Review Function Overloading

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

float halveMe(float x){return x/2;}
int halveMe(int x){return x/2;}
int main()
{
    cout << halveMe(5.0f) << "," << halveMe(5) << halveMe(5.0) << endl;
    return 0;
}
```

Generic Functions

- **Generic functions** are functions with **type variables** whose **type values** are inferred by compilers based on the argument types of the functions call.

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

template<class T>
T halveMe(T x){return x/2;}

int main()
{
    cout << halveMe(5.0f) << "," << halveMe(5) << halveMe('0') << halveMe("10") << endl; // type error!!
    return 0;
}
```

Generic Functions - Continue

- Based on the argument types of the function call, **compiler infers the type of the type variable**:
  - `halveMe(5.0f): T is float`

- Compiler then substitutes the type variable with the inferred type to generate a **copy of the function** (similar to `cpp` expands macros):
  - `float halveMe(float x){return x/2;}`
Type Variable Instantiation

- A generic function is preceded by `template <class T, class S,...>(<typename T, typename S>...),` where `S,T` are dummy names.
- Each declared `type variable` in the template `(S, T)` must appear at least once in the function signature.
- Type variables need to be instantiated consistently:
  ```cpp
template <class T>
T max(T x, T y){return x < y ? y : x;}
- max(5.0, 10) //error
```

Generic Functions overhead

- Code bloat, because every instance of the generic function call leads to a copy of the function definition to be generated.
- Generic function declaration and definition have to be in the header file (like inline functions) so that compiler can do the type inference and function code generation.
- Longer compilation time.
- Larger executable.

C++ Class

- In addition to the built-in types (int, char, double, float, bool), C++ allows users to create their own `type` using `class`.
- A `class` is a grouping of data and functions into one unit, where the functions (member functions) operate on the data. (vs struct?)
- An `instance` of a `class` is called an `object`.

Point Class Declaration

```cpp
//Point.h
#ifndef __POINT_H
#define __POINT_H
class Point{
public:
    //constructors
    Point();
    Point(int, int);
    //inline functions
    int xv() const{return x;}
    int yv() const{return y;}
private:
    int x, y;
};
#endif
```

- Public Block:
  - Members in the public block can be accessed from both inside and outside of the class.
- Private Block:
  - Members in the private block can only be accessed from inside of the class.
Point Class Declaration -Continue

• Point() and Point(int, int):
  – overloaded constructors;
  – no return type (not even void).
  – have the same name as the class name.
• xv() and yv():
  – accessor functions.
  – functions which are implemented within the class declaration are inline functions.
  – They are declared as const member function, which are not allowed to change the internal state of the object.

Constructors

//Point.cc
#include "Point.h"
Point::Point(): x(0), y(0) {}
Point::Point(int x_in, int y_in)
{
  x = x_in;
  y = y_in;
}
• Constructors specifies how objects of a class is initialized.
• Default Constructor: constructor with no argument.
  – Using default arguments to initialize the given data members.

Objects Creation

#include "Point.h"
#include <iostream>
int main()
{
  Point p1;
  Point p2(10, 10);
  std::cout << p1.xv() <<"," << p1.yv() << std::endl;
  std::cout << p2.xv() <<"," << p2.yv() << std::endl;
}
• An instance of the point class is created by specifying the class name and the object name with constructor arguments:
  • Point p1; //default constructor
  • Point p2(10,10);
  • Access object public member function by dot () operator:
    – p1.xv()
    – p2.xv()

Built-in Types Objects

• Typename Objectname(initial value);
• Example:
  int i(6)
  //is equivalent to int i=6;
  std::cout << ++i;
  //?