Functions

Dr. Donald Davendra *Ph.D.*

Department of Computing Science, FEI VSB-TU Ostrava

Function

Function

– The basic building block of a procedural program - not to be extensive,
– Addresses a complex problem,
– If the problem is too complicated, calling for support other functions,
– Can return the return value,
– Can have arguments

Fisrt C++ program:

[void, int] main()...

General:

type name(formal arguments) {
body functions
}

function definition - identifier, type of return value, and cartridge type arguments increases (between (a)) and body functions (between {and}) - code that will be called for each m the function performed ({and}) vs..

function declaration.

() vs. void **formal arguments** vs **real arguments**

Function call

name (real arguments)
actual argument - variables, constants, expression, directly mentioned constant value.
Number of *actual arguments* is determined by the needs of the programmer:
() – Zero arguments
(.....) – number of arguments
without the brackets - the address of the function (entry point function).
function return value return expression of the appropriate type;
Example

```
int isqr(int i) {
 return i * i;
}
...
 int result;

result = isqr(4);
```

Arguments of functions and their transfer

- **value** - a copy of the actual argument to a stack, changing the formal arguments in a function
- **address** - pointer to implement formal link between it and the real value,
- **link** - a link between function and value.

Functions

```
// fn-argumenty.cpp

#include <iostream>

using namespace std;

void switch_by_value_imposible(int fa, int fb) {
 int pomocna = fa; // value
 fa = fb;
 fb = pomocna;
}

void switch_by_pointer(int *fa, int *fb) {
 int pomocna = *fa; // dereference
 *fa = *fb;
 *fb = pomocna;
}

void switch_by_reference(int& fa, int& fb) {
 int pomocna = fa; // reference
 fa = fb;
 fb = pomocna;
}
```

Functions

```
void main() {
 int sfirst= 123, ssecond = -456;
 cout << "starting values " << endl;
 cout << "sfirst= " << sfirst<< ", ssecond = "
      << ssecond << endl << endl;

 switch_by_value_imposible(sfirst, ssecond);
 cout << "after 'switch_by_value_imposible()'"
      << endl;
 cout << "sfirst= " << sfirst<< ", ssecond = "
      << ssecond << endl << endl;

 switch_by_pointer(&sfirst, &ssecond);
 cout << "after 'switch_by_pointer()'" << endl;
 cout << "sfirst= " << sfirst<< ", ssecond = "
      << ssecond << endl << endl;

 switch_by_reference(sfirst, ssecond);
 cout << "after 'switch_by_refernce()'" << endl;
 cout << "sfirst= " << sfirst<< ", ssecond = "
      << ssecond << endl;
} // void main()
```

Functions

Address operator & and deference operator $*$

```
starting values
sfirst= 123, ssecond = -456

after 'switch_by_value_imposible()'
sfirst= 123, ssecond = -456

after 'switch_by_pointer()'
sfirst= -456, ssecond = 123

after 'switch_by_refernce()'
sfirst= 123, ssecond = -456
```

Parameter	Example
auto	placement of it on the stack, uninitialized;
extern	to be connected to another module;
register	placed at the register processor uninitialized;
static	placement of it into the data segment initialized with zero;

Modifier	Example
const	expresses immutability;
volatile	expresses the constant variability.

Object	memory class, interpretation, location
global.	static a extern, initialized to zero;
local.	auto, initialized, stack;
formal arg.	auto, initialized, stack;
def. fce	extern, according to the definition of the code

Recursive

Formal arguments of the function and its local variables on **stack** - LIFO *Last In First Out*

$n! = n \times (n-1)!$
$0! = 1.$

```
Enter natural number n: 5
5! = 120

Enter natural number n: 69
69! = 1.71122e+98
```

Function

```
/*******************
 * file fact-r.cpp
 * factorial by recursion
 *******************/

#include <iostream>

using namespace std;

double fact(long n) {
 if (n == 0L)
   return 1.0L;
 else
   return n * fact(n-1);
} // double fact(long n)

void main() {
 static long n;
 cout << "Enter natural number n: ";
 cin >> n;
 cout << n << "! = " << fact(n) << endl;
} // void main()
```

Function

```
// file fact-rst.cpp
#include <iostream>
using namespace std;
double fact(long n) {
 static int h;
 double navrat;
 cout << "depth= " << ++h << "\tn= "
      << n << endl;
 if (n == 0L)
   navrat = 1.0L;
 else
   navrat = n * fact(n-1);
 cout << "depth= " << h-- << "\tn= "
      << n << "\treturn= " << navrat << endl;
 return navrat;
}
void main() {
 long n;
 cout << "Enter natural number n: ";
 cin >> n;
 cout << n << "! = " << fact(n) << endl;
}
```

Function

```
Enter natural number n: 5
depth= 1 n= 5
depth= 2 n= 4
depth= 3 n= 3
depth= 4 n= 2
depth= 5 n= 1
depth= 6 n= 0
depth= 6 n= 0 return= 1
depth= 5 n= 1 return= 1
depth= 4 n= 2 return= 2
depth= 3 n= 3 return= 6
depth= 2 n= 4 return= 24
depth= 1 n= 5 return= 120
5! = 120
```

Headers and Libraries

Headers in C++ and equivalent libraries in C

<cassert> <ciso646> <csetjmp> <cstdio> <ctime>
<cctype> <climits> <csignal> <cstdlib> <cwchar>
<cerrno> <clocale> <cstdarg> <cstring> <cwctype>
<cfloat> <cmath> <cstddef>

Function Declaration – prototype:

<algorithm> <iomanip> <list> <ostream> <streambuf>
<bitset> <ios> <locale> <queue> <string>
<complex> <iosfwd> <map> <set> <typeinfo>
<deque> <iostream> <memory> <sstream> < utility >
<exception> <istream> <new> <stack> <valarray>
<fstream> <iterator> <numeric> <stdexcept> <vector>
<functional> <limits>

Function Declarations

function declarations - information for the compiler,
function definition has the memory requirements.
identifiers form. arg. to be passed to the function definition.
ISO C + + requires a prototype of each function we use.

New and Delete

Overloading **new** and **delete** variant allows the allocation or deallocation of free memory space (heap). Pre-overload the new operator must

- return a pointer of type void $*$
- have a first argument of type size_t .

Overloaded operator **delete**

- not have any return value (or its designation as a void),
- must have a pointer of type void $*$ as its first argument,
- can have a second argument of type size_t .

Implicit Functions

Default valuesof function arguments we can assign to their arguments listed in order of declaration / definition of function only from behind.

```
//declaration:
  int fn(int i = 1, int j = 2);// O.K.
  int fn(int i = 1, int j);//error

//calling:
  fn();      // same as fn(1, 2)
  fn(5);     // same as fn(5, 2)
  fn(7, 8); // as fn(7, 8), not implicit values

//definition
  int fn(int i, int j) {...
 // O.K. implicit values from declaration

  int fn(int i, int j = 8) {...
 // error, implicit values are in declaration
```

Inline Functions

Inline functions optimization for speed or size inline modifier.

- Repeated calls definitely increases the length of the resulting program.
- Placing code inline functions instead of calling time to reduce system overhead - acceleration applications, it may not:
  - Copy the actual arguments
  - Run-pass control to another place
  - after passing back the correct address back
  - Cleaning up unneeded copies of actual arguments.

Function Declaration

Function declarations

type name (*argument list*);

- type is a function return value
- name is the identifier,
- ( ) is the obligatory pair of parentheses, the declaration setting out the arguments,
- list argument is optional (no, one or more arguments), the default argument values (backward) or variable number of arguments (...). Separate the arguments with each other by commas. Each argument must have a separately identified data type.

Function Declarations

The declaration describes the function inputs and outputs (interface function), which provides a function, but does not define a sequence of commands that has a proper function.

The function does not perform actions with other data that it will pass as arguments.

Output from the function should take place only as its return value, or via arguments passed by reference or address. Otherwise, error will occur.

Declaration of custom functions in header files - no memory demands. Otherwise linker problems.

Function

```
/*************************************************
 * file fn-argn.cpp
 * function with variable number of arguments of diffrent types
 *************************************************/

#include <iostream>
#include <cstdarg>

using namespace std;

double polynom(short count, ...) {
/*************************************************
 * enumerate polynomial an*x^n + an-1*x^n-1 + ... a1*x + a0
 * count = count of coefficents (n+1), next
 * x of type double, coefficents of type int
 *************************************************/
 double value = 0.0, x;
 va_list ap;

 va_start(ap, count);
 x = va_arg(ap, double);
 while (count-- != 0)
     value = value * x + va_arg(ap, int);
 va_end(ap);
 return value;
}
```

Function

```
void main(void) {
 double x = 2.0;
 /* polynomial of 2. degree; 2x^2 + 3x + 4, x=2 -> 18 */
 double f = polynom(3, x, 2, 3, 4);
 cout << "p(" << x << ") = " << f << endl;

 x = 3.0;
 /* polynomial of 3. degree; x^3, x=3 -> 27 */
 f = polynom(4, x, 1, 0, 0, 0);
 cout << "p(" << x << ") = " << f << endl;

 x = 5.0;
 /* polynomial of 4. degree; 2x^4 - 10x^3 + 2x^2 + 3x - 1, x=5 -> 64 */
 f = polynom(5, x, 2, -10, 2, 3, -1);
 cout << "p(" << x << ") = " << f << endl;
} // void main(void)
```

Function

```
p(2) = 18
p(3) = 27
p(5) = 64
```