CSCI 262

4 – Functions

Functions

- A block of code separated out, given a name
- Typically has a single purpose
  - E.g., calculate the square root of a number
- In C++:

  ```cpp
  int myfunc(string p1, double p2) {
    // code goes here
    return answer;
  }
  ```

- You call a function, passing in arguments

Overloading

You can make more than one function with the same name:
- Can have different return types
- **Must** have different parameter lists
  
  **E.g.**, 
  ```cpp
  void doit() { cout << "I did it!" << endl; }
  int doit(int x) { return x; }
  double doit(double x, double y) { return x/y; }
  ```

Overloading

Above is a poor example: why?

Generally, we want overloaded functions of the same name to serve similar purposes:

```cpp
string to_string(int x);
string to_string(double x);
...
```

Or maybe

```cpp
void print_list(Vector<int> list);
void print_list(int[] list, int size);
``` 

Default Parameters

Alternative when one overload is just a specialized version of another:

```cpp
// splits a string into substrings between occurrences
// of delim in the string. Default delimiter is space.
Vector<string> split(string s, char delim = ' ');
```

Can now call:

```cpp
split("Hello world");
```

Rules:

- Cannot omit earlier parameters, supply later ones
- Cannot overload if parameter list interpretable as call to function with default params omitted, e.g., cannot also define

```cpp
Vector<string> split(string s)
```

Parameter Types

What does this program print?

```cpp
void set_to_zero(int x) { x = 0; }

int main() {
  int n = 42;
  set_to_zero(n);
  cout << n << endl;
}
```

**Answer:** 42
Reference Parameters

```c++
void set_to_zero(int &x) { x = 0; }

int main() {
    int n = 42;
    set_to_zero(n);
    cout << n << endl;
}
```

This prints: 0

Reference Parameter

When to use:
• In place of return values (make sure intent is clear!)
• When parameter copy is too expensive (e.g., a large collection object)
• Generally, when passing an object type

When not to use (generally):
• When passing base types (int, double, etc.)

Memory Model of a C++ Program

- **Stack Segment**: The stack: local variables, function arguments, return values. Grows "down".
- **Heap**: The heap: dynamically allocated memory. Grows "up".
- **Data Segment**: Global and static variables, constants.
- **Text Segment**: Program code. Read only!

The Stack

- Holds "stack frames" aka "activation records"
- Each function call results in a new stack frame
- Each stack frame contains memory for:
  - Local variables declared in the function
  - Arguments passed into function
  - Return address for function
- When the function is exited, all of this memory is returned to the stack automatically.

Function Call Example

```c++
void quotient(double num, double den) {
    double q = num / den;
    cout << num << '/' << den << " is " << q << endl;
}

void print_quotients(int x, int y) {
    quotient(x, y);
    quotient(y, x);
}

int main() {
    int a, b;
    cout << "Please enter 2 non-zero integers: ";
    cin >> a >> b;
    print_quotients(a, b);
    return 0;
}
```

Example

At start of main()

```
main
int a = ?
int b = ?
```

Top of Stack

```
main()
int a = ?
int b = ?
```
After getting input:

```c
int main() {
    int a = 7;
    int b = 2;
    cout << "Please enter 2 non-zero integers: ";
    cin >> a >> b;
    print_quotients(a, b);
    return 0;
}
```

Stack

Top of Stack

At beginning of call to `print_quotients`:

```c
int a = 7
int b = 2
print_quotients
int x = 7
int y = 2
Top of Stack
```

At beginning of first call to `quotient`:

```c
main
int a = 7
int b = 2
print_quotients
int x = 7
int y = 2
quotient
double num = 7
double den = 2
double q = ?
Top of Stack
```

At end of call to `quotient`:

```c
main
int a = 7
int b = 2
print_quotients
int x = 7
int y = 2
quotient
double num = 7
double den = 2
double q = 3.5
Top of Stack
```

At beginning of second call to `quotient`:

```c
main
int a = 7
int b = 2
print_quotients
int x = 7
int y = 2
quotient
double num = 2
double den = 7
double q = ?
Top of Stack
```

After return from call to `quotient`:

```c
main
int a = 7
int b = 2
print_quotients
int x = 7
int y = 2
Top of Stack
```
Example

At end of second call to quotient:

> Please enter 2 non-zero integers: 7 2
> 7/2 is 3.5
> 2/7 is 0.285714

Example

After return from second call to quotient:

> Please enter 2 non-zero integers: 7 2
> 7/2 is 3.5
> 2/7 is 0.285714

Example

After call to print_quotients:

> Please enter 2 non-zero integers: 7 2
> 7/2 is 3.5
> 2/7 is 0.285714