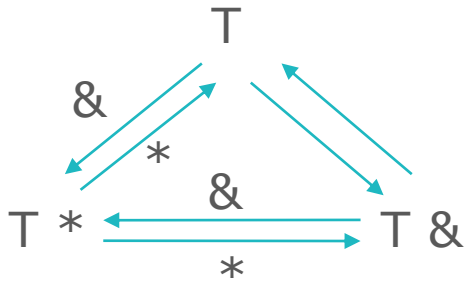
Lecture 5 - Function

Meng-Hsun Tsai
CSIE, NCKU



Solving Sudoku Recursively

```
 1  #include <vector>
 2  #include <iostream>
 3  #include <fstream>
 4  #include "Sudoku.h"
 5  using namespace std;
 6
 7  bool solve(Sudoku question,
                Sudoku & answer)
 8  {
 9      int firstZero;
10      firstZero = question.getFirstZeroIndex();
11      if(firstZero == -1)
12      {   // end condition
13          if(question.isCorrect())
14          {
15              answer = question;
16              return true;
17          }
18          else
19              return false;
20      }
20      }
21      else
22      {
23          for(int num=1; num<=9; ++num)
24          {
25              question.setElement(
                                  firstZero, num);
26              if(solve(question, answer))
27                  return true;
28          }
29          return false;
30      }
31  }
32  int main()
33  {
34      Sudoku ques;
35      Sudoku ans;
36      int num;
37
38      ifstream infile("su_question",ios::in);
```

Solving Sudoku Recursively (cont.)

```
39    for(int i=0;i<81;++i)  // read in question
40    {
41        infile >> num;
42        ques.setElement(i, num);
43    }
44
45    if(solve(ques, ans) == true)
46    {
47        cout << "Solvable!\n";
48        for(int i=0; i<81; i++)
49        {
50            cout << ans.getElement(i) << " ";
51            if(i%9==8)
52                cout << endl;
53        }
54    }
55    else
56        cout << " Unsolvable!!\n";
57
58 }
```

su_question (10 blanks)

```
8 0 5 3 2 0 4 1 7
2 0 3 1 7 5 8 6 9
1 9 7 6 8 4 5 0 3
3 1 9 0 5 8 6 7 4
4 2 6 0 9 1 3 5 8
5 7 8 4 3 0 1 9 2
7 5 4 9 1 3 2 0 6
6 8 2 5 4 0 9 3 1
9 3 1 8 6 2 7 0 5
```

```
> time ./sudoku_solve
Solvable!
8 6 5 3 2 9 4 1 7
2 4 3 1 7 5 8 6 9
1 9 7 6 8 4 5 2 3
3 1 9 2 5 8 6 7 4
4 2 6 7 9 1 3 5 8
5 7 8 4 3 6 1 9 2
7 5 4 9 1 3 2 8 6
6 8 2 5 4 7 9 3 1
9 3 1 8 6 2 7 4 5
632.517u 0.000s
10:32.53 99.9%
10+2758k 0+0io
```

10min!

```
1 2 3 0 5 6 7 8 9
1 2 3 4 5 6 7 8 0
1 0 3 4 5 6 7 8 9
1 2 3 4 5 0 7 8 9
1 2 3 4 5 6 7 0 9
1 2 0 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 0 7 8 9
0 2 3 4 0 6 7 0 9
```

16min!

```
>time ./sudoku_solve
Unsolvable!!
995.174u 0.007s
16:35.20 99.9%
10+2758k 0+0io
```

Sudoku.h and *Sudoku.cpp*

Sudoku.h

```
 1  #include <iostream>
 2  #include <vector>
 3  class Sudoku {
 4  public:
 5      Sudoku();
 6      Sudoku(const int init_map[]);
 7      void setMap(const int set_map[]);
 8      int getElement(int index);
 9      void setElement(int index,
                int value);
10      int getFirstZeroIndex();
11      bool isCorrect();
12      static const int sudokuSize = 81;
13
14  private:
15      bool checkUnity(int arr[]);
16      int map[sudokuSize];
17  };
```

Sudoku.cpp

```
        …
24  void Sudoku::setElement(int index, int value)
25  {
26      map[index] = value;
27  }
28  int Sudoku::getFirstZeroIndex()
29  {
30      for(int i=0;i<sudokuSize;++i)
31          if(map[i] == 0)
32              return i;
33      return -1;
34  }
        …
```

References and Reference Parameters

- Two ways to pass arguments to functions in many programming languages are pass-by-value and pass-by-reference.
- When an argument is passed by value, a *copy of the argument's value* *is made and passed (on the function call stack) to the called function.*
  - Changes to the copy do not affect the original variable's value in the caller.
  - One disadvantage is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.

```
7  bool solve(Sudoku question, Sudoku & answer)
```

Pass-by-value

References and Reference Parameters (cont.)

- With pass-by-reference, the caller gives the called function the ability to access the caller's data directly, and to modify that data.
- A reference parameter is an alias for its corresponding argument in a function call.
- To indicate that a function parameter is passed by reference, simply follow the parameter's type in the function prototype by an ampersand (&).
- Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.

```
7  bool solve(Sudoku question, Sudoku & answer)
```

Pass-by-reference

References and Reference Parameters (cont.)

- Pass-by-reference can weaken security; the called function can corrupt the caller's data.
- Because reference parameters are mentioned only by name in the body of the called function, you might inadvertently treat reference parameters as pass-by-value parameters. This can cause unexpected side effects if the original variables are changed by the function.
- To specify a reference to a constant, place the const qualifier before the type specifier in the parameter declaration.
- For passing large objects, use a constant reference parameter to simulate the appearance and security of pass-by-value and avoid the overhead of passing a copy of the large object.

```
int func(const Sudoku & su)
```

Error: Reference Without Initialization


```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int x = 3;
 7      int &y;
 8      cout << "x = " << x << "\t" << " y = " << y << endl;
 9      y = 7;
10      cout << "x = " << x << "\t" << " y = " << y << endl;
11  }
```

```
> g++ -o reference reference.cpp
reference.cpp: In function 'int main()':
reference.cpp:7: error: 'y' declared as reference but not initialized
```

Error: Reference Initialized as a Literal


```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int x = 3;
 7      int &y = 3;
 8      cout << "x = " << x << "\t" << " y = " << y << endl;
 9      y = 7;
10      cout << "x = " << x << "\t" << " y = " << y << endl;
11  }
```

```
> g++ -o reference2 reference2.cpp
reference2.cpp: In function 'int main()':
reference2.cpp:7: error: invalid initialization of non-const
reference of type 'int&' from a temporary of type 'int'
```

Reference Initialized as Another Variable


```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int x = 3;
 7      int &y = x;
 8      cout << "x = " << x << "\t" << " y = " << y << endl;
 9      y = 7;
10      cout << "x = " << x << "\t" << " y = " << y << endl;
11  }
```

```
> ./reference3
x = 3    y = 3
x = 7    y = 7
```

Initializing a Reference

- References can also be used as aliases for other variables within a function.

```
7       int &y = x;
```

- Reference variables must be initialized in their declarations and cannot be reassigned as aliases to other variables.

```
int &y = x;
y = z;
```

- Once a reference is declared as an alias for another variable, all operations supposedly performed on the alias are actually performed on the original variable.

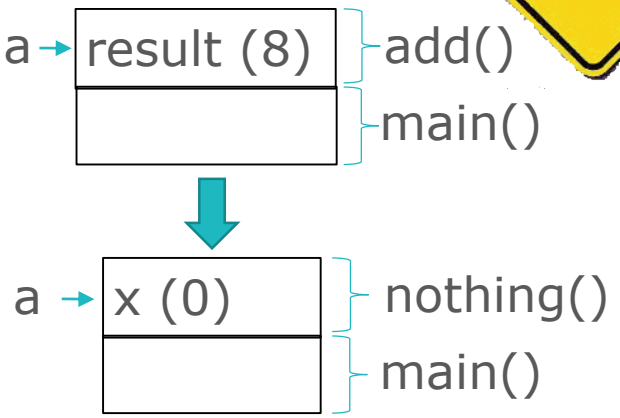
Returning a Reference

- Functions can return references, but this can be dangerous.
- Returning a reference to an automatic variable in a called function is a logic error. Some compilers issue a warning when this occurs.
- When returning a reference to a variable declared in the called function, the variable should be declared static in that function.

Error: Returning a Reference of a Local Variable

```
int & add ( int x, int y)
{
        int result;
        result = x + y;
        return result;
}
int nothing()
{
        int x = 0;
        return x;
}
```

```
int main()
{
    int & a = add(3, 5);
    cout << a << endl;
    nothing();
    cout << a << endl;
    return 0;
}
```

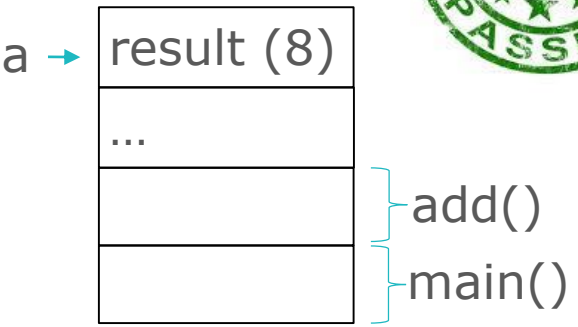


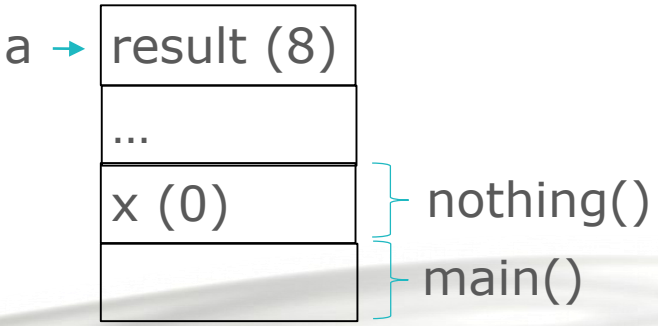

```
$ g++ -o reference4 reference4.cpp
reference4.cpp: In function `int& add(int, int)':
reference4.cpp:6: warning: reference to local variable `result' returned
$ ./reference4
8
0
```

Returning a Reference of a *Static* Local Variable

```
int & add ( int x, int y)
{
        static int result;
        result = x + y;
        return result;
}
int nothing()
{
        int x = 0;
        return x;
}
```

```
int main()
{
    int & a = add(3, 5);
    cout << a << endl;
    nothing();
    cout << a << endl;
    return 0;
}
```



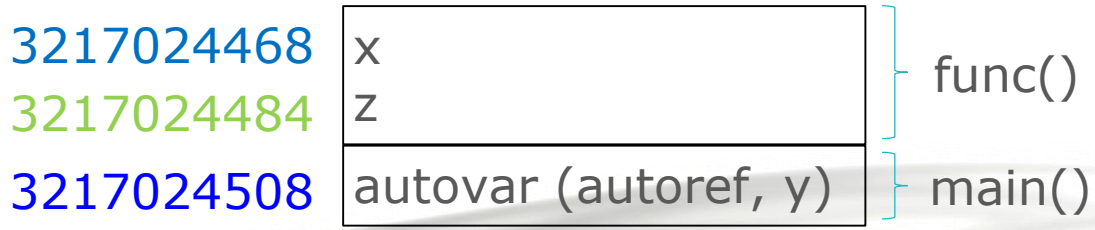



```
$ g++ -o reference5 reference5.cpp
$ ./reference5
8
8
```

Passing a Variable vs. Passing a Reference

```
void func(int & y, int z)
{
    int x;
    printf("In func():\n &x = %u\n", &x );
    printf("&y = %u\n", &y );
    printf("&z = %u\n", &z );
}
int main()
{
    int autovar;
    int & autoref = autovar;
    printf("In main():\n&autovar = %u\n", &autovar );
    printf("&autoref = %u\n", &autoref );
    func(autovar, autovar);
    func(autoref, autoref);
    return 0;
}
```

```
In main():
&autovar = 3217024508
&autoref = 3217024508
In func():
&x = 3217024468
&y = 3217024508
&z = 3217024484
In func():
&x = 3217024468
&y = 3217024508
&z = 3217024484
```



Default Arguments

- It is common for a program to invoke a function repeatedly with the same argument value for a particular parameter.
- In this case, the programmer can specify that such a parameter has a default argument, i.e., a default value to be passed to that parameter.
- When a program omits an argument for a parameter with a default argument in a function call, the compiler rewrites the function call and inserts the default value of that argument.
- Default arguments must be the rightmost (trailing) arguments in a function's parameter list.

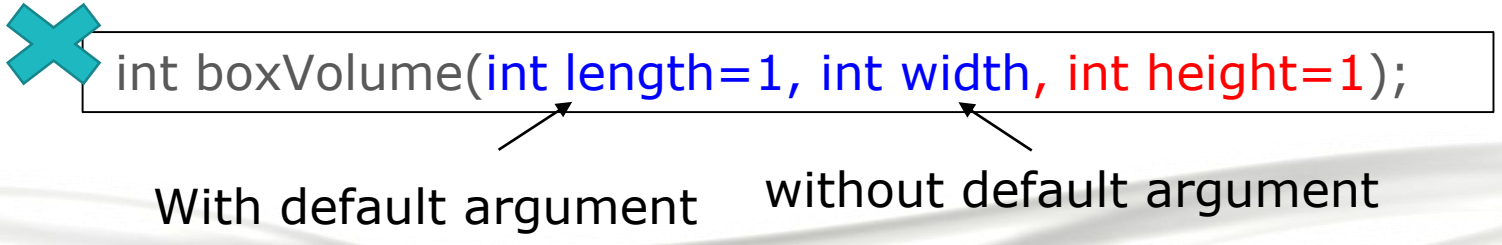
Using Default Arguments in *boxVolume()*

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int boxVolume(int length=1, int width=1, int height=1);
 5
 6  int main()
 7  {
 8      cout << boxVolume() << endl;
 9      cout << boxVolume(20) << endl;
10      cout << boxVolume(20, 10) << endl;
11      cout << boxVolume(20, 10, 3) << endl;
12      return 0;
13  }
14
15  int boxVolume(int length, int width, int height)
16  {
17      return length * width * height;
18  }
```

```
1
20
200
600
```

Notice on Using Default Arguments

- Using default arguments can simplify writing function calls. However, some programmers feel that explicitly specifying all arguments is clearer.
- If the default values for a function change, all client code using the function must be recompiled.
- Specifying and attempting to use a default argument that is not a rightmost argument (while not simultaneously defaulting all the rightmost arguments) is a syntax error.



Function Overloading

- C++ enables several functions of the same name to be defined, as long as they have different signatures.
- This is called function overloading.
- The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call.
- Function overloading is used to create several functions of the same name that perform similar tasks, but on different data types.
- Overloading functions that perform closely related tasks can make programs more readable and understandable.

Function Overloading (cont.)

- Overloaded functions are distinguished by their signatures.
- A signature is a combination of a function's name and number, types and order of its parameters.
- The compiler encodes each function identifier with the number and types of its parameters to enable type-safe linkage.
  - Ensures that the proper overloaded function is called and that the types of the arguments conform to the types of the parameters.

Print Start Time of *Timer* and *Clock* Objects

```
 1  #include <iostream>
 2  #include "Timer.h"
 3  #include "Clock.h"
 4
 5  void printStart(Timer & t)
 6  {
 7      cout << "Start time is ";
 8      cout << t.getStart();
 9      cout << " seconds since 1970/1/1
                  00:00:00. \n";
10  }
11  void printStart(Clock & c)
12  {
13      cout << "Start time is ";
14      cout << c.getStart();
15      cout << " virtual clocks since the
                   program executes. \n";
16  }
17
```

```
18  int main()
19  {
20      Timer tmr;
21      Clock clk;
22
23      tmr.start();
24      clk.start();
25
26      printStart(tmr);
27      printStart(clk);
28      return 0;
29  }
```

```
>  nm get_start
0000000000400ba0 T _Z10printStartR5Clock
0000000000400bf0 T _Z10printStartR5Timer
0000000000400c40 T main
```

```
Start time is 1391758338 seconds since 1970/1/1 00:00:00.
Start time is 3 virtual clocks since the program executes.
```

Encoding of Function Identifier

```
> cat overloading.cpp
double average(double n1, double n2)   {   return ((n1 + n2) / 2.0);  }
double average(double n1, double n2, double n3)
{
      return ((n1 + n2 + n3) / 3.0);
}
int average(int n1, int n2)  {    return ((n1 + n2) / 2);  }
int main()  {   return 0;  }
> g++ -o overloading overloading.cpp
> nm overloading
…
0000000000400600 T _Z7averagedd
0000000000400640 T _Z7averageddd
0000000000400680 T _Z7averageii
00000000004006a0 T main
```

From nm's man page:
nm - list symbols from object files
"T" The symbol is in the text (code) section.

Error: Take Return Type as Part of Signature

- Creating overloaded functions with identical parameter lists and different return types is a compilation error.
- A function with default arguments omitted might be called identically to another overloaded function; this is a compilation error.

```
1  double average(int n1, int n2)
2  {
3          return ((n1 + n2) / 2);
4  }
5  int average(int n1, int n2)
6 {
7          return ((n1 + n2) / 2);
8  }
9  int main()  {    return 0;   }
```

```
$ g++ -o overloading2 overloading2.cpp
overloading2.cpp: In function `int
average(int, int)':
overloading2.cpp:5: error: new
declaration `int average(int, int)'
overloading2.cpp:1: error: ambiguates
old declaration `double average(int, int)'
```

Error: Ambiguous Function Call to Overloaded Functions

```
1  #include <iostream>
2  using namespace std;
3  double add(int x, double y)  {   return x + y;   }
4  double add(double x, int y)  {   return x + y;   }
5  int main()
6  {
7          cout << add(1.3,1.5);
8          return 0;
9  }
```


```
$ g++ -o overloading3 overloading3.cpp
overloading3.cpp: In function `int main()':
overloading3.cpp:7: error: call of overloaded `add(double, double)' is ambiguous
overloading3.cpp:3: note: candidates are: double add(int, double)
overloading3.cpp:4: note:                 double add(double, int)
```

Function Templates

- If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently by using function templates.
- You write a single function template definition.

Return the Maximum Value

max.h

```
 1  template <typename T>
 2  T maximum(T v1, T v2, T v3)
 3  {
 4      T max = v1;
 5      if(v2 > max)
 6          max = v2;
 7      if(v3 > max)
 8          max = v3;
 9      return max;
10  }
```

```
maximum integer is 7
maximum double is 5.2
maximum char is C
```

max.cpp

```
 1  #include <iostream>
 2  #include "max.h"
 3  using namespace std;
 4
 5  int main()
 6  {
 7      int i1 = 7, i2 = 2, i3 = 3;
 8      double d1 = 2.1, d2 = 5.2, d3 = 3.3;
 9      char c1 = 'A', c2 = 'B', c3 = 'C';
10      cout << "maximum integer is " <<
11          maximum(i1, i2, i3) << endl;
12      cout << "maximum double is " <<
13          maximum(d1, d2, d3) << endl;
14      cout << "maximum char is " <<
15          maximum(c1, c2, c3) << endl;
16      return 0;
17  }
```

Syntax of Function Templates

- All function template definitions begin with the template keyword followed by a template parameter list to the function template enclosed in angle brackets (< and >).
- Every parameter in the template parameter list (often referred to as a formal type parameter) is preceded by keyword typename or keyword class.
- The formal type parameters are placeholders for fundamental types or user-defined types.
- These placeholders are used to specify the types of the function's parameters, to specify the function's return type and to declare variables within the body of the function definition.

```
1  template <typename T>
2  T maximum(T v1, T v2, T v3)
3  {
4     T max = v1;
```