Процедуры и функции

Абстракция параметризованной совокупности действий.

- Функция – описание вычисления значения; вызов функции – выражение;
- Процедура – описание действий по изменению состояния памяти; вызов функции – оператор
- C: процедура – функция, «вырабатывающая значение» типа **void**.

Описание функции

- Описание типа (спецификации) функции:
  - тип результата
  - типы (и имена) аргументов – *формальных параметров*
- Именование функции
- Описание тела функции
- Область видимости

Процедуры и функции

Синтаксис:

описатель:

функция:

тип – *описатель*

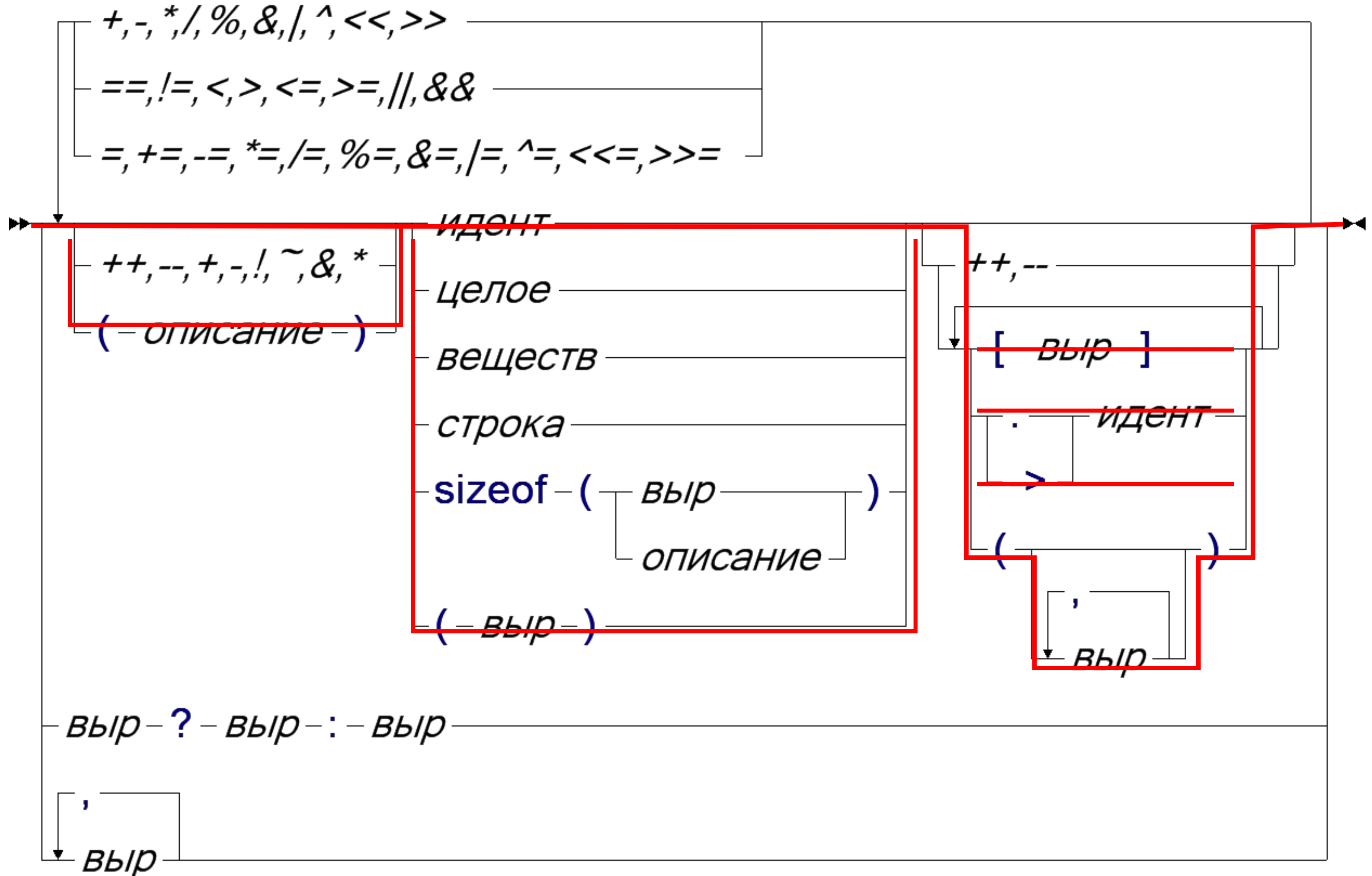
{ *описание* ;

оператор }

Вызов функции

- Выражение, значением которого является вызываемая функция
- *Фактические параметры -* выражения, значения которых подставляются вместо формальных параметров

Вызов функции



Вызов функции – пример (С)

- ToPolar(x, y, &alpha, &ro)
- * (shift ? sin : cos) (n * pi / 3)
- (* F[i])(x > 0 ? 1 : x=-x , -1)
- Ack(m-1, Ack(m,n-1))
- WriteLn; - типичная ошибка
- WriteLn(); - правильно

Вызов функции – шаги исполнения

1. Вычисляется вызываемая функция
2. Вычисляются фактические параметры
3. Создаются локальные объекты: формальные параметры, локальные объекты тела функции
4. Значения фактических параметров «связываются» с формальными параметрами
5. Выполняется тело функции
6. Удаляются локальные объекты
7. Возвращается результат

Время жизни локальных объектов

Оператор **return**

Синтаксис:

```
—return ─┬──────┬─ ; —
         └ expr ┘
```

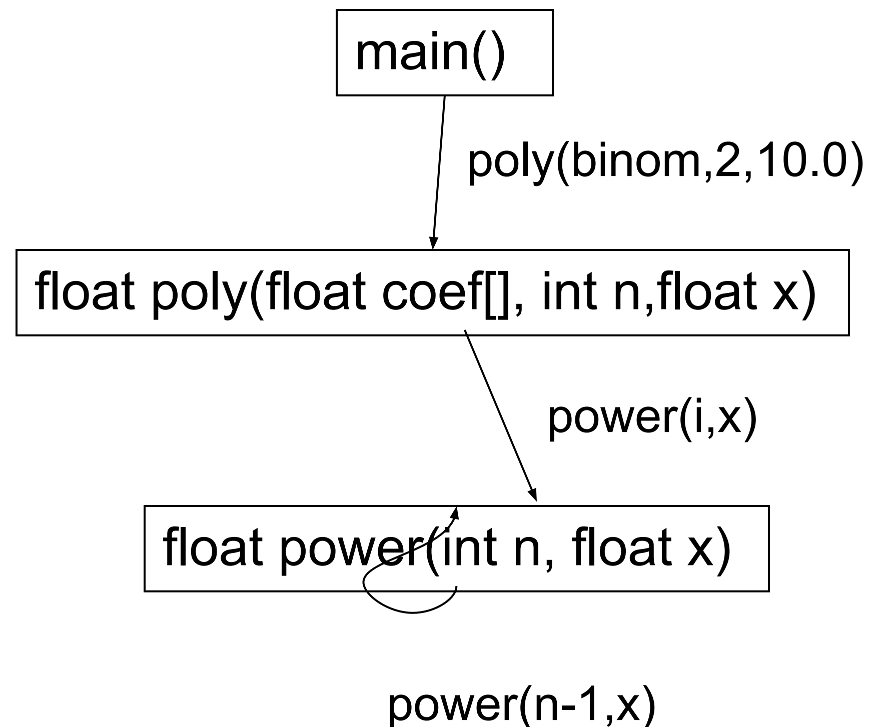
- Вычисление результата функции
- Завершение выполнения функции

Функции - пример

```
float poly(float coef[], int n,float x)
{
   float sum = 0f;
   for (int i=0; i<=n; i++)
      sum += coef[i] * power(i.x);
   return sum;
}
float power(int n, float x)
{
  return n==0 ? 1 : x*power(n-1,x);
}
void main()
{
  float binom[] = {1,2,1};
  printf("%d", poly(binom,2,10.0));
}
```

Граф вызовов:

- Вершины - функции
- Дуги -вызовы



Функции - пример

```
float poly(float coef[], int n,float x)
{
   float sum = 0f;
   for (int i=0; i<=n; i++)
      sum += coef[i] * power(i,x);
   return sum;
}
float power(int n, float x)
{
  return n==0 ? 1 : x*power(n-1,x);
}
void main()
{
  float binom[] = {1,2,1};
  printf("%d", poly(binom,2,10.0));
}
```

Стек:

main:

	{1,2,1}
binom	

poly:

coef	
n	2
x	10.0
i	3
sum	121.0

power:

n	2
x	10.0

power:

n	1
x	10.0

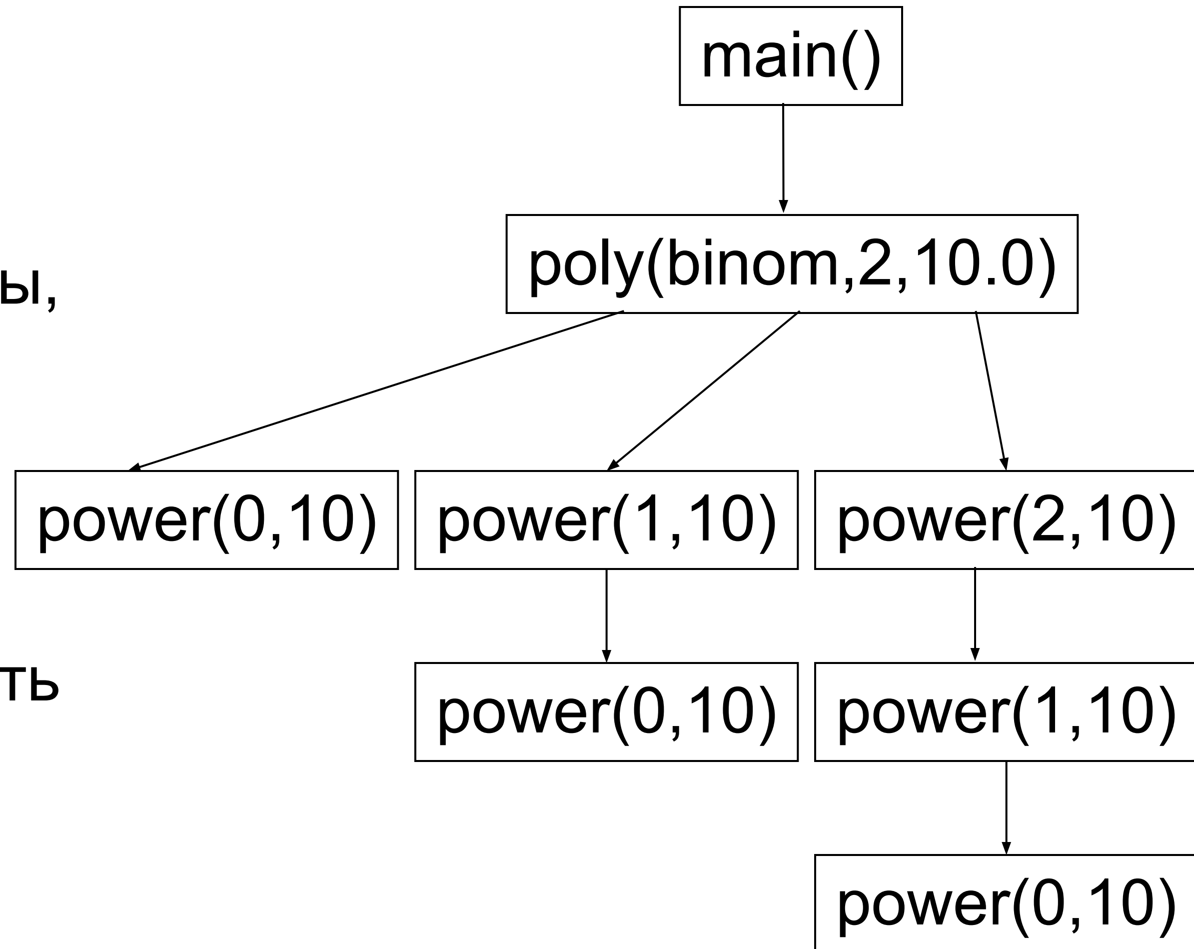
power:

n	0
x	10.0

Дерево вызовов

Дерево вызовов – упорядоченное дерево:

- Вершины – вызовы, с указанием значений фактических параметров
- Дуги - вложенность вызовов



Рекурсия

- Статическая рекурсия - цикл в графе вызовов (разрешимое свойство)
- Динамическая рекурсия – при некотором исполнении программы вершина и некоторый её потомок в дереве вызовов соответствуют одной и той же функции (неразрешимое свойство)

Рекурсия

Вопрос: сколько раз вызывается power?

- «Статический» ответ: 2
  - power(i,x)
  - power(n-1,x)
- «Динамический» ответ: 6
  - power(2,10) – 1 раз
  - power(1,10) – 2 раза
  - power(0,10) – 3 раза

  В общем случае n*(n-1)

Рекурсия – эффективность?

- Вызов функции – дорогостоящая операция (отведение памяти, пересылка параметров, запоминание точки возврата и т.д.)
- Для организации вызовов требуется дополнительная память, пропорциональная высоте дерева вызовов
- Нерекурсивные функции могут быть реализованы эффективнее, например, за счёт статического выделения памяти для локальных объектов
- Рекурсия затрудняет статический анализ программы

Рекурсия - достоинство

- Позволяет естественно реализовать по-существу рекурсивный алгоритм

```
struct Person
{
   struct Person * Parent;
   char Name[32];
   unsigned int ChildrenCount;
   struct Person * Children;
} * root;
```

```
struct Person* Find(
            struct Person * p,
            char * Name)
{
   if (p == NULL || strcmp(p->Name,Name) == 0)
      return p;
   struct Person * res = NULL;
   for (int i = 0; i<ChildrenCount; i++)
      if ((res = Find(p->Children[i],Name)) != NULL)
         break;
   return res;
}
```

Вложенные процедуры

```
float poly(float coef[], int n,float x)
{
   float sum = 0f;
   for (int i=0; i<=n; i++)
      sum += coef[i] * power(i.x);
   return sum;
}
float power(int n, float x)
{
  return n==0 ? 1 : x*power(n-1,x);
}
void main()
{
  float binom[] = {1,2,1};
  printf("%d", poly(binom,2,10.0));
}
```

```
float poly(float coef[], int n,float x)
{
  float power(int n)
  {
   return n==0 ? 1 : x*power(n-1);
  }
   float sum = 0f;
   for (int i=0; i<=n; i++)
      sum += coef[i] * power(i);
   return sum;
}
void main()
{
  float binom[] = {1,2,1};
  printf("%d", poly(binom,2,10.0));
}
```

Вложенные процедуры – динамический контекст

```
float poly(float coef[], int n,float x)
{
  float power(int n)
  {
    return n==0 ? 1 : x*power(n-1);
  }
   float sum = 0f;
   for (int i=0; i<=n; i++)
      sum += coef[i] * power(i);
   return sum;
}
void main()
{
  float binom[] = {1,2,1};
  printf("%d", poly(binom,2,10.0));
}
```

main:

	{1,2,1}
binom	

poly:

coef	
n	2
x	10.0
i	2
sum	21.0

power:

n	2

power:

n	1

x

Вложенные процедуры

- Реализация существенно сложнее, поскольку требуется поддерживать динамическую цепочку контекстов
- Может существенно уменьшить количество передаваемых параметров
- Pascal – есть, C – нет.

Переменное число параметров - printf (C)

```
#include <stdarg.h>

extern void print_char(unsigned c);
extern void print_int(int c);
extern void print_float(float c);

void my_printf(char * format, …)
{
  va_list     argptr;
  va_start(argptr, format);
  for (char * f = format; *f; f++)
    if (f[0]==‘%’ && f[1])
    {
      switch(f[1])
      {
        case ‘c’ :
          print_char((unsigned) va_arg(argptr,cell));
          break;
         case ‘d’ :
          print_int((int) va_arg(argptr,int));
          break;
        case ‘f’ :
          print_float((float) va_arg(argptr,float));
          break;
        default :
          print_char(f[1]);
      }
      f++;
    }
    else
      print_char(f[0]);
    va_end(argptr);
}
```

```
my_printf(“%d: Hello, %s!”, cnt++, UserName);
```

Переменное число параметров - недостатки (С)

- my_printf(“%s + %s = ?”, UserName, 0.7L);
  - Несоответствие типов параметров формату
- my_printf(“%f + %f = %f”, x, y);
  - Несоответствие количества параметров формату
- (Почти) невозможно передать все параметры другой процедуре с переменным числом параметров, например, printf

Переменное число параметров (Visual Basic)

```
Function Average(ParamArray A() As Single) _
          As Single
  If UBound(A) = 0 Then
    Average = 0
    Exit Function
  End if
  Dim i As Integer
  Dim s As Single = 0;
  For i = 1 To UBound(A)
    s = s + A(i)
   Next i
   Average = s / UBound(A)
End Function

Print Average()
Print Average(1)
Print Average(2, 3, 5.7, 11.13, 17, 19)
```

- Контроль типов
- Проверка выхода индексов за границы массива

Необязательные и именованные параметры

```
void DrawBox(
   long Left,
   long Top,
   long Width,
   long Height,
   long OffsetX,
   long OffsetY,
   int BorderWidth,
   long BorderColor,
   unsigned char Fill,
   long FillColor,
   int Transparency)
{    ….
}
```

Чему равны параметры?

```
DrawBox(100, 200,50,100,0,0,
        1,0,1,16777215, 50);
```

«Если в Вашей процедуре 17 параметров, то скорее всего одного не хватает»

Необязательные и именованные параметры (Visual Basic)

```
Sub DrawBox( _
    Left As Long, _
    Top As Long, _
    Width As Long, _
    Height As Long, _
    Optional OffsetX As Long = 0, _
    Optional OffsetY As Long = 0, _
    Optional BorderWidth As Integer = 1, _
    Optional BorderColor As Long = vbBlack, _
    Optional Fill as Boolean = True,
    Optional FillColor As Long = vbWhite, _
    Optional Transparency As Integer = 0)
  ….
End Sub
```

```
DrawBox 100,200, _
10,10,,,,, vbRed
```

```
DrawBox _
  Left:=100, _
  Width:=10, _
  Top:=200, _
  Height:=10, _
  FillColor:=vbRed
```

Подстановка параметров по ссылке

- Доступ во время исполнения к объектам, переданным параметрами:

```
int A, B;
void  swap(int x, int y)
{
   x += y; y = x – y; x -= y;
}
…
A = 1; B = 2; swap(A,B);
printf("A=%d, B=%d\n",A,B);
```

```
int A, B;
void  swap(int * x, int * y)
{
   *x += *y; *y = *x – *y; *x -= *y;
}
…
A = 1; B = 2; swap(&A,&B);
printf("A=%d, B=%d\n",A,B);
```

Состояние после x += y;

A	1
B	2

swap:

x	3
y	2

Состояние после *x += *y;

A	3
B	2

swap:

x	
y	

Подстановка параметров по ссылке

- Процедуры с несколькими результатами:

C:

```
void ToPolar(
	double x,
	double y,
	double * r,
	double * a)
{
  * r = sqrt(x*x + y*y);
  * a = atan2(x,y)
}
```

Pascal:

```
procedure ToPolar(
	x : double;
	y : double;
	var r : double;
	var a : double)
{
  r : = sqrt(x*x + y*y);
  a := atan2(x,y)
}
```

Visual Basic:

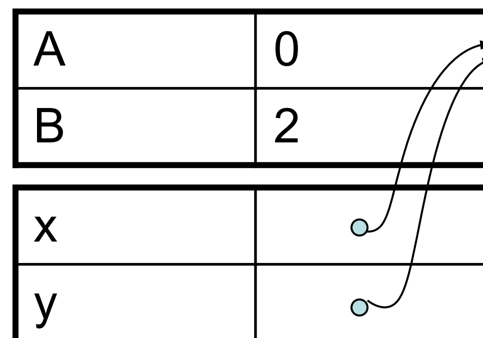
```
procedure ToPolar(
	ByVal x As Double;
	ByVal y As Double,
	r As Double,
	a As Double)
{
  r = sqrt(x*x + y*y);
  a = atan2(x,y)
}
```

Подстановка параметров по ссылке

- Проблема синонимов

```
int A,B;
void  swap(int * x, int * y)
{
   *x += *y; *y = *x – *y; *x -= *y;
}
…
A = 1; B=2; swap(&A,&A);
printf(“A=%d, B=%d\n”,A,B);
```

Состояние после *x += *y; *y = *x – *y;



A, *x, *y – *синонимы*, поскольку обозначают один и тот же объект

Подстановка параметров по ссылке

- Проблема синонимов
  - Затрудняет понимание программ
  - Мешает оптимизации, поскольку приходится предполагать, что параметр может указывать куда угодно

```
int x;
void  P(int a, int * b)
{
   a += 7;
   *b *= 2;
   x += a + *b;
}
…
x = 2;
P(x,&x);
printf(“%d”, x);
…
```

Результат?

17

Подстановка параметров по значению-результату

```
int A,B;
void  swap(int * x, int * y)
{
   *x += *y; *y = *x – *y; *x -= *y;
}
…
A = 1; B=2; swap(&A,&A);
printf(“A=%d, B=%d\n”,A,B);
```

```
int A,B;
void  swap(int * x, int * y)
{
   int xx = *x, yy = *y;
   xx += yy; yy = xx – yy; xx -= yy;
   *x = xx; *y = yy;
}
…
A = 1; B=2; swap(&A,&A);
printf(“A=%d, B=%d\n”,A,B);
```

Состояние после *x += *y; *y = *x – *y;

A	1
B	2

swap:

x	→ A
y	→ A
xx	3
yy	1

Строгое vs нестрогое вычисление

- *Строгое* – фактические параметры полностью вычисляются до применения функции
- *Нестрогое* – не вычисляются до тех пор, пока не потребуются

```
float IfSqr(int cond, float t, float f)
{
   if (cond)
      return t * t;
   else
      return f * f;
}

printf("%f", IfSqr(x==0, 0, 1/x));
```

Подстановка параметров по имени

- *thunk* – функция без параметров, вычисляющая значение фактического параметра
- Нестрогие параметры вычисляются только при обращении
- Строгие параметры вычисляются до обращения к функции при любом исполнении

```
float IfSqr(int cond, float * t(), float * f())
{
    if (cond)
        return (*t)() * (*t)();
   else
       return (*f)() * (*f)();
}

float t_thunk() {  return 0; }
float f_thunk() { return 1/x; }

printf("%f", IfSqr(x==0, t_thunk, f_thunk));
```

Подстановка параметров по имени

- *Всегда выдаёт ответ*, если он существует, так как не зацикливается, если зацикливается вычисление неиспользуемого фактического параметра
- Может приводит к *дублированию вычислений*, если параметр используется в теле несколько раз

```
float Triple(x)
{
    return x * x * x;
}

printf("%f", triple(triple(triple(sqrt(x*x+1)))));
```

Сколько раз вызывается sqrt?

1

Подстановка параметров по имени

- *Всегда выдаёт ответ,* если он существует, так как не зацикливается, если зацикливается вычисление неиспользуемого фактического параметра
- Может приводит к *дублированию вычислений*, если параметр используется в теле несколько раз

```
float Triple(float * x())
{
    return (*x)() * (*x)() * (*x)();
}

float x_thunk() {  return sqrt(x*x+1); }
float t1_thunk() { return triple(x_thunk); }
float t2_thunk() { return triple(t1_thunk); }

fprinf(“%f”, triple(t2_thunk);
```

Сколько раз вызывается sqrt?

27

Подстановка параметров по необходимости

- То же, что и передача параметров по имени, но после первого вычисления значение параметра запоминается
- Результат может отличаться, если в промежутке между обращениями к параметру его значение изменяется.

```
float Triple(float * x())
{
    return (*x)() * (*x)() * (*x)();
}

int x_done, t1_done, t2_done;
float x_val, t1_val, t2_val;
float x_thunk()
{
  return x_done ? x_val
      : (x_done=1, x_val = sqrt(x*x+1)); }
float t1_thunk() { … }
float t2_thunk() { … }

x1_done = t1_done = t2_done = 0;
printf("%f", triple(t2_thunk));
```

Функции обратного вызова (callback)

- Интеграл

$$\int_{0}^{\pi} \sin(x)dx$$

$$\int_{-\pi .2}^{\pi / 2} \cos(x)dx$$

```
extern double sin(double x);
extern double cos(double x);

double Integral(double x1, double x2,
  double h, double (*f)(double x))
{
   double s = 0;
   for (double x=x1+h/2; x < x2; x+=h)
      res += (*f)(x);
   return res * h;
}

main()
{
  printf("sin[0..pi] = %e, cos[-pi/2...pi/2]= %e\n",
      Integral(0,1,0.01,sin),
      Integral(-0.5,0.5,0.001,cos));
}
```

Функции обратного вызова (callback)

- Сортировка
  – Упорядочить строки вещественной матрицы по значению скалярного произведения с вектором v;

```
extern void qsort (void * base,
        size_t num, size_t size,
        int (*cmp) (void *,void *));

float v[N];
float M[N][N];

float SProd(float * x, float  *y)
{
   float sum = 0;
   for (int i=0; i<N; i++)
      sum += x[i] * y[i];
}
```

```
int LineCmp(void * x; void * y)
{
   float v = ScProd((float*) x, v)
              - ScProd((float*) y, v);
   return (v==0 ? 0 : v>0 ? 1 : -1);
}
main()
{
   …
   qsort(M, N, N*sizeof(float), LineCmp);
}
```

```
qsort( )
{
   … cmp(…) …
}
```

Функции - реализация

```
float poly(float coef[], int n,float x)
{
   float sum = 0f;
   for (int i=0; i<=n; i++)
      sum += coef[i] * power(i.x);
   return sum;
}
float power(int n, float x)
{
  return n==0 ? 1 : x*power(n-1,x);
}
void main()
{
  float binom[] = {1,2,1};
  printf(“%d”, poly(binom,2,10.0));
}
```

План:

1. Упрощение выражений
2. Функции в процедуры
3. Процедуры с одним параметром
4. Стек, процедуры без параметров
5. Переход по переменным меткам, без процедур
6. Оптимизация стека

1. Упрощение выражений

- Цель: эксплицировать последовательность выполнения побочных эффектов в выражении
- Метод: разбить выражение на последовательность «элементарных» операторов присваивания
  - параметр вызова, возвращаемое значение, индекс или аргумент операции – переменная или константа
  - временные переменные для хранения промежуточных результатов
  - условные выражения – в условные операторы
- Свойство: в любом операторе не более одного присваивания или вызова

Упрощение выражений

<table>
<tr><td>sum += coef[i] * power(i,x);</td><td>float t1 = power(i,x);
sum += coef[i] * t1;</td></tr>
<tr><td>return n==0 ? 1 : x*power(n-1,x);</td><td>if (n==0)
return 1;
else
{
int n1 = n-1;
float t2 = power(n1,x);
float t3 = x * t2;
return t3;
}</td></tr>
<tr><td>printf(“%d”, poly(binom,2,10.0));</td><td>float t4 = poly(binom,2,10.0));
printf(“%d”,t4);</td></tr>
</table>

Результат

```
float poly(float coef[], int n,float x)
{
   float sum = 0f;
   float t1;
   int i;
   for (i=0; i<=n; i++)
   {
      t1 = power(i,x);
      sum += coef[i] * t1;
   }
   return sum;
}

float power(int n, float x)
{
   float t2,t3;
   int n1;
```

```
   if (n==0)
      return 1;
   else
   {
      n1 = n-1;
      t2 = power(n1,x);
      t3 = x * t2;
      return t3;
    }
}
void main()
{
   float binom[] = {1,2,1};
   float t4;
   t4 = poly(binom,2,10.0));
   printf("%d",t4);
}
```

2. Функции в процедуры

- Любой вызов функции имеет вид
  float t = F(…);
- Метод:
  - Добавление параметра, передаваемоего по ссылке
  - замена оператора **return** присваиванием

Функции в процедуры

<table>
<tr><td>float poly(float coef[], int n,float x)
{
   …
   return sum;
}</td><td>float poly(float coef[], int n, float x,
    float * res;)
{
 …
  {* res = sum; return;}
}</td></tr>
<tr><td>t4 = poly(binom,2,10.0));</td><td>poly(binom,2,10.0, &t4));</td></tr>
</table>

Функции в процедуры

<table>
<tr><td>

```
float power(int n, float x)
{
   …
   return 1;
   …
   return t3;
   …
}
```

</td><td>

```
void power(int n, float x, float *res)
{
   …
   {* res = 1; return;}
   …
   {* res = t3; return;}
   …
}
```

</td></tr>
<tr><td>

```
t1 = power(i,x);
t2 = power(n1,x);
```

</td><td>

```
power(i,x,&t1);
power(n1,x,&t2);
```

</td></tr>
</table>

Результат

```
void poly(float coef[], int n,float x,
    float * res)
{
   float sum = 0f;
   float t1;
   int i;
   for (i=0; i<=n; i++)
   {
      power(i,x, &t1);
      sum += coef[i] * t1;
   }
   * res = sum;
}

void power(int n, float x, float * res)
{
   float t2,t3;
   int n1;
   if (n==0)
      *res = 1;
   else
   {
      n1 = n-1;
      power(n1,x,&res);
      t3 = x * t2;
      * res = t3;
   }
}
void main()
{
   float binom[] = {1,2,1};
   float t4;
   poly(binom,2,10.0, &t4));
   printf("%d",t4);
}
```

3. Процедуры с одним параметром

- Метод:
  - Все локальные данные процедуры собрать в структуру - *фрейм*
  - Размещать фрейм и заполнять перед вызовом
  - В теле процедуры обращения к локальным переменным заменить на обращения к полям фрейма
  - Удалять фрейм в конце тела процедуры

Процедуры с одним параметром (poly)

void poly(float coef[], int n,
float x, float * res)
{
float sum = 0f;
float t1;
int i;
…
}
	struct polyFrame
	{
	float * coef;
	int n;
	float x;
	float * res;
	float sum;
	float t1;
	int i;
	}
	void poly(struct polyFrame * f)
	{
	f->sum = 0f;
	…
	free(f);
	}

Процедуры с одним параметром (poly)

<table>
<tr><td>poly(binom,2,10.0, &t4));</td><td>struct polyFrame * a;
new(a);
a->coef = f->binom;
a->n = 2;
a->x = 10.0;
a->res = &(f->t4);
poly(a);</td></tr>
</table>

Процедуры с одним параметром (power)

void power(int n, float x, float * res)
{
   float t2,t3;
   int n1;
   …
}
	struct powerFrame
	{
	   int n;
	   float x;
	   float * res;
	   float t2;
	   float t3;
	   int n1;
	}
	void power(struct powerFrame * f)
	{
	…
	free(f);
	}

Процедуры с одним параметром (power)

power(i,x, &t1);	struct powerFrame * a;
	new(a);
	a->n = f->i;
	a->x = f->x;
	a->res = &(f->t1);
	power(a);

power(n1,x, &t2);	struct powerFrame * a;
	new(a);
	a->n = f->n1;
	a->x = f->x;
	a->res = &(f->t2);
	power(a);

Процедуры с одним параметром (main)

<table>
<tr>
<td>

```
void main()
{
   float binom[] = {1,2,1};
   float t4;
   …
}
```

</td>
<td>

```
struct mainFrame
{
   float * binom;
   float t4;
}

void main_helper(struct mainFrame * f)
{
…
    free(f);
}
```

</td>
</tr>
</table>

Процедуры с одним параметром (poly)

void main()
{
  float binom[] = {1,2,1};
  …
}
	void main()
	{
	  float binom[] = {1,2,1};
	  struct mainFrame * a;
	  a->binom = binom;
	  new(a);
	  main_helper(a);
	}

Результат (1)

```
void poly(struct polyFrame * f)
{
   f->sum = 0f;
   for (f->i = 0; f->i <= f->n;  f->i++)
   {
      struct powerFrame * a;
      new(a);
      a->n = f->i;
      a->x = f->x;
      a->res = &(f->t1);
      power(a);
      f->sum += f->coef[i] * f->t1;
   }
   * (f->res) = f->sum;
   free(f);
}
```

```
void power(struct powerFrame *f)
{
   if (f->n==0)
      * (f->res) = 1;
   else
   {
      f->n1 = f->n-1;
      struct powerFrame * a;
      new(a);
      a->n = f->n1;
      a->x = f->x;
      a->res = &(f->t2);
      power(a);
      f->t3 = f->x * f->t2;
      * (f->res) = f->t3;
   }
   free(f);
}
```

Результат (2)

```
void main_helper(
              struct mainFrame * f)
{
  struct polyFrame * a;
  new(a);
  a->coef = f->binom;
  a->n = 2;
  a->x = 10.0;
  a->res = &(f->t4);
  poly(a);
  printf(“%d”,f->t4);
  free(f);
}
```

```
void main()
{
  float binom[] = {1,2,1};
  struct mainFrame * a;
  new(a);
  a->binom = binom;
  main_helper(a);
}
```

4. Стек, процедуры без параметров

- В каждой процедуре есть доступ только к одному фрейму
- Метод:
  - в каждом фрейме хранить ссылку на фрейм вызывающей процедуры
  - поскольку вызовы могут быть из разных процедур, использовать явное приведение типов
  - ссылка на текущий фрейм – глобальная

Стек, процедуры без параметров

```
union Frame
{
   struct polyFrame * poly;
   struct powerFrame * power;
   struct mainFrame * main;
} f, a;
```

```
struct powerFrame
{
   int n;
   float x;
   float * res;
   float t2;
   float t3;
   int n1;
   union Frame parent;
};
```

```
struct polyFrame
{
   float * coef;
   int n;
   float x;
   float * res;
   float * sum;
   float t1;
   int i;
   union Frame parent;
};
```

```
struct mainFrame
{
   float * binom;
   float t4;
   union Frame parent;
};
```

Стек, процедуры без параметров (poly)

void poly(struct polyFrame * f)
{
  …
  {
    struct powerFrame * a;
    new(a);
    …
    power(a);
    …
  }
  …
  free(f);
}
	void poly()
	{
	  …
	  {
	   new(a.power);
	    …
	   a.power->parent = f; f=a; power();
	    …
	  }
	  …
	  a=f.poly->parent; free(f.poly); f = fa;
	}

Результат (1)

```
void poly()
{
   f.poly->sum = 0f;
   for (f.poly->i = 0;
      f.poly->i <= f.poly->n;  f.poly->i++)
   {
      new(a.power);
      a.power->n = f.poly->i;
      a.power->x = f.poly->x;
      a.power->res = &(f.poly->t1);
      a.power->parent = f; f = a; power();
      f.poly->sum +=
           f.poly->coef[i] * f.poly->t1;
   }
   * (f.poly->res) = f.poly->sum;
   a=f.poly->parent; free(f.poly); f = a;
}
```

```
void power()
{
   if (f.power->n==0)
      * (f.power->res) = 1;
   else
   {
      f.power->n1 = f.power->n-1;
      new(a.power);
      a.power->n = f.power->n1;
      a.power->x = f.power->x;
      a.power->res = &(f.power->t2);
      a.power->parent = f; f = a; power();
      f.power->t3 =
            f.power->x * f.power->t2;
      * (f.power->res) = f.power->t3;
   }
   a=f.power->parent; free(f.poly); f = a;
}
```

Результат (2)

```
void main_helper()
{
   new(a.poly);
   a.poly->coef = f.main->binom;
   a.poly->n = 2;
   a.poly->x = 10.0;
   a.poly->res = &(f.main->t4);
   f = a;
   a.poly->parent = f; f = a; poly();
   printf("%d",f.main->t4);
   a=f.main->parent; free(f.main);f = a;
}
```

```
void main()
{
  float binom[] = {1,2,1};
  new(a.main);
  a.main->binom = binom;
  a.main->parent = f; f = a; main_helper();
}
```

5. Без процедур

- От вызова процедуры остался только переход к выполнению телу
- Возврат зависит от того, откуда вызвали
- Метод:
  - Заголовок процедуры заменить на метку
  - После каждого вызова поставить метку
  - В каждом фрейме поле – *метка возврата,* заполняемое перед переходом к телу
  - В конец процедуры – переход по значению метки возврата

Без процедур

```
label e;
```

```
struct polyFrame
{
   float coef[];
   int n;
   float x;
   float * res;
   float * sum;
   float t1;
   int i;
   Frame parent;
   label exit;
}
```

```
struct powerFrame
{
   int n;
   float x;
   float * res;
   float t2;
   float t3;
   int n1;
   Frame parent;
   label exit;
}
```

```
struct mainFrame
{
   float * binom;
   float t4;
   Frame parent;
   label exit;
}
```

Без процедур

void poly()
{
  …
  {
    …
    power();
    …
  }
  …
  a=f.poly->parent; free(f); f = a;
}
	poly:
	  …
	  {
	    …
	   f.power.exit = L1; goto power; L1:
	   …
	  }
	  …
	  e=f.poly->exit;
	  a=f.poly->parent; free(f); f = a;
	  goto e;

Результат (1)

```
poly:
  f.poly->sum = 0f;
  for (f.poly->i = 0;
    f.poly->i <= f.poly->n;  f.poly->i++)
  {
    new(a.power);
    a.power->n = f.poly->i;
    a.power->x = f.poly->x;
    a.power->res = &(f.poly->t1);
    a.power->parent = f; f = a;
    f.power.exit = L1; goto power; L1:
    f.poly->sum +=
        f.poly->coef[i] * f.poly->t1;
  }
  * (f.poly->res) = f.poly->sum;
  e=f.poly.exit;
  a=f.poly->parent; free(f.poly); f = a;
  goto e;
```

```
power:
  if (f.power->n==0)
    * (f.power->res) = 1;
  else
  {
    f.power->n1 = f.power->n-1;
    new(a.power);
    a.power->n = f.power->n1;
    a.power->x = f.power->x;
    a.power->res = &(f.power->t2);
    a.power->parent = f; f = a;
    f.power.exit = L2; goto power; L2:
    f.power->t3 =
        f.power->x * f.power->t2;
    * (f.power->res) = f.power->t3;
  }
  e=f.power.exit;
  a=f.power->parent; free(f.power);f = a;
  goto e;
```

Результат (2)

```
main_helper :
   new(a.poly);
   a.poly->coef = f.main->binom;
   a.poly->n = 2;
   a.poly->x = 10.0;
   a.poly->res = &(f.main->t4);
   f = a;
   a.poly->parent = f; f = a;
   f.poly.exit = L3; goto poly; L3:
   printf("%d",f.main->t4);
   e=f.main.exit;
   a=f.main->parent; free(f.main);f = a;
   goto e;
```

```
void main()
{
   float binom[] = {1,2,1};
   new(a.main);
   a.main->binom = binom;
   a.main->parent = f; f = a;
   f.main.exit = L0; goto main_helper;
L0:
   return;
main_helper :
   …
   goto e;
poly :
   …
   goto e;
power :
   …
   goto e;
}
```

Реализация вычисляемых меток (GCC)

- Расширение C:
  - унарный оператор && - адрес метки;
  - тип «метка»: void * e;
  - переход по вычисляемой метке goto *e;

```
label e;

f.power.exit = L1;

e=f.poly.exit;

goto e;
```

```
void * e;

f.power.exit = &&L1;

e=f.poly.exit;

goto *e;
```

Реализация вычисляемых меток

- Для каждой процедуры известно множество меток возврата
- Метод:
  - «Вычисляемая метка» имеет тип перечисления
  - Переход по вычисляемой метке заменить на переключатель
  - В случае, если возможна единственная метка возврата, то не хранить и возврат заменить на goto

Реализация вычисляемых меток

```
struct polyFrame
{
   float coef[];
   int n;
   float x;
   float * res;
   float * sum;
   float t1;
   int i;
   Frame parent;
}
```

```
struct powerFrame
{
   int n;
   float x;
   float * res;
   float t2;
   float t3;
   int n1;
   Frame parent;
   enum (eL1,eL2) exit;
}
```

```
struct mainFrame
{
   float * binom;
   float t4;
   Frame parent;
}
```

Реализация вычисляемых меток

<table>
<tr>
<td>

```
power:
   …
   {
      …
f.power.exit = L2; goto power; L2:
      …
   }
   e=f.power.exit;
   a=f.power->parent; free(f.power); f = a;
   goto e;
```

</td>
<td>

```
power:
   …
   {
…
f.power.exit = eL2; goto power; L2:
      …
   }
   e=f.power.exit;
   a=f.power->parent; free(f.power); f = a;
   switch (e)
   {
      case eL1 : goto L1;
      case eL2 : goto L2;
   }
```

</td>
</tr>
</table>

Реализация вычисляемых меток

<table>
<tr>
<td>

```
poly:
  …
  {
    …
   f.power.exit = L1; goto power;  L1:
   …
  }
  …
  e=f.poly->exit;
  a=f.poly->parent; free(f.poly); f = a;
  goto e;
```

</td>
<td>

```
poly:
  …
  {
    …
   f.power.exit = eL1; goto power;  L1:
   …
  }
  …
  a=f.poly->parent; free(f.poly); f = a;
  goto L3;
```

</td>
</tr>
</table>

Реализация вычисляемых меток

main_helper :	main_helper :
…	…
f.poly.exit = eL3; goto poly; L3:	goto poly; L3:
…	…
e=f.main.exit;	e=f.main.exit;
a=f.main->parent; free(f.main);f = a;	a=f.main->parent; free(f.main);f = a;
goto e;	goto L0;

Перемещение кода

```
void main()
{
  float binom[] = {1,2,1};
  new(a.main);
  a.main->binom = binom;
  a.main->parent = f; f = a;
  new(a.poly);
  a.poly->coef = f.main->binom;
  a.poly->n = 2;
  a.poly->x = 10.0;
  a.poly->res = &(f.main->t4);
  f = a;
  a.poly->parent = f; f = a;
  f.poly->sum = 0f;
  for (f.poly->i = 0;
      f.poly->i <= f.poly->n;  f.poly->i++)
  {
    new(a.power);
    a.power->n = f.poly->i;
    a.power->x = f.poly->x;
    a.power->res = &(f.poly->t1);
    a.power->parent = f; f = a;
    f.power.exit = eL1; goto power; L1:
    f.poly->sum += f.poly->coef[i] * f.poly->t1;
  }
  * (f.poly->res) = f.poly->sum;
  e=f.poly.exit;
  a=f.poly->parent; free(f.poly); f = a;
  printf("%d",f.main->t4);
  e=f.main.exit;
  a=f.main->parent; free(f.main);f = a;
  return;

power:
  if (f.power->n==0)
    * (f.power->res) = 1;
  else
  {
    f.power->n1 = f.power->n-1;
    new(a.power);
    a.power->n = f.power->n1;
    a.power->x = f.power->x;
    a.power->res = &(f.power->t2);
    a.power->parent = f; f = a;
    f.power.exit = eL2; goto power; L2:
    f.power->t3 = f.power->x * f.power->t2;
    * (f.power->res) = f.power->t3;
  }
  e=f.power.exit;
  a=f.power->parent; free(f.power); f = a;
  switch (e)
  {
    case eL1 : goto L1;
    case eL2 : goto L2;
  }
}
```

6. Реализация стека

- Более эффективный метод управления памятью, ориентированный на специфику времени жизни локальных объектов
- Все фреймы хранятся в одном (подвижном) байтовом массиве
- Указатель на свободное место fp

```
char Stack[10000];
long fp; // Frame Pointer

void * StackAlloc(int size)
{
    void * f =  &(Stack[fp]);
    fp += size;
    return f;
}
void StackFree(int size)
{
   fp -= size;
}
#define stack_new(p) p=StackAlloc(sizeof(*p))
#define stack_free(p) StackFree(sizeof(*p))
```

Реализация стека

poly:
…
new(a.power);
...
a=f.poly->parent; free(f.poly); f = a;
…
	poly:
	…
	stack_new(a.power);
	…
	a=f.poly->parent;
	stack_free(f.poly);
	f = a;
	…

Выход из глубокой рекурсии

```
procedure ReadEvalPrint;
  label ErrExit;
  function Eval(e : Expr) : integer;
  begin
      …
       '/' :
         v1 := Eval(e^.left);
         v2 := Eval(e^.right);
         if v2 = 0 then
          begin
            WriteLn('Деление на ноль');
            goto ErrExit;
          end;
     ...
  end; (* Eval *)
```

```
  var s : string;
  begin
     WriteLn('Привет!');
     while true do
     begin
       Write('>');
       ReadLn(s);
       if  s = '.' then
          break;
       WriteLn(Eval(ParseExpr(s)));
ErrExit:
     end;
     WriteLn('Пока.');
  end; (* ReadEvalPrint *)
```

(1 + (2 * ((3/(2-(1+1))) - 4)))

Выход из глубокой рекурсии (С)

```
long Eval(Expr e)
{
   …
   case ‘/’ :
      v1 = Eval(e->left);
      v2 = Eval(e->right);
      if (v2 == 0)
      {
        fputs(”Деление на ноль”,stderr);
        goto ErrExit;
      }
   …
}  // Eval
```

```
void ReadEvalPrint()
{
   char s[256];
   fputs(“Привет!”,stderr);
   for (;;)
   {
      fputc(‘>’,stderr);
      fgets(s,stderr);
      if (s[0]==‘.’)
         break;
      fprintf(stderr, ”%d\n”,
                  Eval(ParseExpr(s)));
ErrExit: ;
   }
   fputs(“Пока.”,stderr);
} // ReadEvalPrint
```

Неверно: метки локальны

Выход из глубокой рекурсии (C)

```
void ReadEvalPrint()
{
   char s[256];
   int res;
   fputs(“Привет!”,stderr);
   for (;;)
   {
      fputc(‘>’, stderr);
      fgets(s, stderr);
      if (s[0]==‘.’)
         break;
      if (Eval(ParseExpr(s), & res))
          fprintf(stderr,”%d\n”, res);
   }
   fputs(“Пока.”,stderr);
} // ReadEvalPrint
```

```
int Eval(Expr e, long * res)
{
   …
   case ‘/’ :
      if (! Eval(e->binop.left, &v1))
        return 0;
      if (! Eval(e->binop.right), &v2))
        return 0;
      if (v2 == 0)
      {
        fputs(”Деление на ноль”,stderr);
        return 0;
      }
   …
   return 1;
}  // Eval
```

Код ошибки (1)

```
#define EVAL_OK 0
#define EVAL_ERRDIV0 1
#define EVAL_ERROVERFLOW 2
```

```
int Eval(Expr e, long * res)
{
  int ec; // Error Code
   …
   case ‘/’ :
      if (ec=Eval(e->binop.left, & v1))
         return ec;
      if (ec=Eval(e->binop.right), &v2))
         return ec;
      if (v2 == 0)
          return EVAL_ERRDIV0;
   …
   return EVAL_OK;
}  // Eval
```

Код ошибки (2)

```
#define EVAL_OK 0
#define EVAL_ERRDIV0 1
#define EVAL_ERROVERFLOW 2

void ReadEvalPrint()
{
   char s[256];
   int res;
   fputs(“Привет!”,stderr);
   for (;;)
   {
      fputc(‘>’, stderr);
      fgets(s, stderr);
      if (s[0]==‘.’)
         break;
```

```
   switch (Eval(ParseExpr(s), & res))
    {
    case 0 :
      fprintf(stderr,”%d\n”, res);
      break;
    case EVAL_ERRDIV0 :
      fputs(”Деление на ноль”,stderr);
      break;
    case EVAL_ERROVERFLOW :
      …
      break;
    }
  }
  fputs(“Пока.”,stderr);
} // ReadEvalPrint
```

Нелокальные переходы

- #include < setjmp.h >
- int setjmp(jmp_buf env);
  – запоминает обстановку вычислений и возвращает 0.
- void longjmp(jmp_buf env, int val);
  – восстанавливает запомненную *setjmp* обстановку вычислений
  – возвращается в то место, где *setjmp* собирался вернуть 0
  – заставляет *setjmp* выдать *val* вместо 0.

Нелокальные переходы – пример (1)

```
#include <setjmp.h>

#define EVAL_OK 0
#define EVAL_ERRDIV0 1
#define EVAL_ERROVERFLOW 2

//данные для нелокального перехода
jmp_buf env;

void ReadEvalPrint()
{
   char s[256];
   int res;
   fputs("Привет!",stderr);
   for (;;)
   {
      fputc('>', stderr);
      fgets(s, stderr);
      if (s[0]=='.')
         break;
```

```
      int ec; // Error Code;
      if ((ec = setjmp(env)) == 0)
         fprintf(stderr,"%d\n",
                Eval(ParseExpr(s)));
      else
        switch (ec)
        {
          case EVAL_ERRDIV0 :
            fputs("Деление на ноль",
                      stderr);
            break;
          case EVAL_ERROVERFLOW :
            …
            break;
        }
   }
   fputs("Пока.",stderr);
} // ReadEvalPrint
```

Нелокальные переходы – пример (2)

```
#define EVAL_OK 0
#define EVAL_ERRDIV0 1
#define EVAL_ERROVERFLOW 2

//данные для
// нелокального перехода
jmp_buf env;
```

```
long Eval(Expr e)
{
  int ec; // Error Code
   …
   case '/' :
      v1 = Eval(e->binop.left);
      v2 = Eval(e->binop.right);
      if (v2 == 0)
         longjmp(jmp_buf,
                        EVAL_ERRDIV0);
   …
}  // Eval
```

Нелокальные переходы

- Дают возможность обработки исключительных ситуаций
- setjmp, longjmp – не являются процедурами в обычном смысле
- Позволяют вернуть только код ответа
- Крайне опасны при неаккуратном использовании
  - например, переход внутрь процедуры, выполнение которой уже закончилось
- В современных языках есть специальные средства обработки исключительных ситуаций (exception, try-блоки)

Классы памяти

- auto - автоматическая память (умолчание, на практике не используется)
- static – глобальная по времени жизни, локальная по области видимости (own в Algol-68)
- register – предписание компилятору использовать регистр (не следует использовать)
- extern – указание внешнего объекта

Классы памяти (static)

```
void PrintLine(char * s)
{
  static int counter = 0;
  printf(“%d:\t%s\n”, counter ++,
     s);
}

char * First10(char *s)
{
  static char buffer[11];
  strncpy(buffer, s, 10);
  return buffer;
}
```

Достоинства

- Ограниченная область видимости

Недостатки

- Примитивная инициализиция
- Видимость только в одной функции

Решение

- Модули
- Объекты

Модули (static, extern)

stack.c

```
#define MAXDEPTH 128
static void * buffer[MAXDEPTH];
static int depth = 0;

int Empty()
{
  return depth == 0;
}
void Push(void * e)
{
   buffer[depth++] = e;
}
void * Pop()
{
      return buffer[--depth];
}
```

stack.h

```
extern int Empty();
extern void Push(void * e);
extern void * Pop();
```

main.c

```
#include “stack.h”
#include “stdio.h”

void main(int ac, char * av[])
{
    for (int i=0; i<ac; i++)
       Push(av[i]);
    while (! Empty())
       printf(“%s\n”,Pop());
}
```

Модули С

Недостатки

- два раздельных файла - .c, .h
  - нет согласования
  - хотя может быть и достоинство для отслеживания изменения спецификации модуля (make)
- отсутствие иерархии
  - например, невозможно выразить свойство «видимо только внутри данной библиотеки»
  - следствие – увеличение размера модулей
- extern – по умолчанию