6. Overloading: *Functions, Inlining, etc.*

**Function Overloading**

Notational convenience for functions that perform the same task on objects of different types.

```c
void print (int a);
void print (double b);
void print (char c);
void print (double a, double b);
```

Overloaded functions must have different argument signature, i.e., types and/or ordering of arguments.
Name resolution - Argument Matching

- Exact match, with no or unavoidable conversion
  - int to const int
- Matching using integral promotions
  - short to int
- Matching using standard conversions
  - int to double
- Matching using user-defined conversions

Let's match the calls to functions in the print example

```cpp
print(1); ________________________ print(1.2, 3.7); ________________________
print('a'); ________________________ print(1.2, 3); ________________________
print(1.2); ________________________ print(2, 3); ________________________
```

Constructors as Overloaded Functions

class Complex {
  double re;
  double im;
public:
  Complex (double rep, double imp);
  Complex (double rep); // imp is assumed 0
  Complex (const Complex& obj); // Copy constructor
};

Complex c1(5.2, 8.1); // ________________________
Complex c2(2.8); // ________________________
Complex c3(8); // ________________________
Complex c4(2.2, 4); // ________________________
Complex c5(c4); // ________________________
Matching: user-defined Conversion

```c
void print (int a);
void print (double b);
void print (char c);
void print (double a, double b);
void print (Complex a);
```

Let's match the calls to functions in the print example

```c
print(1); ____________________ print(1.2, 3.7);_____________________
print(‘a’);  ___________________ print(1.2, 3);   _____________________
print(1.2); ___________________ print(c3);        _____________________
```

Functions with Default Arguments

- Complex may be created with real and imaginary values, or just with real value and an assumed 0 for the imaginary value.

Using overloading

```c
Complex (double rep, double imp);
Complex (double rep);
```

Using default argument

```c
Complex (double rep, double imp = 0);
```

double power(double value, int pwr=2);
// Compute value^pwr, if pwr not given, compute value^2
power(3, 3);     // 27
power(3, 2);     // 9
power(3);        // 9
Rule for Default Arguments

- Default Arguments must be trailing

```c
void foo(int a = 0, int b, int c = 5); // Error
```

Example:

```c
void compute(int i = 1, int j = 2, int k = 3);
```

Effective values of arguments i, j, k in the following:

- `compute(3, 4, 5);` — 3, 4, 5
- `compute(3, 4);` — 3, 4, 3
- `compute(3);` — 3, 2, 3
- `compute();` — 1, 2, 3

Avoiding C-Style Macros

- Macros were used to help coding!

```c
#define MAX 1000
```

Problems with macros:

- Textual replacement
- Pre-processor handles it
- No precedence/Type checks.

```c
#define SQUARE (X) X * X
```

```c
SQUARE (2 + 3); // Surprise!!
```
Encapsulation & Performance: inlining functions

Function Call Overhead

mywatch.Reset(); // Calls a function to
// set seconds to 0

Functions may be “inline”-d to avoid function call overhead.
Expands the function where called.
Done by the compiler not pre-processor
Checks for types, and precedence rules are enforced.

Writing an Inline function

• If function is defined within the class, it is considered inline
• If function is defined outside the class, specify inline explicitly

class StopWatch {
    long getSeconds() { return seconds; }
    // Inline since defined within class.
    inline void Reset(); // Inline since specified.
    void start(); // Not inline
    ...
};
Inline Functions - Usage

• “inline” not an order - request honored if:
  • function is small (a few lines in size)
  • function is non-recursive

• The code for the function should be available to the compiler for expanding on call
  – This requires that the function be placed in the header file - in case of class member inline functions: not so desirable.

• Suggestion: worry about inlining functions when nearing release - only advantage is performance.

Ellipsis ...

Function with unspecified # of arguments
Provides flexibility to pass variable # of arguments

void sort(int number...); // Sort variable # of ints.

• Sacrifice one of the best features of C++ - strong type checking
• Implemented using va_list structure
• Don’t even think about it!!
• Only prudent use: catching arbitrary exceptions
Lab Work: Details provided on-line.