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

Лекция 1а

Тема:

Проектиране и Работа с Подпрограми (част 1)

Съдържание:

- Концепцията ПП:
  - Главна и подчинена ПП (**call**, **return**)
  - ПП с равностойна подчиненост (**resume**)
- Обмен на данни между ПП:
  - Чрез съответствие аргументи<>параметри
  - Чрез общи данни
- Развитие на концепцията ПП (част 2)

Защо ПП?

Етапи в развитието на софтуерния процес:

- **Въвеждане на концепцията ПП**
- ПЕВН
- Програмни технологии
  - Структурно програмиране
  - ООП (компонентно, събитийно, обработка изключения)
- Конвенции за именуване на идентификатори
  - Мнемоника, нотации: унгарска, камила, Паскал
- Софтуерно инженерство (жизнен цикъл)
- Средства за автоматизация (CASE tools)
  - SW продукти за ОО анализ и проектиране

Причини за работа с ПП

- **Намалява сложността**
  - "Properly designed functions permit to ignore **how** a job's done. Knowing **what** is done is sufficient."

    B.Kernighan & D.Ritchie

  - "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.

Още Причини за работа с ПП

- Създава четлив код
- Опростява сложни логически изрази
  - Среща се дефиниция на функция, която се вика веднъж, само защото прави по-ясен сегмент от първичен код.
- Скрива опериране с указатели
- Подобрява преносимостта
- Подобрява х-ки на продукта по памет

Още Причини за работа с ПП

- Изолира, локализира сложни елементи
- Скрива подробности от реализацията
- Ограничава ефекти от промяна
- Скрива глобални данни
- Предоставя мултиупотреба (code reuse)
- Създава условия за ефективен рефакторинг

Още Причини за работа с ПП

- Reduce complexity
- Avoid duplicate code
- Make a section of code readable
- Simplify complicated Boolean tests
- Hide sequences
- Hide pointer operations
- Improve portability
- Improve performance
- Isolate complexity
- Hide implementation details
- Limit effects of change
- Hide global data
- Make central points of control
- Facilitate reusable code
- Accomplish a specific refactoring

ПП: Общи черти

- Всяка ПП има единствена входна точка.
- Извикващата ПП се суспендира (не е активна), докато извикваната ПП работи. Т.е. във всеки момент има само една работеща (активна) ПП.
- Управлението винаги се връща в извикващата ПП след като извикваната ПП приключи работа.

ПП: Основни понятия

- Дефиниция, определение (**subprogram definition)** на ПП
- Извикване на ПП (обръщение към ПП) (**subprogram call)**
- Кога една ПП е активна (**active)**
- Заглавие на ПП **(subprogram header)**
- Профил **(Parameter profile)** на ПП
- Протокол **(Protocol)** на ПП

ПП: Основни понятия

- Дефиниция, определение (**subprogram definition)** на ПП:
  - Описва интерфейса към ПП и действията, които ПП капсулира в себе си като отделна, самостоятелна, независима програмна единица.
- Извикване на ПП или обръщение към ПП (**subprogram call):**
  - Явна заявка, обява извиканата ПП да бъде изпълнена.
- ПП е активна (**active)**, докато се изпълнява.

ПП: Основни понятия

- Заглавие на ПП (**subprogram header)** има следните функции:
  - Указва начало на дефиницията на ПП
  - Съдържа името на ПП
  - Описва списък на формалните параметри, ако има такива
- Профил **(Parameter profile)** на ПП: брой, последователност (ред) и тип на формалните параметри.
- Протокол **(Protocol)** на ПП: Профилът на ПП плюс типът на връщаната стойност, ако ПП е функция.

Концепцията класическа ПП

// Прогр. текст	// модиф. Прогр. текст	// ПП текст	
A1	**A1 . . . .**	**ROUTINE B**	**// заглавие**
	CALL B	**. . . .**	
B	**A2 . . . .**	**. . . .**	**// тяло**
	CALL B	**. . . .**	
A2	**A3 . . . .**	**RETURN**	**// лог. край**
	CALL B	**END**	**// физ. край**
B	**A4 . . . .**		
A3			
B			
A4			

Поток на управление

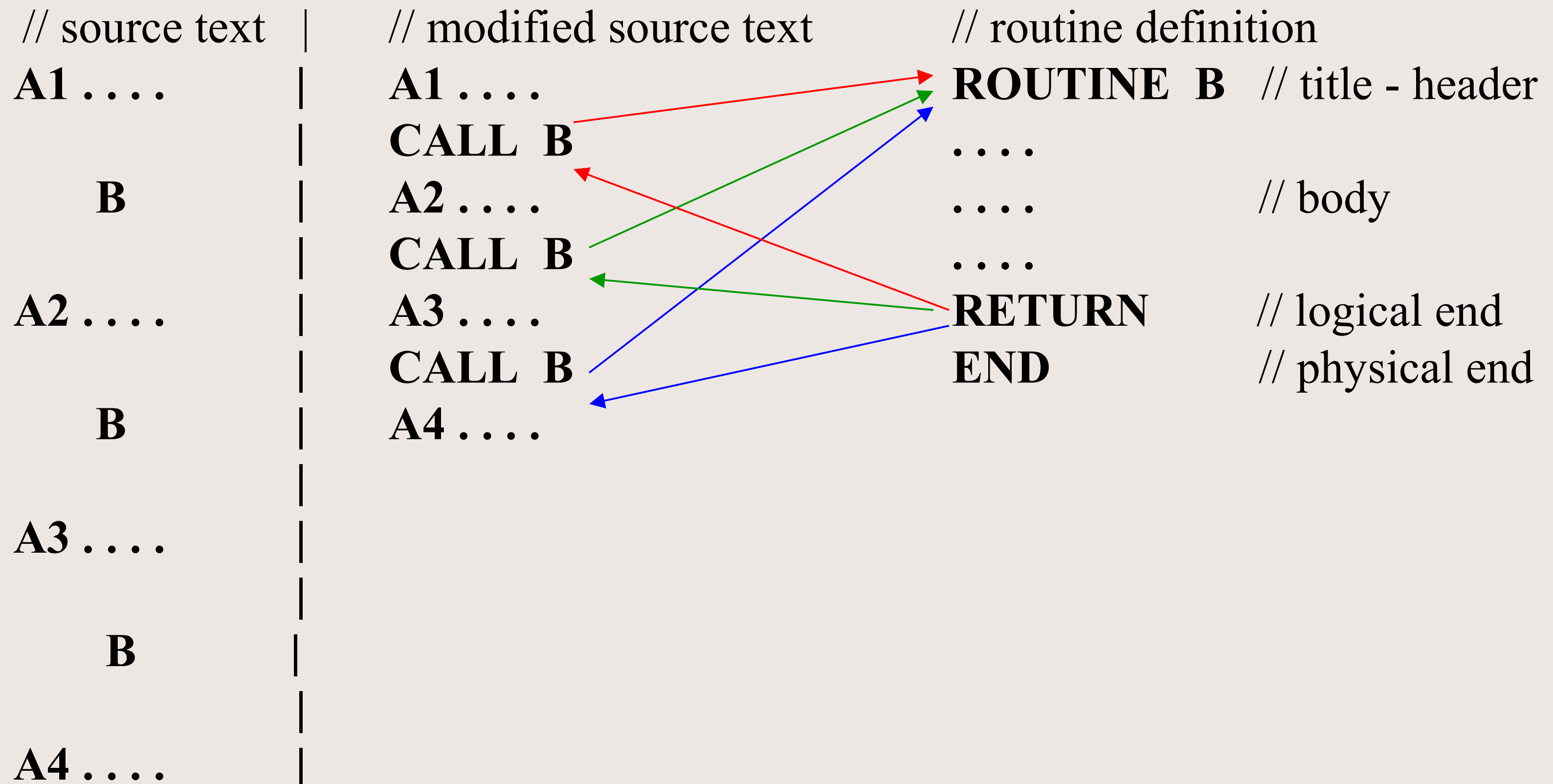
- Извикваща ПП (Master)

CALL B Управлението се предава в подчинената ПП B Slave (еднократно или многократно) през една и съща входна точка.

- Извиквана ПП (Slave)

RETURN Управлението се връща обратно в Master.

Поток на управление



Механизъм на действие

- Как работи оператор CALL?

CALL B: *STACK[SP] = IP + 1;*
SP++;
IP = StartAddressB

- Как работи оператор RETURN?

RETURN: *SP--;*
IP = STACK[SP];

CALL/RETURN операторите по същество са оператори за БП + допълнителни действия

За ползата от ПП

- Предимства:
  - Размерът на обектния код намалява
  - Възможност за модифицирани обработки при различни стойности на фактическите аргументи
- Недостатъци:
  - По-ниско бързодействие поради:

a/ съхр. Възвр. адрес b/зар. Форм. параметри

c/ модифицира стек d/възст. Възвр. адрес

Концепцията ПП

- Проблеми при проектиране и реализация на ПП в ПЕВН:
  - Избор на механизъм за предаване на параметри?
  - Прави ли се проверка по тип на формалните параметри и фактическите аргументи?
  - Как се заделя памет за локалните променливи? Статично или динамично?
  - Допуска ли се вграждане една в друга на ПП дефиниции?

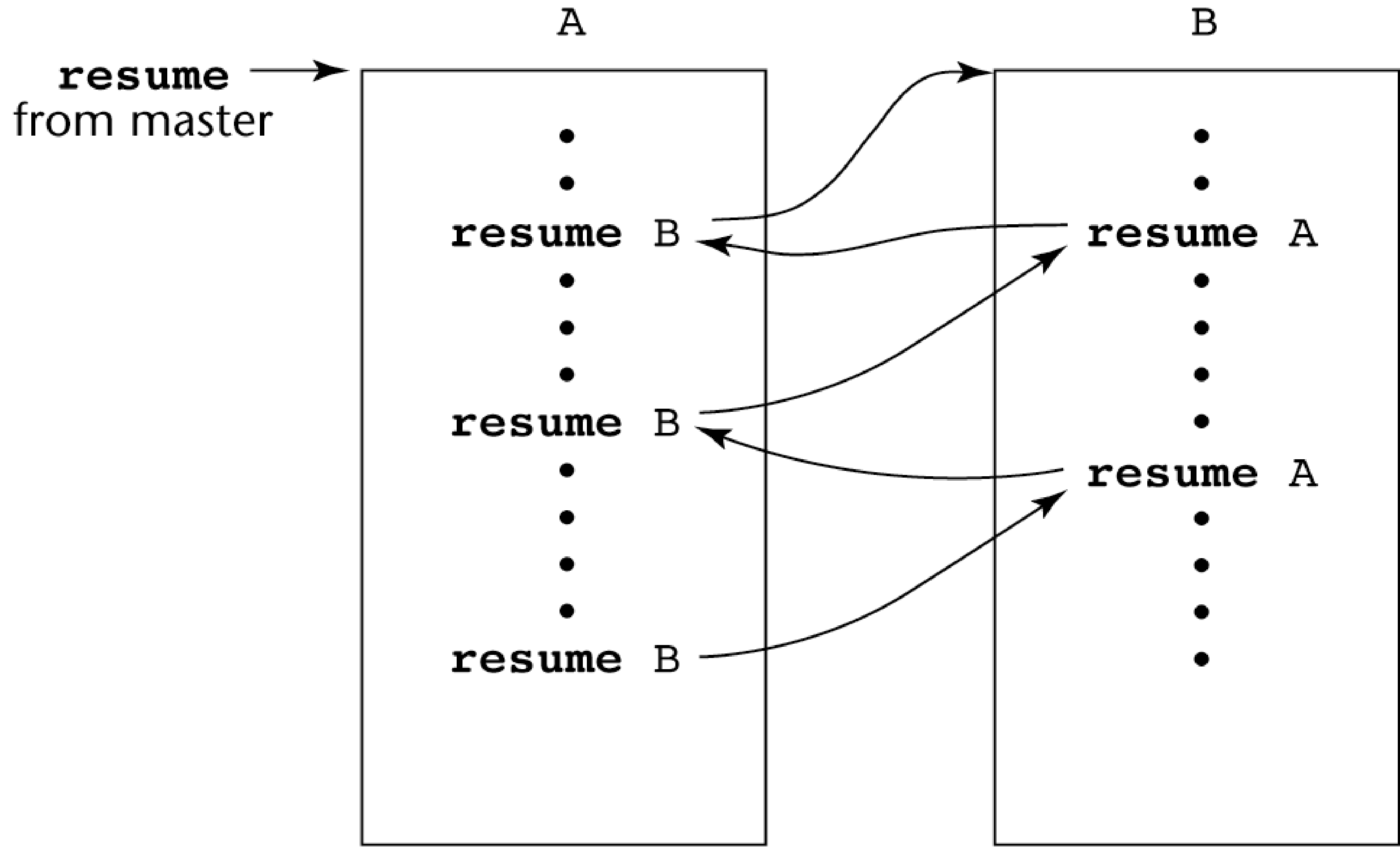
Концепцията ко-ПП

- Симетричен модел на управление без Master и без Slave, same level of majority
- Ко-ПП се управляват от главен модул (master unit).
- Главният модул предава контрола на ко-ПП.

ко-ПП A	ко-ПП B
RESUME B	***RESUME A***
RESUME B	***RESUME A***
RESUME B	***RESUME A***

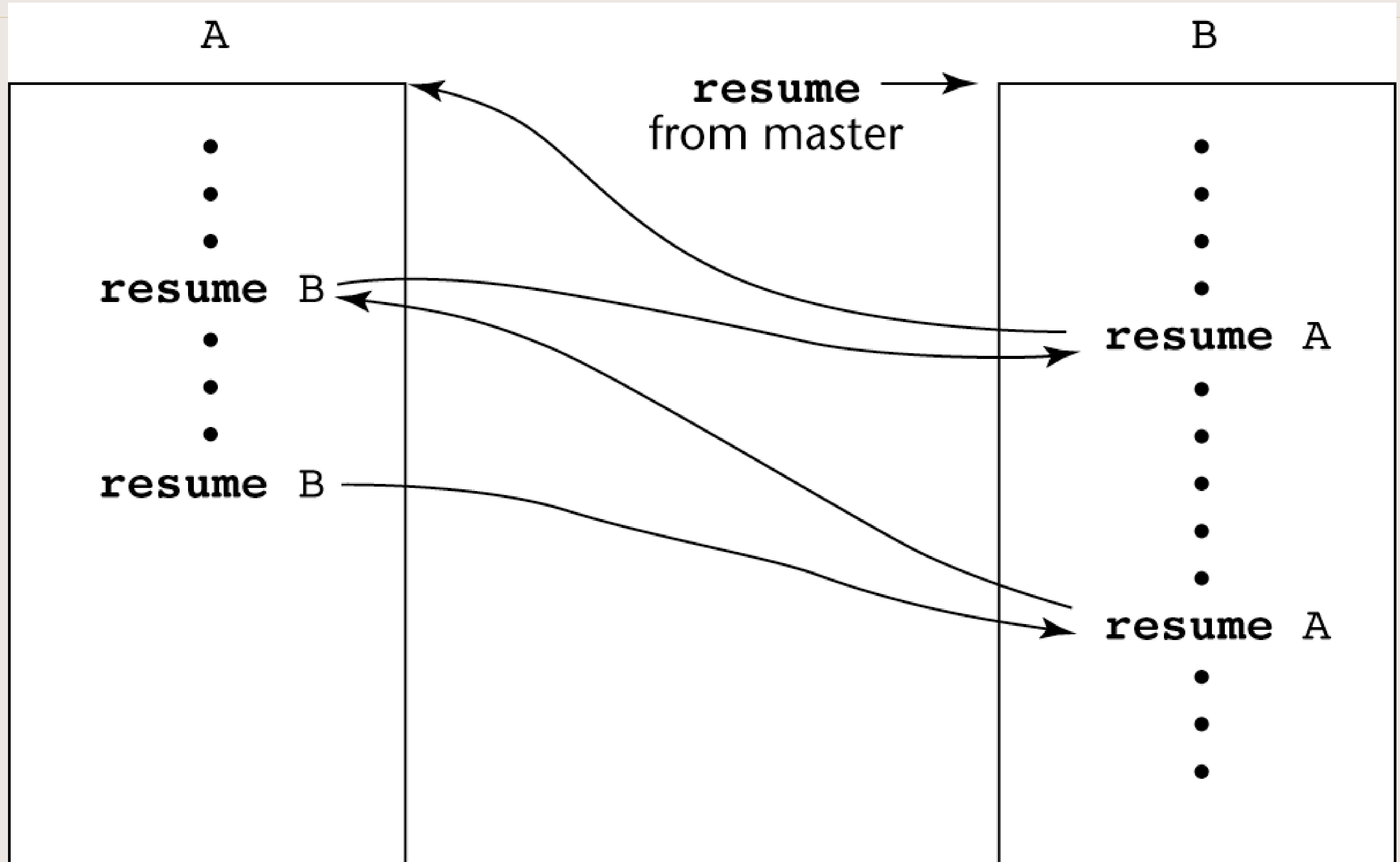
- Управлението се връща на главния модул.

Coroutines Illustrated: Possible Execution Controls



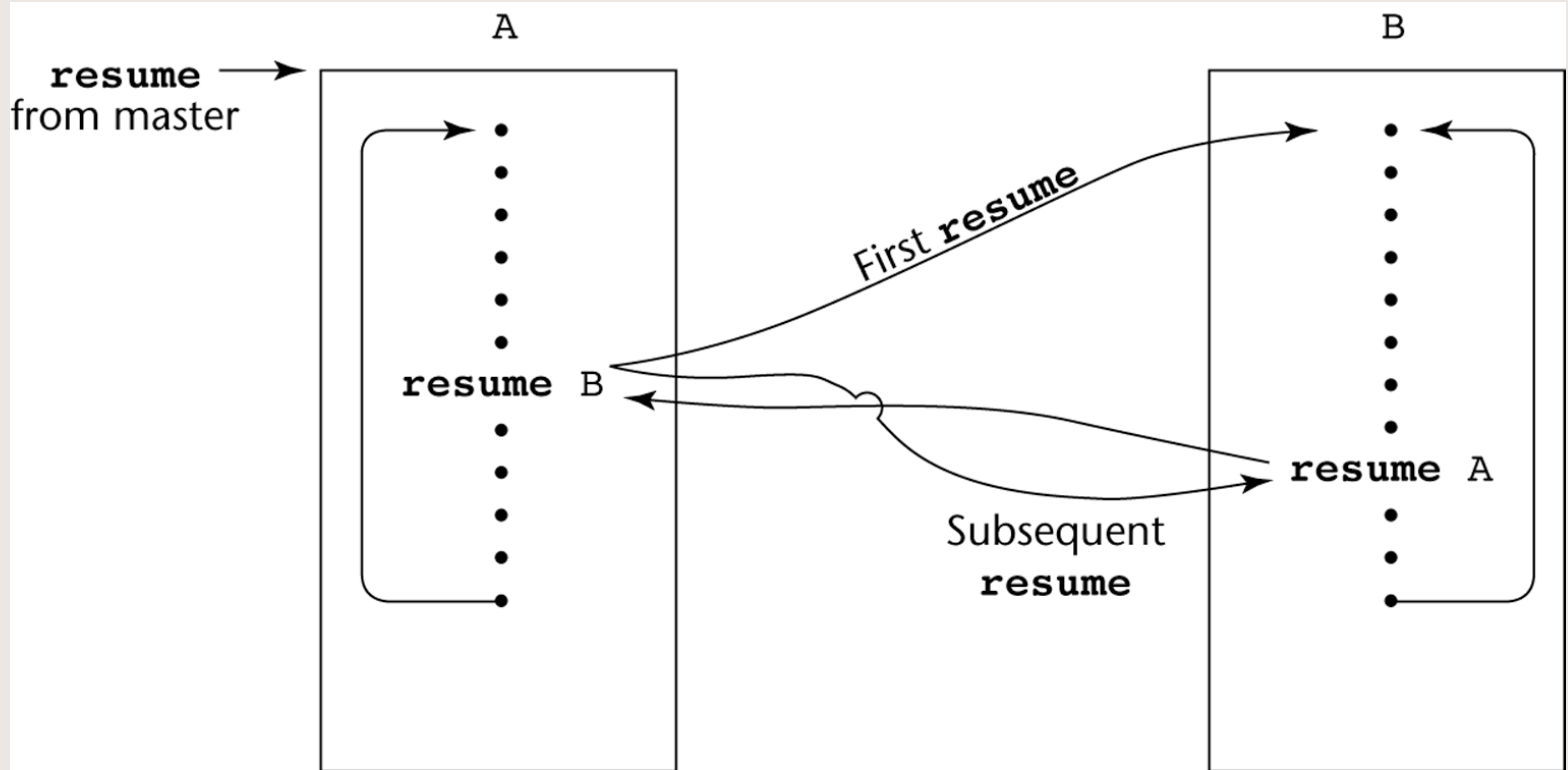
Copyright © 2009 Addison-Wesley. All rights reserved.

Coroutines Illustrated: Possible Execution Controls



Copyright © 2009 Addison-Wesley. All rights reserved.

Coroutines Illustrated: Possible Execution Controls with Loops



Copyright © 2009 Addison-Wesley. All rights reserved.

Вътрешен механизъм

BEG A:	*SP++*
	STACK[SP] = StartAddressB
RESUME B:	*POM = STACK[SP]*
	STACK[SP] = IP + 1
	IP = POM
RESUME A:	*POM = STACK[SP]*
	STACK[SP] = IP + 1
	IP = POM
END A:	*SP--*

Концепцията ко-ПП

- Примери
  - Компилатор <> Документатор (documenter)
  - Игра с 2 или повече участника

ПП-процедури и ПП-функции

- Две основни категории подпрограми:
  - Подпрограми процедури (Procedures)
  - Подпрограми функции (Functions)

Подпрограми Процедури

- *Procedures* са съвкупност от оператори (collection of statements) за параметризирани обработки.
- Процедурите се активират чрез *call* statement.
- Процедурите описват (създават) нови оператори **statements**. C/C++ няма оператор за сортиране на данни, но има ПП за сортинг.
- Процедури предоставят резултат по 2 начина:
  – Чрез общи променливи, достъпни в извикваната и извикващата ПП
  – Чрез формални параметри, които (ако) допускат пренос на данни от извикваната в извикващата ПП

Подпрограми Функции

- *Functions* структурно наподобяват процедури като съвкупност от оператори, но носят семантиката на математически функции.
- Функции се активират чрез името, следвано от фактически аргументи, което се среща като операнд в израз(и).
- Функциите описват нови операции **- user defined operators – sqrt(), pow(), ….**
- Функциите предоставят резултат по 3 начина:
  - Единична стойност – операнд на оператор return.
  - Други два начина като процедури (преден слайд)

ПП-процедури и ПП-функции

- ПЕВН с двата вида (Fortran, Pascal, VBasic).
- Си-подобните ПЕ имат само функции (и/или методи), които са с поведение на:
  - ПП функции, ако връщат определена стойност.
  - ПП процедури, ако типът връщана стойност е *void*. Т.к изразите се допуска да се употребяват в контекста на оператори, синтаксисът позволява stand-alone извикване на *void* функция.

    sort(scores, 100);
- Методите в Java, C++, C# по синтаксис наподобяват функциите в C.

Подпрограми в С/С++

Функция (метод):

декларация, прототип, сигнатура (prototype)

```
void func();
```

извикване, обръщение (call)

```
func();
```

определение, дефиниция (definition)

```
void func()
{
   . . .
}
```

Подпрограми в Pascal

Функция

Function func(. . .): integer;
(definitions *)*
BEGIN
(statements *)*
END

Ф.извикване:

res = func1;
res = func1(. . .);

Процедура

Procedure proc1(. . .);
(definitions *)*
BEGIN
(statements *)*
END

П.извикване:

proc1;
proc1(. . .);

Подпрограми в VBasic

Function

Function func(. . .) As type
(definitions *)*
(statements *)*
Return returnvalue **ИЛИ**
func = return vaue
End Function

Function call:

res = func1
res = func1(. . .)

Sub

Sub proc1(. . .)
(definitions *)*
. . .
(statements *)*
End Sub

Sub call:

[Call] proc1
[Call] proc1(. . .)

Подпрограми в Java,C#

Метод: Прототип не е необходим, т.к. Няма изискване за обявяване на методи преди да се използват оператори за обръщение към тях

```
public class Object {
    public void func1() { . . . }
    public static void func2()    { . . . }
}


Object a = new Object();
a.func1();                  // Instance method called
Object.func2();             // Class method called
```

Локални данни в среда на ПП

- В ПП се дефинират собствени променливи, **local** променливи. Те имат обсег (scope) и видимост вътре в ПП, където са дефинирани.
- Локални променливи са *static* и *stack-dynamic*.
- *Stack-dynamic* променливи се заделя памет в стека при активиране на ПП и паметта се освобождава при излизане от ПП. Те не запазват стойност между обръщенията.
- *Static* променливи заделят памет така, че те запазват стойност между отделни обръщения.

Локални данни в среда на ПП

- C/C++: Локалните променливи по подразбиране (by default) са стек-динамични. Статични променливи със запазена дума *static*.
- ПП в Ada и методите в Java, C++, C# работят със stack-dynamic променливи.

Обмен на данни м/у ПП

Два начина за достъп до данни, които една ПП следва да обработва:

- Чрез механизъма за съответствие между формални параметри и фактически аргументи
- Чрез пряк достъп до общи, глобални или не-локални променливи (обявени в друга част от проекта, но видими в ПП)

Параметри

- *Formal parameters* се срещат в ПП дефиниция.
- *Actual arguments* се срещат при извикване на ПП.
- Съответствие и съвместимост между формални параметри и фактически аргументи
- Формалните параметри се различават от фактическите аргументи поради наличие на различни ограничения върху тяхната форма, представяне и употреба.

Параметри

- Съответствие между формални параметри и фактически аргументи – 2 начина:
  - **Positional parameters**. Просто позиционно съответствие
  - **Keyword parameters**. Името на формалния параметър се цитира заедно с фактическия аргумент при извикването на ПП
  - Смесване на двата подхода

Пример за keyword parameters

- Ada ПП:
  - Заглавие с формални параметри *Nums*, *Res*

    procedure Sum(Nums:in Integer; Res:out Integer)

  - Call с фактически аргументи *n*, *FinalVal1*

    Sum(Nums => n, Res =>FinalVal1);
  - Call с фактически аргументи *m*, *FinalVal2*

    Sum(Res => FinalVal2, Nums => m);

Методи за предаване на параметри

Обмен на данни между ПП чрез съответствие между формални параметри и фактически аргументи:

- **Концептуални (Conceptual) модели**
- **Семантични (Semantic) модели**:
  - in-mode, out-mode, in/out-mode
- **Модели на реализация (Implementation)**:
  - Pass-by value, Pass-by-result,
  - Pass-by-value-result, Pass-by-reference

Методи за предаване на параметри

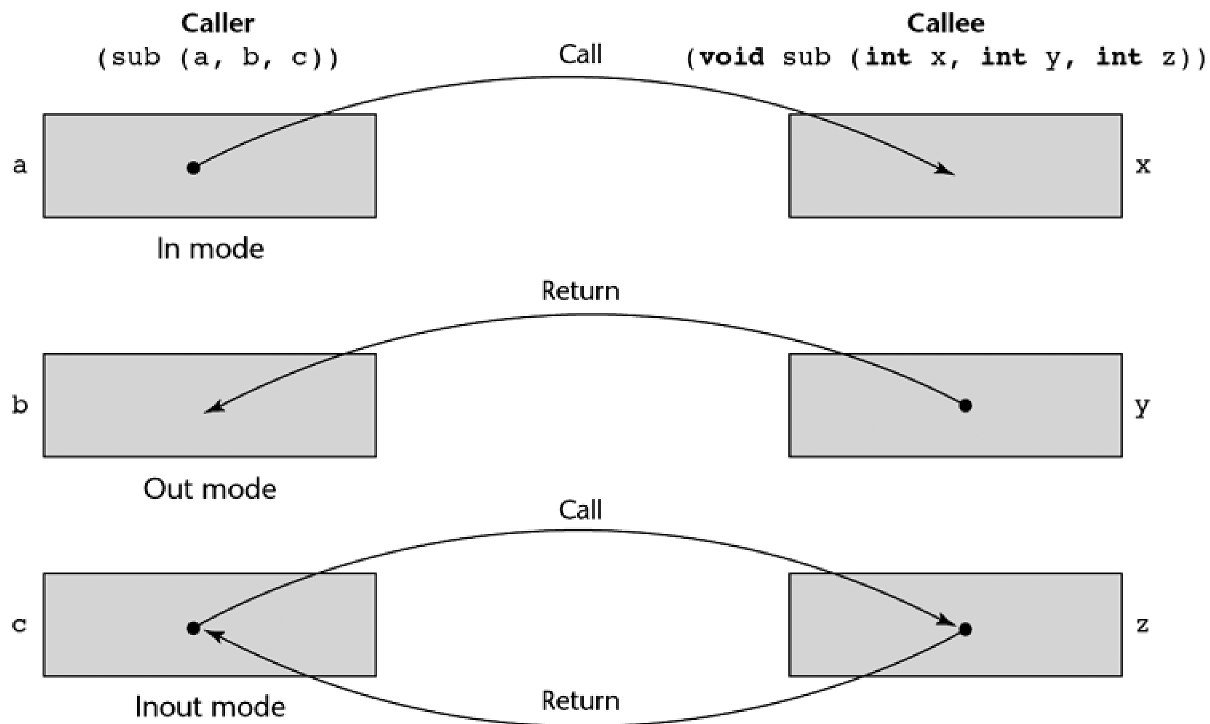
- **Концептуални модели**
- **Conceptual models**
  - Описват как се обменят данни при предаване на параметри:
    - Актуални данни се копират в едната, в другата или в двете посоки
    - Предава се пътека за достъп an access path (by pointer или by reference)

Семантични модели

- Известни са три семантични модела за предаване на параметри:
  - *In* mode
  - *Out* mode
  - *In out* mode

Фигурата на следния слайд показва разликата в трите модела при избран концептуален модел за копиране на данни.

Семантични модели



Модели на реализация при предаване на параметри

- Видове implementation models:
  - По стойност Pass-by-value
  - По резултат Pass-by-result
  - По стойност-резултат Pass-by-value-result
  - По синоним или чрез позоваване Pass-by-reference

Pass by Value

- *Pass by value* is an implementation model for **in mode** parameters. When a parameter is passed by value, the value of the actual argument is used to initialize the corresponding formal parameter, which then acts as a local variable in the routine
- *Pass by value* нормално се постига с копиране /copy/, т.е. пренос на данни.

Pass by Result

- *Pass by result* is an implementation model for **out** mode parameters. When a parameter is passed by result, no value is transmitted to the routine. The corresponding formal parameter acts as a local variable but just before the control is transferred back to the caller, its value is transmitted back to the caller's actual argument, which obviously must be a variable (no literal or expression can receive data sent by called).
- *Pass by result* нормално се постига с копиране /copy/, т.е. пренос на данни.

Pass by Value-Result

- *Pass by value-result* is an implementation model for **in out** mode parameters in which actual values are copied. The value of the actual argument is used to initialize the corresponding formal parameter, which then acts as a local variable. In fact, pass by value-result parameters must have local storage associated with called routine. At sub program termination, the value of the formal parameter is transmitted back to the caller's actual argument.
- *Pass by value-result* нормално се постига с копиране /copy/, т.е. пренос на данни и по тази причина се нарича още ***pass by copy***

Pass by Reference

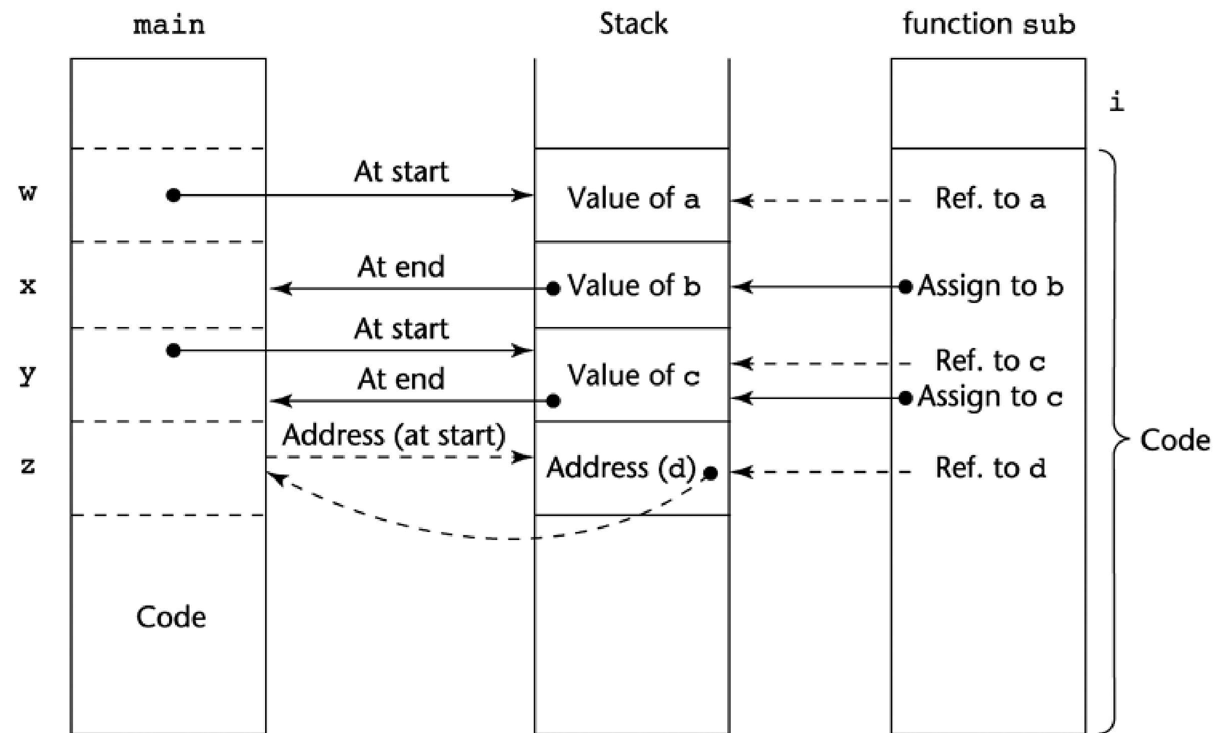
- *Pass by reference* is a second implementation model for **in out** mode parameters. Rather than copying data values back and forth, this method transmits an access path, sometimes just an address to called routine. Thus the slave is allowed to access the actual argument in the calling program unit. In effect, the actual argument is shared with the called routine
- Advantage: duplicate space is not required, nor is any copying to take place.

Предаване на параметри - реализация

- В повечето езици обменът на фактически и формални данни се осъществява посредством системния run-time stack.

- Фиг. на следния слайд илюстрира връзката между фактическите аргументи *w,x,y,z* и формалните параметри *a,b,c,d*
  - Pass by value (*w, a*)
  - Pass by result (*x, b*)
  - Pass by value result (*y, c*)
  - Pass by reference (*z, d*)

Предаване на параметри - реализация



Предаване на параметри

- Си прилага pass by-value. Pass-by-reference (in out mode) се постига чрез използване на адреси за фактически аргументи и указатели като формални параметри.
- C++ поддържа специфичен тип указател, наречен *reference* type.

```
void f(const int &p1, int p2, int &p3) { … }
```

- Всички Java параметри са pass by value

Предаване на параметри

- Ada поддържа трите семантични модела. За целта са въведени запазени думи *in, out, in out*.
- Пример заглавие на Ада ПП

```
procedure Adder(   A : in out Integer;
                   B : in Integer;
                   C : out Float)
```

Предаване на параметри

- VBasic:Parameter Passing by Value
- Кл. Дума *ByVal* означава “By Value”

- *ByVal* аргументите запазват своите original value след като ПП Sub или Function завърши

- Параметри се декларират в следния формат

[ByVal] Variable_Name As Data_Type

Предаване на параметри

- VBasic:Parameter Passing by Reference
- Кл. Дума *ByRef* означава “By Reference”

- *ByRef* аргументите могат да се променят стойности от ПП Sub или Function, като задържат новите стойности след като ПП Sub или Function завърши.

- Параметри се декларират в следния формат

  ByRef Variable_Name As Data_Type

Предаване на параметри

C# поддържа по подразбиране метод pass by value. Pass by reference може явно да се зададе чрез запазена дума *ref*, едновременно преди формалния параметър и преди фактическия аргумент

```
void adder(ref int oldSum, int newSum) {…}
 . . .
adder(ref sum, newValue);
```

Параметри by value C,C++

```
void swap(int, int);
//-------------------------------------
void main()
{int a=20, b=50;
     swap(a, b);
}
//-------------------------------------
void swap(int x, int y)
{ int temp;
  temp = x;     x = y;     y = temp;
}
```

Параметри by address C,C++

```
void swap(int *, int *);
//---------------------------------------
void main()
 { int a=20, b=50;
     swap(&a, &b);
}
//---------------------------------------
void swap(int *px, int *py)
{ int temp;
  temp=*px; *px=*py; *py=temp;
}
```

Параметри by reference C++

```
void swap(int&, int&);
//------------------------------------
void main()
{ int a=20, b=50;
  swap(a, b);
}
//------------------------------------
void swap(int& sx, int& sy)
{ int temp;
  temp = sx;   sx = sy;   sy = temp;
}
```

Без параметри, общи данни

```
int a=20, b=50;
void swap();
//-------------------------------------
void main()
{ swap( );
 }
//-------------------------------------
void swap()
{ int temp;
  temp = a;   a = b;   b = temp;
}
```

Проблемът swap в C#

```
static void Main(string[] args)
        {
            int a = 20, b = 50;
            swap(a, b);

        a = 33; b = 66;
         swap(ref a, ref b);
         } // end of Main

        static void swap(int pa, int pb)
        {
            int temp; temp = pa; pa = pb; pb = temp;
        }

        static void swap(ref int pa, ref int pb)
        {
            int temp; temp = pa; pa = pb; pb = temp;
        }
```

Проблемът swap в Java

```
class Int {
  private int x;

  public Int() { x=0; }
  public Int( int x ) { this.x = x; }

  public int getNum() { return this.x;}
  public void setNum(int x) { this.x = x;}

}  // end of class Int
```

Проблемът swap в Java

```
public class ProgSwapProblem {
     public static void main(String args[]) {
        int a = 20, b=50;
        swap(a, b);

        Int aa = new Int(), bb = new Int();
        aa.setNum(a);  bb.setNum(b);
        swap( aa, bb );
        a=aa.getNum();  b=bb.getNum();
       } // end of main

         static void swap(int pa, int pb) {
         int temp;
         temp=pa; pa = pb;  pb=temp;
         }

      static void swap(Int pa, Int pb) {
        int temp;
        temp = pa.getNum();
        pa.setNum(pb.getNum());
        pb.setNum(temp);
      }
   }  // end of main class
```

Параметри на командния ред

```
int main( )
{ return 0; }

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

Параметри на командния ред

```
int main( )
{ return 0; }

//----------------------------------------------------
int main(int argc)
{ return 0; }

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

Параметри на командния ред

```
int main( )
{ return 0; }

//------------------------------------------------
int main(int argc)
{ return 0; }

//------------------------------------------------
int main(int argc, char *argv[])
{ return 0; }

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

Параметри на командния ред

```
int main( )
{ return 0; }

//----------------------------------------------
int main(int argc)
{ return 0; }

//----------------------------------------------
int main(int argc, char *argv[])
{ return 0; }

//----------------------------------------------
int main(int argc, char *argv[], char **envp)
{ return 0; }
```

Параметри на командния ред

C++

```
int main(int argc,char *argv[]) {
  … }
```

C#

```
static void Main(string[] args)
  { … }
```

Java

```
static void main(String[] args)
  { … }
```

Развитие на ПП в C/C++

- Тук е краят на ПП лекция 1.
- Следните слайди до края са прехвърлени в ПП лекция 2

Развитие на ПП в C/C++

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

```
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;}
```

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 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.;
   }
```

Вградени функции

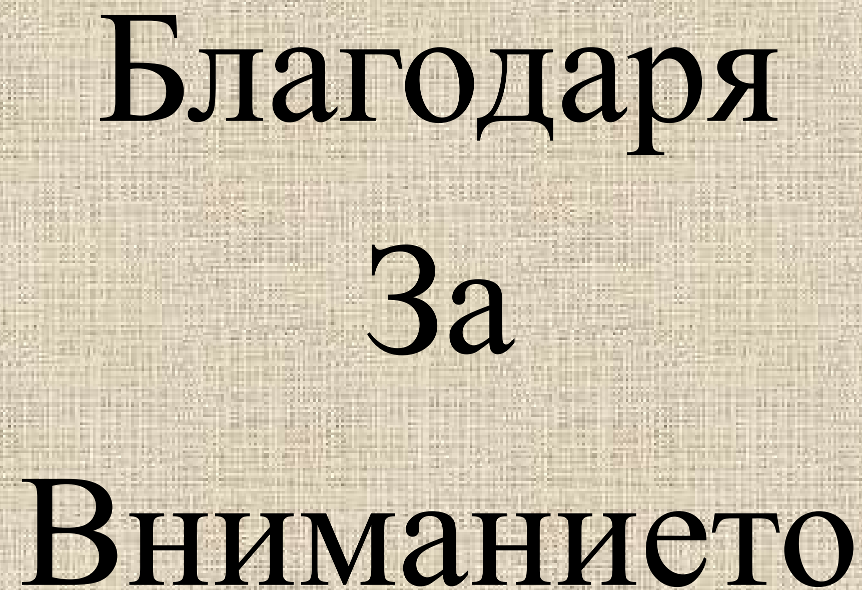
```
inline float area(float r) {return PI*r*r;}
```

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);
```


Благодаря
За
Вниманието