IM 1003: Programming Design

Functions (II)

Ling-Chieh Kung

Department of Information Management
National Taiwan University

March 17, 2014

Outline

- **More about functions**
- Self-defined libraries
- Randomization

Call-by-value mechanism (1/4)

- Consider the example program.
- Is the result strange?

```
void swap (int x, int y);
int main()
{
  int a = 10, b = 20;
  cout << a << " " << b << endl;
  swap(a, b);
  cout << a << " " << b << endl;
}
void swap (int x, int y)
{
  int temp = x;
  x = y;
  y = temp;
}
```

Call-by-value mechanism (2/4)

- The default way of invoking a function is the "**call-by-value**" (pass-by-value) mechanism.
- When the function **`swap()`** is invoked:
  - First two **new** variables **`x`** and **`y`** are created.
  - The values of **`a`** and **`b`** are **copied** into **`x`** and **`y`**.
  - The values of **`x`** and **`y`** are swapped.
  - The function ends, **`x`** and **`y`** are **destroyed**, and memory spaces are released.
  - The execution goes back to the main function. Nothing really happened…

Address	Identifier	Value
-	a	10
-	b	20

Memory

Call-by-value mechanism (3/4)

- The call-by-value mechanism is adopted so that:
  - Functions can be written as **independent entities**.
  - Modifying parameter values do **not** affect any other functions.
- **Work division** becomes easier and program **modularity** can also be enhanced.
  - Otherwise one cannot predict how her program will run without knowing how her teammates implement some functions.
- In some situations, however, we do need a callee to modify the values of some variables defined in the caller.
  - We may "**call by reference**" (to be introduced in the next week).
  - Or we may pass an **array** to a function.

Call-by-value mechanism (4/4)

- When an array parameter is modified in a function, the caller also see it modified!
- Why?
- Passing an array is **passing an address**.
  - The callee modifies whatever contained in those addresses.

```
void shiftArray (int [], int);
int main()
{
  int num[5] = {1, 2, 3, 4, 5};
  shiftArray(num, 5);
  for (int i = 0; i < 5; i++)
    cout << num[i] << " ";
  return 0;
}
void shiftArray (int a[], int len)
{
  int temp = a[0];
  for (int i = 0; i < len - 1; i++)
    a[i] = a[i + 1];
  a[len - 1] = temp;
}
```

Constant parameters (1/3)

- In many cases, we do not want a parameter to be modified inside a function.
- For example, consider the factorial function:

```
int factorial (int n)
{
  int ans = 1;
  for (int a = 1; a <= n; a++)
    ans *= a;
  return ans;
}
```

- For no reason should the parameter **n** be modified. You know this, but how to prevent other programmer from doing so?

Constant parameters (2/3)

- We may declare a parameter as a **constant parameter**:

```
int factorial (const int n)
{
  int ans = 1;
  for (int a = 1; a <= n; a++)
    ans *= a;
  return ans;
}
```

- Once we do so, if we assign any value to **n**, there will be a compilation error.
- The argument passed into a constant parameter can be a non-constant variable.

Constant parameters (3/3)

- For arguments whose values **may be** but **should not be** modified in a function, it is good to protect them.
  - E.g., arrays.

```
void printArray (const int [5], int);
int main()
{
  int num[5] = {1, 2, 3, 4, 5};
  printArray(num, 5);
  return 0;
}
void printArray (const int a[5], int len)
{
  for (int i = 0; i < len; i++)
    cout << a[i] << " ";
  cout << endl;
}
```

Function overloading (1/4)

- There is a function calculating x^y:
  - `int pow (int base, int exp);`
- Suppose we want to calculate x^y where y may be fractional:
  - `double powExpDouble (int base, double exp);`
- What if we want more?
  - `double powBaseDouble (double base, int exp);`
  - `double powBothDouble (double base, double exp);`
- We may need a lot of **`powXXX()`** functions, each for a different parameter set.

Function overloading (2/4)

- To make programming easier, C++ provides **function overloading**.
- We can define many functions having **the same name** if their parameters are not the same.
- So we do not need to memorize a lot of function names.
  - `int pow (int, int);`
  - `double pow (int, double);`
  - `double pow (double, int);`
  - `double pow (double, double);`
- Almost all functions in the C++ standard library are overloaded, so we can use them conveniently.

Function overloading (3/4)

- Different functions must have different **function signatures**.
  - This allows the computer to know which function to call.
- A function signature includes
  - Function name.
  - Function parameters (**number** of parameters and their **types**).
- A function signature does not include return type! Why?
- When we define two functions with the same name, we say that they are **overloaded** functions. They **must** have different parameters:
  - Numbers of parameters are different.
  - Or at least one pair of corresponding parameters have different types.

Function overloading (4/4)

- Here are two functions:
  - `void print(char c, int num);`
  - `void print(char c);`
- `print()` can print `c` for `num` times. If no `num` is assigned, print a single `c`.

```
void print (char c, int num)
{
  for (int i = 0; i < num; i++)
    cout << c;
}
```

```
void print (char c)
{
  cout << c;
}
```

Default arguments (1/2)

- In the previous example, it is identical to give **num** a **default value 1**.
- In general, we may assign default values for some parameters in a function.
- As an example, consider the following function that calculates a circle area:

```
double circleArea (double, double = 3.14);
// ...
double circleArea (double radius, double pi)
{
  return radius * radius * pi;
}
```

- When we call it, we may use **circleArea(5.5, 3.1416)**, which will assign 3.1416 to **pi**, or **circleArea(5.5)**, which uses 3.14 as **pi**.

Default arguments (2/2)

- Default arguments must be assigned **before** the function is called.
  - In a function declaration or a function definition.
- Default arguments must be assigned **just once**.
- You can have as many parameters using default values as you want.
- However, parameters with default values must be put **behind** (to the **right** of) those without a default value.
  - Once we use the default value of one argument, we need to use the default values for **all** the **following** arguments.
- How to choose between function overloading and default arguments?

Inline functions (1/2)

- When we call a function, the **system** needs to do a lot of works.
  - Allocating memory spaces for parameters.
  - Copying and passing values as arguments.
  - Record where we are in the caller.
  - Pass the program execution to the callee.
  - After the function ends, destroy all the parameters and get back to the calling function.
- When there are a lot of function invocations, the program will take a lot of time doing the above stuffs. It then becomes **slow**.
- How to save some time?

Inline functions (2/2)

- In C++ (and some other modern languages), we may define **inline functions**.
- To do so, simply put the keyword `inline` in front of the function name in a function prototype or header.
- When the compiler finds an inline function, it will **replace** the invocation by the function statements.
  - The function thus does not exist!
  - Statements will be put in the caller and executed directly.
- While this saves some time, it also expands the program size.
- In most cases, programmers do not use inline functions.

Outline

- More about functions
- **Self-defined libraries**
- Randomization

Libraries

- There are many C++ standard **libraries**.
  - **<iostream>**, **<climits>**, **<cmath>**, **<cctype>**, **<cstring>**, etc.
  - Many (constant) variables and functions are defined there.
  - Many more.
- We may also want to define **our own libraries**.
  - Especially when we collaborate with teammates.
  - Typically, one implements a function for the others to call.
  - That function can be defined in a self-defined library.
- A library includes a **header file** (.h) and a **source file** (.cpp).
  - The header file contains declarations; the source file contains definitions.

Example

- Consider the following program with a single function **myMax()**:

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

int myMax (int [], int);
int main ()
{
  int a[5] = {7, 2, 5, 8, 9};
  cout << myMax (a, 5);
  return 0;
}
```

```
int myMax (int a[], int len)
{
  int max = a[0];
  for (int i = 1; i < len; i++)
  {
    if (a[i] > max)
      max = a[i];
  }
  return max;
}
```

- Let's define a constant **variable** for the array length in **an header file**.

Defining variables in a library

myMax.h

```
const int LEN = 5;
```

main.cpp

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

int myMax (int [], int);
int main ()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  cout << myMax (a, LEN);
  return 0;
}
```

```
int myMax (int a[], int len)
{
  int max = a[0];
  for (int i = 1; i < len; i++)
  {
    if (a[i] > max)
      max = a[i];
  }
  return max;
}
```

Including a header file

- When your main program wants to include a self-defined header file, simply indicate its path and file name.
  - `#include "myMax.h"`
  - `#include "D:/test/myMax.h"`
  - `#include "lib/myMax.h"`
  - Using `\` or `/` does not matter (on Windows).
- We still compile the main program as usual.
- Let's also define **functions** in our library!
  - Now we need a source file.

Defining functions in a library

myMax.h

```
const int LEN = 5;
int myMax (int [], int);
```

main.cpp

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

int main ()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  cout << myMax (a, LEN);
  return 0;
}
```

myMax.cpp

```
int myMax (int a[], int len)
{
  int max = a[0];
  for (int i = 1; i < len; i++)
  {
    if (a[i] > max)
      max = a[i];
  }
  return max;
}
```

Including a header and a source file

- When your main program also wants to include a self-defined source file, the include statement needs not be changed.
  - `#include "myMax.h"`
- We add a source file myMax.cpp.
  - In the source file, we **implement** those functions declared in the header file.
  - The main file names of the header and source files can be different.
- The two source files (main.cpp and myMax.cpp) must be **compiled together**.
  - Each environment has its own way.
  - In Dev-C++, we simply create a "console project".

Defining one more function

myMax.h

```
const int LEN = 5;
int myMax (int [], int);
void print (int);
```

main.cpp

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

int main ()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  print (myMax (a, LEN));
  return 0;
}
```

myMax.cpp

```
int myMax (int a[], int len)
{
  int max = a[0];
  for (int i = 1; i < len; i++)
  {
    if (a[i] > max)
      max = a[i];
  }
  return max;
}
void print (int i)
{
  cout << i; // error!
}
```

Defining one more function

- Each source file contains statements to run.
- Each source file must include the libraries it needs for its statements.

```
#include <iostream>
using namespace std;
int myMax (int a[], int len)
{
  int max = a[0];
  for (int i = 1; i < len; i++)
  {
    if (a[i] > max)
      max = a[i];
  }
  return max;
}
void print (int i)
{
  cout << i; // good!
}
```

The complete set of files

myMax.h

```
const int LEN = 5;
int myMax (int [], int);
void print (int);
```

main.cpp

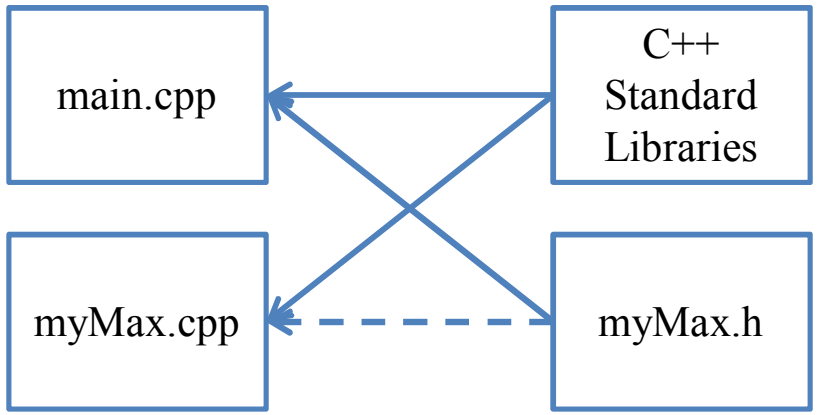
```
#include <iostream>
#include "myMax.h"
using namespace std;

int main ()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  print (myMax (a, LEN));
  return 0;
}
```

myMax.cpp

```
#include <iostream>
using namespace std;
int myMax (int a[], int len)
{
  int max = a[0];
  for (int i = 1; i < len; i++)
  {
    if (a[i] > max)
      max = a[i];
  }
  return max;
}
void print (int i)
{
  cout << i;
}
```

Remarks

- In many cases, myMax.cpp also include myMax.h.
  - E.g., if **LEN** is accessed in myMax.cpp.
- More will be discussed when we introduces classes.
  - More than two source files.
  - A header file including another header file.



Outline

- More about functions
- Self-defined libraries
- **Randomization**

Random numbers

- In some situations, we need to generate **random numbers**.
  - For example, a teacher may want to write a program to randomly draw one student to answer a question.
- In C++, randomization can be done with two functions, **`srand()`** and **`rand()`**.
- They are defined in **`<cstdlib>`**.

rand()

- `int rand();`
- It **"randomly"** returns an **integer** between 0 and **`RAND_MAX`** (in **`<cstdlib>`**, typically 32767).
- Try to run it for multiple times.
  - What happened?

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
  int rn = 0;
  for (int i = 0; i < 10; i++)
  {
    rn = rand();
    cout << rn << " ";
  }

  return 0;
}
```

`rand()`

- **`rand()`** returns a "**pseudo-random**" integer.
  - They just look **like** random numbers. But they are not really random.
  - There is a formula to produce each number.
  - e.g., $r_i = (a * r_{i-1} + b) \bmod c$.
- You need to have a "random number **seed**".
  - r_0 for this example.

srand()

- `void srand (unsigned int);`
  - A **seed** can be generated based on the input number.
- The sequence is now different.
- Try to run it for multiple times.
  - What happened?

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
  srand(0);
  int rn = 0;
  for (int i = 0; i < 10; i++)
  {
    rn = rand();
    cout << rn << " ";
  }

  return 0;
}
```

`srand()`

- We must give **`srand()`** **different arguments**.
- In many cases, we use **`time(0)`** to be the argument of **`srand()`**.
  - The function **`time(0)`**, defined in **`<ctime>`**, returns the number of seconds that have past since 0:0:0, Jan, 1st, 1970.
  - The argument **`0`** is hard to be explained now.

srand() and time()

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
  srand(time(0)); // good
  int rn = 0;
  for (int i = 0; i < 10; i++)
  {
    rn = rand();
    cout << rn << " ";
  }
  return 0;
}
```

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
  int rn = 0;
  for (int i = 0; i < 10; i++)
  {
    srand(time(0)); // bad
    rn = rand();
    cout << rn << " ";
  }
  return 0;
}
```

Random numbers in a range

- If you want to produce random numbers in a specific range, use `%`.
- What is the range in this program?
- How about this?

```
rn = (static_cast<double>(rand() % 501)) / 100;
```

- More powerful random number generators are provided in **<random>** (if your compiler is new enough).

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
  srand(time(0));
  int rn = 0;
  for (int i = 0; i < 10; i++)
  {
    rn = ((rand() % 10)) + 100;
    cout << rn << " ";
  }
  return 0;
}
```