More about Functions

Dr. Hakimzadeh
CS & Informatics, IUSB
What do we know about functions?

1. Forward declaration? (why)
2. Passing parameters (by value, by reference, by address)
3. Returning values (simple, struct, object)
4. Function with default parameters
5. Template Functions
6. Private vs. public functions
7. Inline functions
Why do we use forward declaration?

- To tell the compiler that the function being called will be declared at a later time. (prevents compiler errors!)
- To allow the compiler to check the method signature and warn us if there are type incompatibilities between the calling function and the called function.
- If the functions referenced in the program are not found during the link stage of compilation, then we get a link error.
Pass by value, reference, by address

**By value:**

declaration: void F1(int x) {}
call: f1(x);

**By reference:**

declaration: void F1(int & x) {}
call: f1(x);

**By address:**

declaration: void F1(int *x) {}
call: f1(&x);
Returning values from a function

- Void

- Simple data type (int, double, char, bool, etc.)

- User-defined data type (struct, class)
Default Parameters

- Default arguments can be specified for “call by value” parameters only.

- During the call, if the corresponding argument is omitted, it is replaced by the default value.

- Default values should be given either in the forward declaration or the function definition (whichever appears first. Not both, some compilers complain!)

- Default parameters must be from right to left. Right most parameter must be given a default value, then the one to its left, and so on. (no gaps!)

- Watch out for ambiguity!
Default Parameters

```cpp
void showVolume(int l, int h=2, int w=1)
{
    cout << l * h * w << endl;
}

showVolume(5);   // 5*2*1
showVolume(5,3);  // 5*3*1
showVolume(5,3,4); // 5*3*4
```
Default Parameters

```cpp
void showVolume(int l=3, int h, int w=2)
{
    cout << l * h * w << endl;
}

Problem! There is a gap!
```

Default parameters must be from right to left. Rightmost parameter must be given a default value, then the one to its left, and so on. (no gaps!)
Template Functions

- The book calls these “function templates”.

- Creating a “flexibly typed” function.

- Functions that do the same thing, except for different data types are good candidates for Template functions!
Template Functions

Examples:

Swap (char & var1, char &var2)
Swap (int & var1, int &var2)
Swap (double & var1, double &var2)

Sort (char arr[], int size)
Sort (int arr[], int size)
Sort (double arr[], int size)
Template Functions

**Syntax:**

```
Template <class T>
void swap (T & var1, T &var2)
{
    T temp;       // declare a temporary variable of type T
    temp = var1;
    var1 = var2;
    var2 = temp;
}
```

Template prefix for the function.

Specifies that this is a **template function**.

Also, "T" is the generic TYPE parameter.
Template Functions

Template <class T>
void swap (T & var1, T &var2)
{
    T temp; // declare a temporary variable of type T
    temp = var1;
    var1 = var2;
    var2 = temp;
}

Void main(){
    int x=1, y=2;
    swap(x, y);

    char a='p', b='h';
    swap(a, b);
}

To use a template function
Using Function Templates

// Template GenericSum Function will take
// Int, float, double, etc and return the sum
// of two parameters passed to it.

// Note: A template function is really not a function
// until it is instantiated. In other words,
// it is more like a class definition or struct
// definition. It is only a definition until is
// instantiated by the user. At that time, the
// proper TYPE is passed to the definition and a
// COPY of the template function is made and added to the
// code and compiled into the program.

template<typename TYPE>
TYPE GenericSum(TYPE a, TYPE b)
{
    cout << a + b << "endl;"
    return(a+b);
}

#include "TemplateFunctions.h"
#include <iostream>
using namespace std;

void main()
{
    int intResult = -1;
    float floatResult = -1.0;
    char charResult = 'A';

    cout << "---Make a template function for adding two integers" << endl;
    intResult = GenericSum<int>(5, 10);
    cout << "Sum: " << intResult << "endl;"

    cout << "---Make a template function for adding two floats" << endl;
    float floatResult = GenericSum<float>(5.7, 10.9);
    cout << "Sum: " << floatResult << "endl;"

    cout << "---Make a template function for adding two chars" << endl;
    char charResult = GenericSum<char>('1', '2'); // Ascii 49 + ascii 50
    cout << "Sum: " << charResult << "endl;"

    cout << "---Make a template function for adding two chars" << endl;
    char charResult = GenericSum<char>('1', '3'); // Note the type is implicitly determined
    cout << "Sum: " << charResult << "endl;"

    //cout << "---Make a template function for adding two chars" << endl;
    //char charResult = GenericSum<char>('1', 3); // Note we can't send two different types
    //cout << "Sum: " << charResult << "endl;"
}  

(c) Copyright 2007, H. Hakimzadeh 13
Template Functions

- More than one TYPE parameters

```cpp
Template <class T1, class T2>
void doStuff (T1 & var1, T2 & var2)
{

}

Void main()
{
    int x=1;
    char a='p';
    dostuff(x, a);

    double y=1;
    int b='p';
    dostuff(y, b);
}

(c) Copyright 2007, H. Hakimzadeh
Private / Public / Protected

- Public – Accessible outside of class.
- Private – Not accessible outside of class.
- Protected – Not accessible outside of class except for inherited classes.
Example of Protected Class Variables

class Parent
{
public:
    int P0;
    Parent() { P0 = 0;  P1 = 5; P2 = 10; }
    void print()
    { cout << "P1 = " << P1 << "\n" << "P2 = " << P2 << endl; }
private:
    int P1;
protected:
    int P2;
};

class Child: public Parent // inherit from Parent class
{
    int C1;
public:
    void initialize()
    {
        C1 = 1;
        // P1 = P1*P1; // Unable to access P1 because it is private
        P2 = P2*P2;  // No problem accessing P2 because it is protected
    }
    void print() { Parent::print(); cout << "C1 = " << C1 << endl; }
};

void main()
{
    Child myChild;
    myChild.print();
    // Call initialize to see if we can access
    // private vs. protected variables of
    // the parent class.
    myChild.initialize();
    myChild.print();
    // can still use the public variable
    // of the parent class
    myChild.P0 = 67;
}
Inline Functions

- We have used these already!

- Complete definition of the member function is provided within the class definition.

- Compiler treats inline functions differently. The code of the inline function is replicated everywhere it is called in the program. (this saves the overhead of function call)

- More efficient.