 4.0, 2 ,3) instead symbolic constants
- The routine uses only two fields of the CORP_DATA type of parameter. No need to transfer all the structure as a parameter.
- Some of the routine’s parameters aren’t used. ScreenX and ScreenY are not referenced within the routine.
- One of the routine’s parameters is passed incorrectly: prevColor is labeled as a reference (&) parameter even though it isn’t assigned a value within the routine
- The routine has too many parameters. Recommended up to 7. This routine has 11 parameters.
- Routine’s parameters are poorly ordered and not documented.

Проектиране на ПП

Design at the Routine level

Design at the Routine level

- Важни критерии/изисквания при проектиране на ПП:
- Coupling /връзки м/у ПП/
  - Loosely coupled routines
- Cohesion /вътрешна структура на ПП/
  - Strongly cohesive routines

Coupling: **Keep Coupling Loose**

- Coupling describes how tightly a routine is related to other routines. The goal is to create routines with small, direct, visible, and flexible relations to other routines, which is known as "loose coupling."
- Good coupling between routines is loose enough that one routine can easily be used by other routines. Model railroad cars are coupled by opposing hooks that latch when pushed together. Connecting two cars is easy—you just push the cars together. Imagine how much more difficult it would be if you had to screw things together, or connect a set of wires, or if you could connect only certain kinds of cars to certain other kinds of cars. The coupling of model railroad cars works because it's as simple as possible.
- In software, make the connections among routines as simple as possible.

Coupling

- Try to create routines that depend little on other routines. Make them detached, as business associates are, rather than attached, as Siamese twins are.
- A routine like **`sin()`** is loosely coupled because everything it needs to know is passed in to it with one value representing an angle in degrees.
- A routine **`InitVars(var1,var2,var3,…,varN)`** is more tightly coupled because, with all the variables it must pass, the calling routine practically knows what is happening inside **`InitVars()`**.

Cohesion

- For routines, cohesion refers to how closely the operations in a routine are related.
- Some programmers prefer the term "strength": how strongly related are the operations in a routine?
- A function like **`Cosine()`** is perfectly cohesive because the whole routine is dedicated to performing one function.
- A function like **`CosineAndTan()`** has lower cohesion because it tries to do more than one thing. The goal is to have each routine do one thing well and not do anything else

Cohesion

- Discussions about cohesion typically refer to several levels of cohesion

Cohesion

- **Functional cohesion** is the strongest and best kind of cohesion, occurring when a routine performs one and only one operation. Examples of highly cohesive routines include **`sin()`**, **`GetCustomerName()`**, **`EraseFile()`**, **`CalculateLoanPayment()`**, **`and`** **`AgeFromBirthdate()`**.
- Of course, this evaluation of their cohesion assumes that the routines do what their names say they do—if they do anything else, they are less cohesive and poorly named

Cohesion

- **Sequential cohesion** exists when a routine contains operations that must be performed in a specific order, that share data from step to step, and that don't make up a complete function when done together.
- An example of sequential cohesion is a routine that, given a birth date, calculates an employee's age and time to retirement. If the routine calculates the age and then uses that result to calculate the employee's time to retirement, it has sequential cohesion. If the routine calculates the age and then calculates the time to retirement in a completely separate computation that happens to use the same birth-date data, it has only communicational cohesion.
- How would you make the routine functionally cohesive? You'd create separate routines to compute an employee's age given a birth date and compute time to retirement given a birth date. The time-to-retirement routine could call the age routine. They'd both have functional cohesion.

Cohesion

- **Communicational cohesion** occurs when operations in a routine make use of the same data and aren't related in any other way. If a routine prints a summary report and then reinitializes the summary data passed into it, the routine has communicational cohesion: the two operations are related only by the fact that they use the same data.
- To give this routine better cohesion, the summary data should be reinitialized close to where it's created, which shouldn't be in the report-printing routine. Split the operations into individual routines. The first prints the report. The second reinitializes the data, close to the code that creates or modifies the data. Call both routines from the higher-level routine that originally called the communicationally cohesive routine

Cohesion

- The remaining kinds of cohesion are generally unacceptable. They result in code that's poorly organized, hard to debug, and hard to modify

Cohesion

- **Procedural cohesion** occurs when operations in a routine are done in a specified order. An example is a routine that gets an employee name, then an address, and then a phone number. The order of these operations is important only because it matches the order in which the user is asked for the data on the input screen. Another routine gets the rest of the employee data. The routine has procedural cohesion because it puts a set of operations in a specified order and the operations don't need to be combined for any other reason.
- To achieve better cohesion, put the separate operations into their own routines. Make sure that the calling routine has a single, complete job: **`GetEmployee()`** rather than **`GetFirstPartOfEmployeeData()`**. You'll probably need to modify the routines that get the rest of the data too. It's common to modify two or more original routines before you achieve functional cohesion in any of them.

Cohesion

- **Logical cohesion** occurs when several operations are stuffed into the same routine and one of the operations is selected by a control flag that's passed in. It's called logical cohesion because the control flow or "logic" of the routine is the only thing that ties the operations together—they're all in a big if statement or case statement together. It isn't because the operations are logically related in any other sense. Considering that the defining attribute of logical cohesion is that the operations are unrelated, a better name might "illogical cohesion."
- One example would be an **`InputAll()`** routine that inputs customer names, employee timecard information, or inventory data depending on a flag passed to the routine. Other examples would be **`ComputeAll(), EditAll(), PrintAll(), and SaveAll().`** The main problem with such routines is that you shouldn't need to pass in a flag to control another routine's processing. Instead of having a routine that does one of three distinct operations, depending on a flag passed to it, it's cleaner to have three routines, each of which does one distinct operation. If the operations use some of the same code or share data, the code should be moved into a lower-level routine and the routines should be packaged into a class.
- It's usually all right, however, to create a logically cohesive routine if its code consists solely of a series of if or case statements and calls to other routines. In such a case, if the routine's only function is to dispatch commands and it doesn't do any of the processing itself, that's usually a good design. The technical term for this kind of routine is "event handler."

Cohesion

- **Coincidental cohesion** occurs when the operations in a routine have no discernible relationship to each other. Other good names are "no cohesion" or "chaotic cohesion." The low-quality C++ routine at the beginning of this chapter had coincidental cohesion. It's hard to convert coincidental cohesion to any better kind of cohesion—you usually need to do a deeper redesign and reimplementation.

Cohesion

- None of these terms are magical or sacred. Learn the ideas rather than the terminology. It's nearly always possible to write routines with functional cohesion, so focus your attention on functional cohesion for maximum benefit.

Good Routine Names
ПП с подходящи имена

- A good name for a routine clearly describes everything the routine does

ПП с подходящи имена

- **Describe everything the routine does.** In the routine's name, describe all the outputs and side effects. If a routine computes report totals and opens an output file, **`ComputeReportTotals()`** is not an adequate name for the routine. **`ComputeReportTotalsAndOpenOutputFile()`** is an adequate name but is too long and silly. If you have routines with side effects, you'll have many long, silly names. The cure is not to use less-descriptive routine names; the cure is to program so that you cause things to happen directly rather than with side effects.

ПП с подходящи имена

- **Avoid meaningless, vague, or wishy-washy verbs** Some verbs are elastic, stretched to cover just about any meaning. Routine names like `HandleCalculation()`, `PerformServices()`, `OutputUser()`, `ProcessInput()`, **and** `DealWithOutput()` don't tell you what the routines do. At the most, these names tell you that the routines have something to do with calculations, services, users, input, and output.

ПП с подходящи имена

- **Don't differentiate routine names solely by number** One developer wrote all his code in one big function. Then he took every 15 lines and created functions named **`Part1, Part2`**, and so on. After that, he created one high-level function that called each part. This method of creating and naming routines is especially egregious (and rare, I hope). But programmers sometimes use numbers to differentiate routines with names like **`OutputUser, OutputUser1, and OutputUser2`**. The numerals at the ends of these names provide no indication of the different abstractions the routines represent, and the routines are thus poorly named.

ПП с подходящи имена

- **`Make names of routines as long as necessary`** Research shows that the optimum average length for a variable name is 9 to 15 characters. Routines tend to be more complicated than variables, and good names for them tend to be longer. On the other hand, routine names are often attached to object names, which essentially provides part of the name for free. Overall, the emphasis when creating a routine name should be to make the name as clear as possible, which means you should make its name as long or short as needed to make it understandable.

ПП с подходящи имена

- **To name a function, use a description of the return value** A function returns a value, and the function should be named for the value it returns. For example, **`cos()`**, **`customerId.Next()`**, **`printer.IsReady()`**, **and** **`pen.CurrentColor()`** are all good function names that indicate precisely what the functions return

ПП с подходящи имена

- **`To name a procedure, use a strong verb followed by an object`** A procedure with functional cohesion usually performs an operation on an object. The name should reflect what the procedure does, and an operation on an object implies a verb-plus-object name. **`PrintDocument()`**, **`CalcMonthlyRevenues()`**, **`CheckOrderlnfo()`**, and **`RepaginateDocument()`** are samples of good procedure names.
- In object-oriented languages, you don't need to include the name of the object in the procedure name because the object itself is included in the call. You invoke routines with statements like **`document.Print(), orderInfo.Check(), and monthlyRevenues.Calc().`**
- Names like **`document.PrintDocument()`** are redundant and can become inaccurate when they're carried through to derived classes.

ПП с подходящи имена

- **Use opposites precisely** Using naming conventions for opposites helps consistency, which helps readability. Opposite-pairs like first/last are commonly understood. Opposite-pairs like **`FileOpen() and _lclose()`** are not symmetrical and are confusing. Here are some common opposites:

add/remove	increment/decrement	open/close
Begin/end	insert/delete	show/hide
create/destroy	lock/unlock	source/target
first/last	min/max	start/stop
get/put	next/previous	up/down
get/set	old/new	

ПП с подходящ размер

- How Long Can a Routine Be?

ПП с подходящ размер

- The theoretical best maximum length is often described as one screen or one or two pages of program listing, approximately 50 to 150 lines. In this spirit, IBM once limited routines to 50 lines, and TRW limited them to two pages (McCabe 1976).
- Modern programs tend to have volumes of extremely short routines mixed in with a few longer routines. Long routines are far from extinct, however.
- Shortly before finishing this book, I visited two client sites within a month. Programmers at one site were wrestling with a routine that was about 4,000 lines of code long, and programmers at the other site were trying to tame a routine that was more than 12,000 lines long!

ПП с подходящ размер

- A study by Basili and Perricone found that routine size was inversely correlated with errors: as the size of routines increased (up to 200 lines of code), the number of errors per line of code decreased (Basili and Perricone 1984).
- Another study found that routine size was not correlated with errors, even though structural complexity and amount of data were correlated with errors (Shen et al. 1985).
- A 1986 study found that small routines (32 lines of code or fewer) were not correlated with lower cost or fault rate (Card, Church, and Agresti 1986; Card and Glass 1990). The evidence suggested that larger routines (65 lines of code or more) were cheaper to develop per line of code.
- An empirical study of 450 routines found that small routines (those with fewer than 143 source statements, including comments) had 23 percent more errors per line of code than larger routines but were 2.4 times less expensive to fix than larger routines (Selby and Basili 1991).
- Another study found that code needed to be changed least when routines averaged 100 to 150 lines of code (Lind and Vairavan 1989).
- A study at IBM found that the most error-prone routines were those that were larger than 500 lines of code. Beyond 500 lines, the error rate tended to be proportional to the size of the routine (Jones 1986a).

ПП с подходящ размер

- Where does all this leave the question of routine length in OOP?
- A large percentage of routines in object-oriented programs will be accessor routines, which will be very short.
- From time to time, a complex algorithm will lead to a longer routine, and in those circumstances, the routine should be allowed to grow organically up to 100–200 lines. (A line is a noncomment, nonblank line of source code.)
- That said, if you want to write routines longer than about 200 lines, be careful. None of the studies that reported decreased cost, decreased error rates, or both with larger routines distinguished among sizes larger than 200 lines, and you're bound to run into an upper limit of understandability as you pass 200 lines of code.

ПП и подбор на параметри

- How to Use Routine Parameters

ПП и подбор на параметри

Put parameters in input-modify-output order Instead of ordering parameters randomly or alphabetically, list the parameters that are input-only first, input-and-output second, and output-only third. This ordering implies the IPO sequence of operations happening within the routine-inputting data, changing it, and sending back a result. Here are examples of parameter lists in Ada:

ПП и подбор на параметри

- **Ada Example of Parameters in Input-Modify-Output Order**

```
procedure InvertMatrix(
    originalMatrix: in Matrix;
     resultMatrix: out Matrix );
...
procedure ChangeSentenceCase(
     desiredCase: in StringCase;
    sentence: in out Sentence );
...
procedure PrintPageNumber(
    pageNumber: in Integer;
    status: out StatusType );
```

- (1)Ada uses in and out keywords to make input and output parameters clear.

ПП и подбор на параметри

- This ordering convention conflicts with the C-library convention of putting the modified parameter first. The input-modify-output convention makes more sense to me, but if you consistently order parameters in some way, you will still do the readers of your code a service.

ПП и подбор на параметри

- **Consider creating your own** in and out keywords Other modern languages don't support the **`in`** and **`out`** keywords like Ada does. In those languages, you might still be able to use the preprocessor to create your own **`in`** and **`out`** keywords:

ПП и подбор на параметри

- **C++ Example of Defining Your Own In and Out Keywords**

```
    #define IN
    #define OUT
    void InvertMatrix(
         IN Matrix originalMatrix,
         OUT Matrix *resultMatrix );
  ...
    void ChangeSentenceCase(
         IN StringCase desiredCase,
         IN OUT Sentence *sentenceToEdit );
  ...
    void PrintPageNumber(
         IN int pageNumber,
         OUT StatusType &status );
```

In this case, the IN and OUT macro-keywords are used for documentation purposes. To make the value of a parameter changeable by the called routine, the parameter still needs to be passed as a pointer or as a reference parameter.

ПП и подбор на параметри

- Before adopting this technique, be sure to consider a pair of significant drawbacks:
- Defining your own IN and OUT keywords extends the C++ language in a way that will be unfamiliar to most people reading your code. If you extend the language this way, be sure to do it consistently, preferably projectwide.
- A second limitation is that the IN and OUT keywords won't be enforceable by the compiler, which means that you could potentially label a parameter as IN and then modify it inside the routine anyway. That could lull a reader of your code into assuming that code is correct when it isn't. Using C++'s const keyword will normally be the preferable means of identifying input-only parameters.

ПП и подбор на параметри

- **If several routines use similar parameters, put the similar parameters in a consistent order**
- The order of routine parameters can be a mnemonic, and inconsistent order can make parameters hard to remember. For example, in C, the `fprintf()` routine is the same as the `printf()` routine except that it adds a file as the first argument. A similar routine, `fputs()`, is the same as `fputs()` except that it adds a file as the last argument. This is an aggravating, pointless difference that makes the parameters of these routines harder to remember than they need to be.
- On the other hand, the routine `strncpy()` in C takes the arguments target string, source string, and maximum number of bytes, in that order, and the routine `memcpy()` takes the same arguments in the same order. The similarity between the two routines helps in remembering the parameters in either routine

ПП и подбор на параметри

- **Use all the parameters**
- If you pass a parameter to a routine, use it. If you aren't using it, remove the parameter from the routine interface. Unused parameters are correlated with an increased error rate. In one study, 46 percent of routines with no unused variables had no errors, and only 17 to 29 percent of routines with more than one unreferenced variable had no errors (Card, Church, and Agresti 1986).
- This rule to remove unused parameters has one exception. If you're compiling part of your program conditionally, you might compile out parts of a routine that use a certain parameter. Be nervous about this practice, but if you're convinced it works, that's OK too. In general, if you have a good reason not to use a parameter, go ahead and leave it in place. If you don't have a good reason, make the effort to clean up the code.

ПП и подбор на параметри

- **Put status or error variables last**
- By convention, status variables and variables that indicate an error has occurred go last in the parameter list. They are incidental to the main purpose of the routine, and they are output-only parameters, so it's a sensible convention.

ПП и подбор на параметри

- **Don't use routine parameters as working variables**
- It's dangerous to use the parameters passed to a routine as working variables. Use local variables instead. For example, in the following Java fragment, the variable **`inputVal`** is improperly used to store intermediate results of a computation

Java Example of Improper Use of Input Parameters

```
int Sample( int inputVal ) {
   inputVal = inputVal * CurrentMultiplier( inputVal );
   inputVal = inputVal + CurrentAdder( inputVal );
   ...
   return inputVal;        <-- 1
}
```

- (1)At this point, **`inputVal`** no longer contains the value that was input.
- In this code fragment, inputVal is misleading because by the time execution reaches the last line, inputVal no longer contains the input value; it contains a computed value based in part on the input value, and it is therefore misnamed. If you later need to modify the routine to use the original input value in some other place, you'll probably use inputVal and assume that it contains the original input value when it actually doesn't

Java Example of Good Use of Input Parameters

A better approach is to avoid current and future problems by using working variables explicitly. :

```
int Sample( int inputVal ) {

    int workingVal = inputVal;

    workingVal = workingVal * CurrentMultiplier( workingVal );

    workingVal = workingVal + CurrentAdder( workingVal );

    ...

        <-- 1

    ...

    return workingVal;

 }
```

- (1)If you need to use the original value of inputVal here or somewhere else, it's still available.
  - Introducing the new variable workingVal clarifies the role of inputVal and eliminates the chance of erroneously using inputVal at the wrong time.
  - Assigning the input value to a working variable emphasizes where the value comes from. It eliminates the possibility that a variable from the parameter list will be modified accidentally. In C++, this practice can be enforced by the compiler using the keyword const. If you designate a parameter as const, you're not allowed to modify its value within a routine.

ПП и подбор на параметри

Document interface assumptions about parameters If you assume the data being passed to your routine has certain characteristics, document the assumptions as you make them. It's not a waste of effort to document your assumptions both in the routine itself and in the place where the routine is called. Don't wait until you've written the routine to go back and write the comments—you won't remember all your assumptions. Even better than commenting your assumptions, use assertions to put them into code.

What kinds of interface assumptions about parameters should you document?

- Whether parameters are input-only, modified, or output-only
- Units of numeric parameters (inches, feet, meters, and so on)
- Meanings of status codes and error values if enumerated types aren't used
- Ranges of expected values
- Specific values that should never appear

ПП и подбор на параметри

Limit the number of a routine's parameters to about seven Seven is a magic number for people's comprehension. Psychological research has found that people generally cannot keep track of more than about seven chunks of information at once (Miller 1956). This discovery has been applied to an enormous number of disciplines, and it seems safe to conjecture that most people can't keep track of more than about seven routine parameters at once.

In practice, how much you can limit the number of parameters depends on how your language handles complex data types. If you program in a modern language that supports structured data, you can pass a composite data type containing 13 fields and think of it as one mental "chunk" of data. If you program in a more primitive language, you might need to pass all 13 fields individually.

If you find yourself consistently passing more than a few arguments, the coupling among your routines is too tight. Design the routine or group of routines to reduce the coupling. If you are passing the same data to many different routines, group the routines into a class and treat the frequently used data as class data.

ПП и подбор на параметри

Consider an input, modify, and output naming convention for parameters If you find that it's important to distinguish among input, modify, and output parameters, establish a naming convention that identifies them. You could prefix them with **`i_`**, **`m_`**, and **`o_`**. If you're feeling verbose, you could prefix them with **`Input_`**, **`Modify_`**, and **`Output_`**.

ПП и подбор на параметри

Make sure actual parameters match formal parameters Formal parameters, also known as "dummy parameters," are the variables declared in a routine definition. Actual parameters are the variables, constants, or expressions used in the actual routine calls.

A common mistake is to put the wrong type of variable in a routine call —for example, using an integer when a floating point is needed. (This is a problem only in weakly typed languages like C when you're not using full compiler warnings. Strongly typed languages such as C++ and Java don't have this problem.) When arguments are input only, this is seldom a problem; usually the compiler converts the actual type to the formal type before passing it to the routine. If it is a problem, usually your compiler gives you a warning. But in some cases, particularly when the argument is used for both input and output, you can get stung by passing the wrong type of argument.

Develop the habit of checking types of arguments in parameter lists and heeding compiler warnings about mismatched parameter types.

Special Considerations in the use of Functions

Modern languages such as C++, Java, and Visual Basic support both functions and procedures. A function is a routine that returns a value; a procedure is a routine that does not. In C++, all routines are typically called "functions"; however, a function with a void return type is semantically a procedure. The distinction between functions and procedures is as much a semantic distinction as a syntactic one, and semantics should be your guide.

When to Use a Function and When to Use a Procedure

Purists argue that a function should return only one value, just as a mathematical function does. This means that a function would take only input parameters and return its only value through the function itself. The function would always be named for the value it returned, as **`sin()`**, **`CustomerID()`**, and **`ScreenHeight()`** are. A procedure, on the other hand, could take input, modify, and output parameters—as many of each as it wanted to.

Special Considerations in the use of Functions

A common programming practice is to have a function that operates as a procedure and returns a status value. Logically, it works as a procedure, but because it returns a value, it's officially a function. For example, you might have a routine called **`FormatOutput()`** used with a report object in statements like this one:

```
if ( report.FormatOutput( formattedReport ) = Success )
   then ...
```

In this example, report.FormatOutput() operates as a procedure in that it has an output parameter, formattedReport, but it is technically a function because the routine itself returns a value. Is this a valid way to use a function? In defense of this approach, you could maintain that the function return value has nothing to do with the main purpose of the routine, formatting output, or with the routine name, **`report.FormatOutput()`**. In that sense it operates more as a procedure does even if it is technically a function. The use of the return value to indicate the success or failure of the procedure is not confusing if the technique is used consistently.

Special Considerations in the use of Functions

The alternative is to create a procedure that has a status variable as an explicit parameter, which promotes code like this fragment:

```
report.FormatOutput( formattedReport, outputStatus )

if ( outputStatus = Success ) then ...
```

I prefer the second style of coding, not because I'm hard-nosed about the difference between functions and procedures but because it makes a clear separation between the routine call and the test of the status value. To combine the call and the test into one line of code increases the density of the statement and, correspondingly, its complexity. The following use of a function is fine too:

```
outputStatus = report.FormatOutput( formattedReport )

if ( outputStatus = Success ) then ...
```

In short, use a function if the primary purpose of the routine is to return the value indicated by the function name. Otherwise, use a procedure.

Special Considerations in the use of Functions

Setting the Function's Return Value

Using a function creates the risk that the function will return an incorrect return value. This usually happens when the function has several possible paths and one of the paths doesn't set a return value. To reduce this risk, do the following:

Check all possible return paths When creating a function, mentally execute each path to be sure that the function returns a value under all possible circumstances. It's good practice to initialize the return value at the beginning of the function to a default value—this provides a safety net in the event that the correct return value is not set.

Don't return references or pointers to local data As soon as the routine ends and the local data goes out of scope, the reference or pointer to the local data will be invalid. If an object needs to return information about its internal data, it should save the information as class member data. It should then provide accessor functions that return the values of the member data items rather than references or pointers to local data.

Макроси и вградени функции

- Macro Routines and Inline Routines

Макроси и вградени функции

- **Fully parenthesize macro expressions** Because macros and their arguments are expanded into code, be careful that they expand the way you want them to. One common problem lies in creating a macro like this one:
- **C++ Example of a Macro That Doesn't Expand Properly**

```
#define Cube( a ) a*a*a
```

If you pass this macro nonatomic values for a, it won't do the multiplication properly. If you use the expression Cube(x+1), it expands to x+1 * x + 1 * x + 1, which, because of the precedence of the multiplication and addition operators, is not what you want. A better, but still not perfect, version of the macro looks like this:

- **C++ Example of a Macro That Still Doesn't Expand Properly**

```
#define Cube( a ) (a)*(a)*(a)
```

This is close, but still no cigar. If you use Cube() in an expression that has operators with higher precedence than multiplication, the (a)*(a)*(a) will be torn apart. To prevent that, enclose the whole expression in parentheses:

- **C++ Example of a Macro That Works**

```
#define Cube( a ) ((a)*(a)*(a))
```

Макроси и вградени функции

- Surround multiple-statement macros with curly braces A macro can have multiple statements, which is a problem if you treat it as if it were a single statement. Here's an example of a macro that's headed for trouble:
-
- C++ Example of a Nonworking Macro with Multiple Statements

```
#define LookupEntry( key, index ) \

   index = (key - 10) / 5; \

   index = min( index, MAX_INDEX ); \

   index = max( index, MIN_INDEX );

...


for ( entryCount = 0; entryCount < numEntries; entryCount++ )

   LookupEntry( entryCount, tableIndex[ entryCount ] );
```

- This macro is headed for trouble because it doesn't work as a regular function would. As it's shown, the only part of the macro that's executed in the for loop is the first line of the macro:
- index = (key - 10) / 5;

Макроси и вградени функции

- To avoid this problem, surround the macro with curly braces:
- C++ Example of a Macro with Multiple Statements That Works

```
#define LookupEntry( key, index ) { \
   index = (key - 10) / 5; \
   index = min( index, MAX_INDEX ); \
   index = max( index, MIN_INDEX ); \
}
```

- The practice of using macros as substitutes for function calls is generally considered risky and hard to understand—bad programming practice—so use this technique only if your specific circumstances require it.

Макроси и вградени функции

- **Name macros that expand to code like routines so that they can be replaced by routines if necessary** The convention in C++ for naming macros is to use all capital letters. If the macro can be replaced by a routine, however, name it using the naming convention for routines instead. That way you can replace macros with routines and vice versa without changing anything but the routine involved.

Макроси и вградени функции

- **Limitations on the Use of Macro Routines**
- Modern languages like C++ provide numerous alternatives to the use of macros:
- **`const`** for declaring constant values
- **`inline`** for defining functions that will be compiled as inline code
- **`template`** for defining standard operations like min, max, and so on in a type-safe way
- **`enum`** for defining enumerated types
- **`typedef`** for defining simple type substitutions

Макроси и вградени функции

- **Inline Routines**
- C++ supports an `inline` keyword. An inline routine allows the programmer to treat the code as a routine at code-writing time, but the compiler will generally convert each instance of the routine into inline code at compile time. The theory is that inline can help produce highly efficient code that avoids routine-call overhead.
- Use inline routines sparingly Inline routines violate encapsulation because C++ requires the programmer to put the code for the implementation of the inline routine in the header file, which exposes it to every programmer who uses the header file.
- Inline routines require a routine's full code to be generated every time the routine is invoked, which for an inline routine of any size will increase code size. That can create problems of its own.
- The bottom line on inlining for performance reasons is the same as the bottom line on any other coding technique that's motivated by performance: profile the code and measure the improvement. If the anticipated performance gain doesn't justify the bother of profiling the code to verify the improvement, it doesn't justify the erosion in code quality either

Благодаря

За

Вниманието