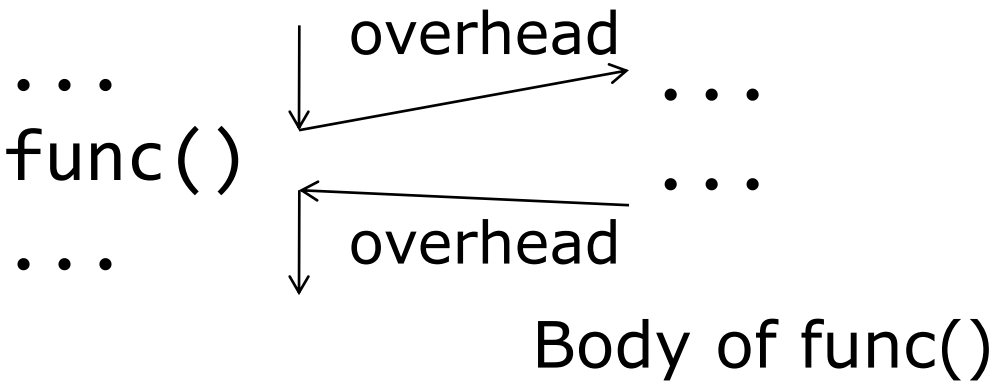
C++ Function Enhancements

Jiun-Long Huang
National Chiao Tung University

Normal Functions

- The calling of a normal function involves a series of instructions that are processed by the CPU.
- The run-time overhead of frequent function call will decrease the speed of the program.



Macro

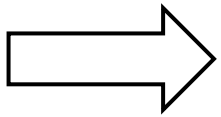
- C offers a tool called a macro to eliminate the function-call overhead.
- The #define preprocessor directive is used to create macros.
- Neither macros nor their parameters have a type.
  - No type checking
  - No type conversion

Macro (contd.)

- The preprocessor handles macros before the program is compiled.
- The process is essentially an expansion of the macro within the code.
  - It simply replaces the macro_identifier with its substitution text every time the identifier is used in the code.

Macro (contd.)

```
...
#define AREA(l,w) l*w
int main()
{
  cout<<AREA(4,3);
}
```



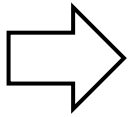
```
...
int main()
{
  cout<<4*3;
}
```

Source code re-writting

Macro (contd.)

- It is good programming practice to enclose all macro parameters in parentheses () to prevent errors.

```
...
#define AREA(l,w) l*w
#define AREA2(l,w) ((l)*(w))
int main()
{
  cout<<AREA(4+1,3+1);
  cout<<AREA2(4+1,3+1);
}
```



```
...
int main()
{
  cout<<4+1*3+1;
  cout<<((4+1)*(3+1));
}
```

Remarks on Macros

- Function-call overhead is eliminated
  - The speed of the program may increase
- The size of the program may be increased as well
  - A macro expansion causes duplicate code.
- Macros can also be a source of logic errors that are difficult to detect.

Inline Functions

- Compiler handles inline functions
  - The C++ compiler may ignore an inline request and compile the function as a normal non-inline function
- Good candidates for inline functions are small, frequently called functions.

```
inline return_type function_name (parameter list)
{
   //body of the function
}
```

```
#include <iostream>
using namespace std;
inline int AREA(int l, int w)
{
  return l*w;
}
int main()
{
  cout<<AREA(4+1,3+1); // output 20
  return 0;
}
```

Remarks on Inline Functions

- Programmers can debug them interactively in order to detect logic or run-time errors.
- Expanding inline functions is less prone to errors than expanding macros.
- Programmers do not require extra parentheses to ensure proper inline expansion.

Remarks on Inline Functions (contd.)

- When handling inline functions, the compiler also performs all necessary data type conversions.
  - With type checking
  - With type conversion
- Use inline functions rather than parameterized macros to reduce the function-call overhead.

Remarks on Inline Functions (contd.)

- Inline function can increase the speed of the program.
- Inserting multiple copies of a function's code into the program can make the program larger.

Remarks on Inline Functions (contd.)

- The following functions are usually not expanded inline:
  - Large functions
  - Functions containing loops
  - Functions containing switch or goto statements
  - Recursive functions
  - Functions containing static variables

Default Arguments

- A C++ function can have a variable number of arguments when called during run-time.
- Default arguments can be used with both inline and non-inline functions.

```
void fun1(char x, int y, float z);

void fun1(char x='$', int y=1, float z=3.14);
```

```
void fun1(char x='$', int y=1, float z=3.14);

fun1();          //same as: fun1('$', 1, 3.14);
fun1('+');       //same as: fun1('+', 1, 3.14);
fun1('%', 9);    //same as: fun1('%', 9, 3.14);

//same as: fun1('%', 9, 6.28);
fun1('%', 9, 6.28);

//Caution! same as: fun1(3, 2.1, 3.14);
//x=(char)3, y=2, z=3.14
fun1(3,2.1);
```

Default Arguments (contd.)

- If a function has both default and non-default arguments, the non-default arguments must be listed first in the function's parameter list.
- The default arguments must always be to the right of those specified as non-default arguments in the parameter list.

```
void fun1(char x, int y=1, float z=3.14);

void fun1(char x='$', int y, float z=3.14); X
```

Function Overloading

- C++ enables the same name to be used for two or more different functions.
  - This concept is called function overloading.
- Requirements:
  - Have a different number of parameters,
    - int fun1(int);
    - int fun1(int, float);
    - int fun1(int, float, int);

Function Overloading (contd.)

- Have the same number of parameters, then:
  - parameter types should be in a different order,
    - void fun2(char, int, float);
    - void fun2(int, char, float);
  - At least one parameter type should be different,
    - void fun3(int, char);
    - void fun3(double, char);

Function Overloading (contd.)

- In a function call, the C++ compiler checks the function's name as well as the function's arguments on each call to determine which function to use.
  - C compilers only checks function's name
- Incorrect examples:
  - int fun(double, int);
  - double fun(double, int);
  - Type of return value cannot be used to distinguish functions

```
#include <iostream>
#include <iomanip>
using namespace std;
const int size = 10;
void sort_int(int arr[]);
void sort_float(float arr[]);
int main()
{
  int nums1[size]={3,9,1,-5,0,1,-3,4,6,7};
  float nums2[size]={9.1,-0.7,4.6,0.3,9.9,
                     1.1,3.2,-1.2,6.7,-4.9};
  sort_int(nums1);
  sort_float(nums2);
  cout<<" Sorted arrays:"<<endl;
  for(int j=0; j<size; j++)
    cout<<setw(5)<<nums1[j]<<setw(8)<<nums2[j]<<endl;
  return 0;
}
```

```
void sort_int(int arr[])
{
  int temp;
  for(int j =1; j<size; j++)
    for(int k =0; k<size-1; k++)
      if(arr[k]>arr[k+1]) {
        temp = arr[k];
        arr[k] = arr[k+1];
        arr[k+1] = temp;
      }
}
void sort_float(float arr[])
{
  float temp;
  for(int j =1; j<size; j++)
    for(int k =0; k<size-1; k++)
      if(arr[k]>arr[k+1]) {
        temp = arr[k];
        arr[k] = arr[k+1];
        arr[k+1] = temp;
      }
}
```

```
#include <iostream>
#include <iomanip>
using namespace std;
const int size = 10;
void sort(int arr[]);          //overloaded function
void sort(float arr[]);
int main()
{
  int nums1[size]={3,9,1,-5,0,1,-3,4,6,7};
  float nums2[size]={9.1,-0.7,4.6,0.3,9.9,
                     1.1,3.2,-1.2,6.7,-4.9};
  sort(nums1);            //Calls overloaded function
  sort(nums2);
  cout<<" Sorted arrays:"<<endl;
  for(int j=0; j<size; j++)
    cout<<setw(5)<<nums1[j]<<setw(8)<<nums2[j]<<endl;
  return 0;
}
```

```
void sort(int arr[])
{
  int temp;
  for(int j =1; j<size; j++)
    for(int k =0; k<size-1; k++)
      if(arr[k]>arr[k+1]) {
        temp = arr[k];
        arr[k] = arr[k+1];
        arr[k+1] = temp;
      }
}
void sort(float arr[])
{
  float temp;
  for(int j =1; j<size; j++)
    for(int k =0; k<size-1; k++)
      if(arr[k]>arr[k+1]) {
        temp = arr[k];
        arr[k] = arr[k+1];
        arr[k+1] = temp;
      }
}
```

The algorithms are the same
→Code duplication
→Template can solve this problem

Ambiguity

- If the compiler is confused when deciding which version of the overloaded function to use, it produces an error message.
- The most common sources of ambiguity are
  - Automatic type conversions
  - Using default arguments
  - Using references as function parameters
  - Calling overloaded functions

```
void fun1(float);
void fun1(double);
void fun2(int=1);
void fun2();
//Assume these declarations
int x=3;
float y=4.4;
double z=5.5555;
//Calling overloaded functions:
fun1(y);   //Valid call, funl(float) is called
fun1(z);   //Valid call, funl(double) is called
fun1(x);   //Ambiguity: x can be
           //converted to float or double
fun2(x);   //Valid call, fun2(int) is called
fun2();    //Ambiguity: fun2(int=1) and fun2()
           //can be used
```

Remarks on Function Overloading

- It may reduce the complexity of program, particularly of large programs.
- In C++, only one function name needs to be used, leaving the compiler to select the correct version of the function.
- It is NOT good programming practice to use the same name for functions that perform logically different operations.