Funciones

Agustín J. González
Versión Original de Kip Irvine
ELO320: Seminario II
2do. Sem 2001

Parámetros

- Un *parámetro actual* (o valor, argumento) es el valor pasado a la función desde el código invocador.
- Un *parámetro formal* (o variable) es la variable declarada en la lista de parámetros de la función que recibe los valores.
  - Parámetros de entrada reciben los valores pasados a la función.
  - Parámetros de salida retornan valores desde la función al código invocador.

Función Valor Futuro

Esta función calcula el valor futuro de una inversión de 1000 sobre 10 años y calculado mensualmente. No es muy flexible, pero se conserva como una sola tarea.

Parámetro formal

```
double future_value( double p )
{
  double b = 1000 *
             pow(1 + p / (12 * 100), 12 * 10);
  return b;
}
```

Función Valor Futuro

La siguiente secuencia llama a future_value():

```
cout << "Please enter the interest rate "
        " in percent:";
double rate;
cin >> rate;

double balance = future_value(rate);
cout << "After 10 years, the balance is "
     << balance << "\n";
```

Haciendo al Función más Flexible

Esta versión de future_value() es más flexible porque el balance inicial y el número de años son pasados como parámetros actuales.

```
double future_value(double initial_balance,
       double p, int nyear)
{
  double b = initial_balance
         * pow(1 + p / (12 * 100), 12 * nyear);

  return b;
}
```

Uso de Comentarios

Esta versión de future_value tiene comentarios según se espera sus tareas de programación:

```
double future_value(double initial_balance,
       double p, int nyear)
// Propósito: calcula el valor de la inversión
//          usando  interés compuesto
// Recibe:
//   initial_balance - El valor inicial de la
//                  inversión.
//   p      - la tasa de interés en porcentaje
//   nyear - el número de años que la
//           inversión será mantenida
// Retorna:  el balance despues de nyear años
// Observaciones:  el interés es actualizado
//           mensuamnete
```

Retorno cuando encontramos un Error

Use la sentencia return para terminar inmediatamente. En el ejemplo, errores de rango son manejados retornando un valor por defecto:

```
double future_value(double initial_balance,
      double p, int nyear)
{
  if( nyear < 0 ) return 0;    // error
  if( p < 0 ) return 0;        // error

  double b = initial_balance
    * pow(1 + p / (12 * 100), 12 * nyear);
  return b;
}
```

Funciones Booleanas (1)

Funciones que retornan un valor booleano son apropiadas para validación de datos. Se recomienda centralizar sólo una tarea por función:

```
bool IsValidYear( int year )
{
  if(( year > 1600 ) and ( year < 2100 ))
    return true;
  else
    return false;
}
```

```
Alternativamente:
bool IsValidYear( int year )
{
  return(( year > 1600 ) and ( year < 2100 ));
}
```

Funciones Booleanas (2)

Las funciones booleanas pueden retornar sus valores dentro de expresiones booleanas, conduciendo así a código más fácil de entender:

```
if( IsValidYear( birthYear )
    and IsLeapYear( birthYear ))
{
  cout << "You were born in a Leap Year\n";
}
```

Prototipo de Funciones

Una declaración de función (or *prototipo*) es requerida cuando un llamado a función no ha sido precedida por una definición de función. El prototipo indica información de llamado que es esencial para el compilador.

```
double future_value(double initial_balance,
       double p, int nyear);
// function prototype
```

```
Alternativamente:
double future_value(double, double, int);
// function prototype
```

Efecto Lateral (o secundario)

Un *efecto lateral* ocurre cuando una función produce algún efecto adicional al valor retornado. Por ejemplo, se podría desplegar información o modificar una variable global.

```
double future_value(double initial_balance,
       double p, int nyear)
{
  if( nyear < 0 )
  {
    cout << "Error: invalid number of years\n";
    return 0;
  }
  double b = initial_balance
           * pow(1 + p / (12 * 100), 12 * nyear);
   return b;
}
```

Procedimientos

Un procedimiento (o función void) siempre produce un efecto lateral porque no retorna valores.

```
void DisplayReport( double initial_balance,
     double p, int nyear)
{
  cout << "An investment of " << initial_balance
   << " will be worth "
   (etc.)
```

Paso por Referencia (1)

Pasar un parámetro por referencia significa que el nombre del parámetro formal queda definido en la misma localización de memoria del parámetro actual. Así la función puede modificar la variable (parámetro actual).

Por ejemplo::

```
void swap( int & x, int & y )
// Exchange the values of two variables.
{
  int temp = x;
  x = y;
  y = temp;
}
```

Paso por Referencia (2)

Una función que recibe entradas del usuario normalmente usa parámetros por referencia porque la función no puede retornar más de un valor.

```
void GetUserName( string & lastName,
                  string & firstName )
{
  cout << "Last name: ";
  cin >> lastName;
  cout << "First name: ";
  cin >> firstName;
}
```

Paso por Referencia (3)

En el código que invoca la función, las variables pasadas a la función son actualizadas:

```
string firstName;
string lastName;
GetUserName( lastName, firstName );
// now the values are changed...
```

Parámetros de Entrada y Salida

- Parámetros de Entrada contienen datos que han sido pasados a la función desde el código llamador.
- Parámetros de Salida contienen datos que son fijados dentro de la función y retornados al código llamador.
- Parámetros de Entrada/Salida reciben datos desde el código llamador, modifica los datos, y retorna los datos al código llamador.

Parámetros de Entrada

ShowReport() usa parámetros de entrada, hours y payRate:

```
double CalcPay( double hours, double payRate )
{
  return hours * payRate;
}
```

Parámetros de Salida

GetUserName(), visto antes, usa dos parámetros de salida:

```
void GetUserName( string & lastName,
                  string & firstName )
{
  cout << "Last name: ";
  cin >> lastName;
  cout << "First name: ";
  cin >> firstName;
}
```

Parámetros de Entrada/Salida

La función swap() usa dos parámetros de entrada y salida:

```
void swap( int & x, int & y )
{
  int temp = x;
  x = y;
  y = temp;
}
```

Paso por Referencia Constante

Cuando pasamos un objeto por referencia, se recomienda usar paso por referencia constante. El compilador impide a la función llamada modificar su valor.

```
void ShowName( const string & firstName,
               const string & lastName )
{
  cout << firstName << " " << lastName;

  firstName = "George";  // error
}
```

No pases Objetos por Valor

El paso de objetos por valor desperdicia memoria y tiempo de procesamiento al crear un duplicado del objeto en el stack.

```
void ShowName( string firstName, string lastName )
{
  cout << firstName << " " << lastName;
}
```

Alcance de Variables (1)

Una variable declarada dentro de un bloque es sólo visible desde el punto donde está declarada hasta el término del bloque.

```
if( X > Y )
{                              // begin block
  int sum = X + Y;
}                              // end block

cout << sum;    // error: not visible
```

Alcance de Variables (2)

No enmascarar variables de alcance mayor dentro con otra que posee alcance interior. Si esta función recibe como entrada 10, qué retornará?

```
int Evaluate( int n )
{
  int j = 99;
  if( n > 5 )
  {
    int j = 20;
  }
  return j;
}
```

Variables Globales

Una variable *global* es declarada fuera de cualquier función. Evitar esto cuando sea posible. Nunca usarlas como una forma de evitar paso de parámetros a funciones.

```
double g_initial_balance, g_principal;
int g_nyear;


double future_value()
{
  return (g_initial_balance *
  pow(1 + g_principal / (12 * 100), 12 * g_nyear));
}
```

Uso de Precondiciones (1)

La macro assert() puede ser llamada cuando se desee garantizar absolutamente que se satisface alguna condición. Chequeo de rango es común:

```
double future_value(double initial_balance,
       double p, int nyear)
{
  assert( nyear >= 0 );
  assert( p >= 0 );
  double b = initial_balance
           * pow(1 + p / (12 * 100), 12 * nyear);
  return b;
}
```

Uso de Precondiciones (2)

Si la expresión pasada a la macro assert() es falsa, el programa se detiene inmediatamente con un mensaje de diagnóstico del tipo:

```
Assertion failure in file mysub.cpp,
    line 201:  nyear >= 0
```

Con assert el programa no tiene la posibilidad de recuperarse del error.

Para eliminar el efecto de assert se debe compilar el programa con la definición de NDEBUG para el procesador.

#define NDEBUG

Uso de Precondiciones (3)

Una mejor forma para manejar errores es usar el mecanismo de *manejo de excepciones de* C++. En este caso el programa tiene la posibilidad de recuperarse e intentar nuevamente:

```
double future_value(double initial_balance,
       double p, int nyear)
{
  if( nyear >= 0 )
    throw RangeException( "nyear", 0 );

  // etc.
```

Recursión (1)

Recursión ocurre cuando una función ya sea (a) se llama así misma, o (b) llama a otra función que eventualmente termina por llamar a la función original. Aquí, una función se llama así misma, creando lo que parece una *recursión infinita*:

```
void ChaseMyTail()
{
  //...
  ChaseMyTail();
}
```

Recursión (2)

Recursión Indirecta ocurre cuando una serie de llamados a función conduce a una función previa en la serie, creando un efecto de bucle. Esto parece trabajar pero su correcto funcionamiento puede ser un problema.

```
void One()
{
  Two();
}

void Two()
{
  Three();
}
```

```
void Three()
{
  Four();
}

void Four()
{
  One();
}
```

Recursión (3)

Recursión tiene que ser controlada proveyendo una condición de término que permite detener la recursión cuando el caso básico es alcanzado.

```
long Factorial( int n )
{
  if( n == 0 )
    return 1;
  else
  {
    long result = n * factorial( n - 1 );
    return result;
  }
}
```

Condición de térmico

Función Factorial Recursiva

Pensemos n! Como el producto de n por (n-1)!. Esto siempre es verdadero mientras n>0.

```
    5! = 5 * 4!
and 4! = 4 * 3!
and 3! = 3 * 2!
and 2! = 2 * 1!
and 1! = 1 * 0!
and 0! = 1          (por definición)
```

Fin