IIT Bombay

Computer Programming

Dr. Deepak B Phatak
Dr. Supratik Chakraborty
Department of Computer Science and Engineering
IIT Bombay

Session: Parameter Passing in Function Calls

Quick Recap of Relevant Topics

IIT Bombay

- Use of simple functions in programs
- Contract-centric view of programming with functions
- Flow of control in function call and return
- Activation records and call stack

Overview of This Lecture

- Paradigms of parameter passing in function calls
  - Call by value
  - Call by reference
- Functions without return values

Recall: Encoding Example

- We want to store quiz 1 and quiz 2 marks of CS101 students in an encoded form

- Encoding strategy:

  The ordered pair of marks (m, n) is encoded as $2^m \times 3^n$

- Assume all marks are integers in {1, 2, … 10}

IIT Bombay

Recall: C++ Program Structure

```
#include <iostream>
using namespace std;
int myEncode(int q1Marks,int q2Marks);
int power(int base, int exponent);
int main() { …
  for ( … ) {  …
   cipher = myEncode(q1Marks, q2Marks);
  …}
…
}
```

```
// PRECONDITION:  …
int myEncode(int q1Marks,
                          int q2Marks)
{ …
 twoRaisedQ1 = power(2, q1Marks);
 threeRaisedQ2 = power(3, q2Marks);
 … }
// POSTCONDITION:  …
```

```
// PRECONDITION:  …
int power(int base, int exponent)
{ … }
// POSTCONDITION:  …
```

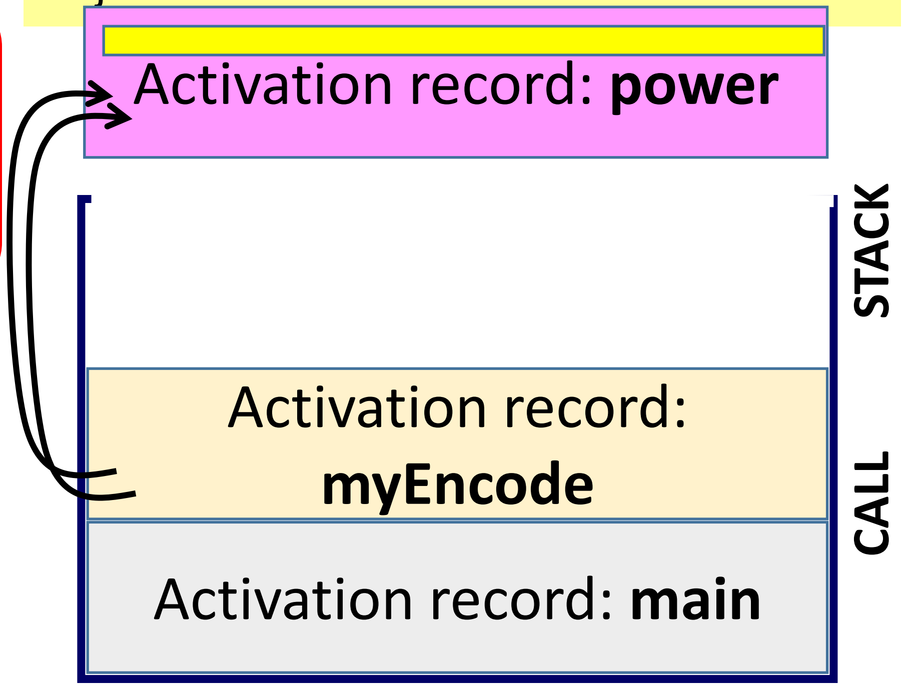
Recall: Activation Records in Call Stack

IIT Bombay

When a function (caller) calls a function (callee)

- a **fresh** activation record for callee created
- **Values of function parameters from caller copied to space allocated for formal parameters of callee**
- PC of caller saved
- Other book-keeping information updated
- Activation record for callee pushed on call stack

```
int
myEncode(int q1Marks, int q2Marks)
{ ….
  twoRaisedQ1 = power(2, q1Marks);
 …}
```



Activation record: **power**
STACK
CALL
Activation record: **myEncode**
Activation record: **main**

Call-by-Value Paradigm

IIT Bombay

Values of function parameters copied from activation record of caller to activation record of callee

Recall:

Formal parameters of callee (**power**) are its local variables

Not confused with parameters used in caller (**myEncode**) when invoking callee (**power**)

Only way in which callee (**power**) can let caller (**myEncode**) see effects of its computation is through return value of callee

Caveat When Using Call-by-Value

IIT Bombay

Any changes done on local variables of callee completely lost when callee returns to caller

Recall: Space for local variables of callee allocated in activation record of callee

Local variables of a function also called its **stack variables**

When callee returns to caller, activation record of callee freed up (lost forever!)

Program For Swapping Numbers

IIT Bombay

```
#include <iostream>
```

Values of m and n as local variables of **swap** (in **swap**'s activation record) are swapped

main doesn't get to see this swap

```
  status = swap(a, b);
  cout << "a: " << a << " b: " << b << endl;
  return 0;
}
```

```
int swap(int m, int n)

 t temp;
 mp = m;
  m = n;
  n = temp;
  return 0;
}
```

How Could We Fix This?

IIT Bombay

```
#include <iostream>
```

```
}
```

```
int swap(int &m, int &n)
{
  int temp;
  temp = m;
  m = n;
  n = temp;
  return 0;
}
```

Can we let the formal parameters refer to the variables used in **main** when calling **swap**?

Can caller and callee refer to the same variable?

Program For Swapping Numbers

IIT Bombay

```
#include <iostream>
using namespace std;
int swap(int &m, int &n);
int main() {

  return 0;
}
```

```
int swap(int &m, int &n)
{
  int temp;
  temp = m;
  m = n;
  n = temp;
  return 0;
}
```

m and n are NOT local variables of swap, but **references** (or **aliases**) to caller variables (a and b) used to pass parameters

Call-by-Reference Paradigm

IIT Bombay

```
#include <iostream>
using namespace std;
```

```
}
```

```
int swap(int &m, int &n)
```

```
}
```

Since m and n are not local variables of swap, **no space allocated for m and n in activation record of swap**

Can lead to significant savings in memory required for call stack

Caveat When Using Call-by-Reference

Cannot pass a constant as parameter of a function called by reference (otherwise constant could be "changed" by callee)

```
#include <iostream>
using namespace std;
int swap(int &m, int &n);
int main() { int status;
  cout << "Give an integer: ";  cin >> a;
  status = swap(2, a);
  cout << "a: " << a << endl;
  return 0;
}
```

```
int swap(int &m, int &n)
{
  int temp;
  temp = m;
  m = n;
  n = temp;
  return 0;
}
```

Call-by-Value vs Call-by-Reference

IIT Bombay

- Call-by-reference allows functions to share variables
- Need to be careful

  Inadvertent updates: **Formal parameters are not local variables, but shared with caller**

  Variables declared in body of function are local variables

- Can save significant memory for activation records on call stack for deeply nested function calls

Call-by-Value vs Call-by-Reference

- Call-by-value by far the safest

  No change to caller except through returned value
- Clean separation of variables of caller and callee
- Can lead to significant memory usage in activation records for deeply nested function calls
- Specific choice is context-dependent

Functions Not Returning Values

```
#include <iostream>
using namespace std;
                                        );
}
```

No scope for errors here!
swap also not used to compute an int result.

Can we let swap return nothing (no value)?

```
int swap(int &m, int &n)
{
  int temp;
  temp = m;
  m = n;
  n = temp;
  return 0;
}
```

Functions Not Returning Values

```
#include <iostream>
using namespace std;
void swap(int &m, int &n);
int main() {
  int status;
  cout << “Giv                    > a >> b;
  status = swa
  cout << “a: “ << a << “ b: “ << b << endl;
  return 0;
}
```

Return type “void”

```
void swap(int  &m, int &n)
{
  int temp;
  temp
  m = r
  n = tem
  return;
}
```

Simply “return” without argument

Summary

IIT Bombay

- Parameter passing in function calls
  - Call-by-value
  - Call-by-reference
- Caveats and benefits of each paradigm
- Functions without return values