IM 1003: Computer Programming
Functions and Randomization

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

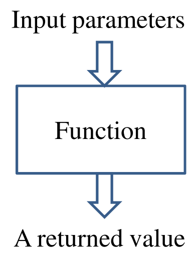
- **Basics of functions**
- More about functions
  - Function overloading
  - Default function arguments
  - Inline functions
- Variable lifetime
- Randomization

Why functions?

- In C++ and most modern programming languages, we may put statements into **functions** that are to be called in the future.
  - Also known as **procedures** in some languages.
- Why functions?
- We need **modules** instead of a huge main function.
  - Easier to divide the works.
  - Easier to debug.
  - Easier to maintain consistency.
- We need something that can be used **repeatedly**.
  - Enhance reusability.

Structure of functions

- In C++, a function is composed of a **header** and a **body**.
- A Header for **declaration**:
  - A function name.
  - A list of input parameters.
  - A return value.
- A body for **definition**:
  - Statements that define the task.



Function declaration

- To implement a function, we first declare its **prototype**:

```
return type function name (parameter types);
```

- Some examples:
  - `int add(int num1, int num2);`
  - `int add(int, int);`
  - `double divide(double, double);`
  - `double divide(double numerator, double denominator);`

Function declaration

- In a function prototype, we declare its appearance and behavior.
- A function name:
  - The name of the function.
  - The naming rule is the same as variable naming.
- A list of parameters:
  - The parameters passed into the function with their types.
  - We must declare their **types**. Declaring their names are optional.
  - There can be any number of parameters. It can also be zero.
- A return value:
  - The **type** of the function return value.
  - There can only be one return value.

Function declaration

- `int add(int num1, int num2);`
  - A function receives two integers and returns an integer.
  - The parameter names may provide "hints" to what this function does.
- `double divide(double, double);`
  - A function receives two double-precision real numbers and returns a double-precision real number.
- For a function declaration, the **semicolon** is required.
- The return type:
  - Every type can be the return type.
  - Or it may be "`void`": return nothing.

Using a function

- Declare the function before using it.
  - Typically after the preprocessors and **before** the main function.
- Then we need to **define** the function by writing the function **body**.
  - Typically **after** the main function, though not required.
- Recall that in a function prototype, we do not need to specify parameter **names**.
- But in a function definition, we need!
  - Otherwise, we will not know how to use them in the function.
- These parameters can be viewed as **variables** declared **inside** the function. They can be accessed only in the function.

Function definition

- You have written one function: the **main** function.
- Defining other functions can be done in the same way.

```
return type function name (parameters)
{
  statements
}
```

  - The first line, the function header, is almost identical to the prototype.
  - However, the parameter **names** must be specified.
- Let's see one example:

Function definition

- The **add()** function:

```
int add (int num1, int num2)
{
  return num1 + num2;
}
```

- Then in the main function we may call the **add()** function:

```
int main ()
{
  int c = add(10, 20);
  cout << c << endl;
  return 0;
}
```

Function invocation

- When a function is invoked in the main function, the program execution **jumps** to the function.
- After the function execution is complete, the program execution jumps **back** to the main function, exactly where the function is called.
- What if another function is called in a function?

```
int add (int, int);

int main ()
{
  int c = add(10, 20);
  cout << c << endl;
  return 0;
}

int add (int num1, int num2)
{
  return num1 + num2;
}
```

Function definition

- You may define a function before the main function.
- In this case, the function prototype can be omitted.

```
int add (int, int);

int main ()
{
  int c = add(10, 20);
  cout << c << endl;
  return 0;
}

int add (int num1, int num2)
{
  return num1 + num2;
}
```

```
int add (int num1, int num2)
{
  return num1 + num2;
}

int main ()
{
  int c = add(10, 20);
  cout << c << endl;
  return 0;
}
```

Function parameters vs. arguments

- When we invoke a function, we need to provide **arguments**.
  - **Parameters**: variables used inside the function.
  - **Arguments**: values passed into the function.
- If an argument's type is different from the corresponding parameter's type, compiler will try to **cast** it.

```
int add (int num1, int num2)
{
  return num1 + num2;
}
int main ()
{
  double c = add(10.5, 20.7); // !
  cout << c << endl;
  return 0;
}
```

```
double add (double num1, double num2)
{
  return num1 + num2;
}
int main ()
{
  int c = add(10, 20); // OK~
  cout << c << endl;
  return 0;
}
```

Function arguments

- Function arguments can be:
  - Literals.
  - Variables.
  - Constant variables.
  - Expressions.
- An exception is that arguments for a call-by-reference parameter can only be variables.
  - This will be discussed later.

```
int main ()
{
  const int C = 5;
  int d = 1;
  cout << add(10, 20);
  cout << add(C, d);
  cout << add(10 + C, 20);
  return 0;
}

int add (int num1, int num2)
{
  return num1 + num2;
}
```

Function return value

- We can return **one or no** value back to the place we invoke the function.
- Use the **`return`** statement to return a value.
- If you do not want to return anything, declare the function return type as **`void`**.
  - In this case, the **`return`** statement can be omitted.
  - Otherwise, having no **`return`** statement results in a compilation error.

Function return value

- There can be as many **`return`** statements as you wish.
- A function runs until the **first** **`return`** statement is encountered.
  - Or the end of the function for a function returning **`void`**.

```
int max (int a, int b)
{
  if(a > b)
    return a; // first return
  else
    return b; // second return
}
```

- We need to ensure that at least one return will be executed!
  - Example "**06_01_return**".

Example

- How to write a function that returns $n!$ (the factorial of n)?

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

- How to write a function that prints out $n!$ (the factorial of n)?

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

Function invocation

- For a function that has no return value, invocation must be

```
function name (parameters);
```

- If a function has a return value, we may use either

```
variable = function name (parameters);
```

or

```
function name (parameters);
```

  – In the latter case, the return value will be dropped.

An example

```
int add (int, int);
void print (int);

int main()
{
  int a = 10, b = 20;
  int c = add(a, b); // c becomes 30
  print(c); // c will be printed out
  add(a, c); // nothing will happen
  int d = print(c); // compilation error
  return 0;
}
```

```
int add(int num1, int num2)
{
  return num1 + num2;
}

void print (int toPrint)
{
  cout << toPrint;
}
```

Good programming style

- Name a function so that its purpose is clear.
- In a function, name a parameter so that its purpose is clear.
- Declare all functions with suitable comments.
  – Ideally, other programmers can understand what a function does without reading the definition.
- Declare all functions at the beginning of the program.
  – A function **must** be declared or defined **before** it can be invoked.
  – Declaring all functions at the beginning removes the possibility of invoking a function that has not be declared or defined.

Outline

- Basics of functions
- **More about functions**
  - Call-by-value mechanism
  - Function overloading
  - Default function arguments
  - Inline functions
- Variable lifetime
- Randomization

Call-by-value mechanism

- Consider example "**06_02_swap**".
- 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

- The default way of invoking a function with parameters is the "**call-by-value**" mechanism.
- When the function `swap()` is invoked:
  - First two **new** variables `x` and `y` are created. Memory spaces are allocated.
  - The values contained in `a` and `b` are **copied and assigned** to `x` and `y`.
  - The function starts and 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…

Why call-by-value?

- The call-by-value mechanism is adopted so that:
  - Functions can be written as **independent entities** that can use any variable or parameter names.
  - Modifying parameter values will **not** affect any other functions.
- These advantages makes work division easier.
- Program modularity can also be enhanced.
- In some situations, however, we do need a called function to modify the values of some variables defined in the calling function.
  - This can be done with the "**call-by-reference**" mechanism, which will be discussed later.
  - This may also happen when we pass an **array** to a function.

Constant parameters

- In many cases, we don't even 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; // ans = ans * a;
  return ans;
}
```

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

Constant parameters

- We may declare a parameter as a constant variable:

```
int factorial (const int n)
{
  int ans = 1;
  for (int a = 1; a <= n; a++)
    ans *= a; // ans = 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 needs not to be a constant variable.

Why function overloading?

- There is a function
  - `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

- To make our lives easier, C++ provides **function overloading**.
- We can define many functions having **the same name** if their parameters are not the same.
- So we don't need to memorize a lot of function names.
  - `int pow (int, int);`
  - `double pow (int, double);`
  - `double pow (double, int);`
  - `double pow (double, double);`

Function signature

- Different functions must have different **function signatures**.
  - This allows the computer to know which function is called.
- A function signature includes
  - Function name.
  - Function parameters (**number** of parameters and their **types**).
- 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.

When to use function overloading?

- Almost all functions in the C++ standard library are overloaded, so we can use them conveniently.
- It can apply to our self-defined functions. But if you are not familiar to it now, it doesn't matter.

An example

- Write two functions
  - `void print(char c, int num);`
  - `void print(char c);`

  that 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

- In the previous example, it is identical to assign `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 radius, double pi = 3.14);`
  - `double circleArea (double, double = 3.14);`
- 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

- Default arguments must be assigned before the function is called.
  - In a function declaration or a function definition.
- 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.
- Function overloading is clearer though more time-consuming.

Inline functions

- When we call a function, we need to do a lot of works.
  - Allocating memory spaces for parameters.
  - Copying and passing values as arguments.
  - Record where we are in the calling function.
  - Pass the program execution to the called function.
  - 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

- 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 calling function and executed directly.
- While this saves some time, it also expands the program size.
- In most cases, programmers do not use inline functions.

Outline

- Basics of functions
- More about functions
  - Function overloading
  - Default function arguments
  - Inline functions
- **Variable lifetime**
- Randomization

Variable lifetime

- There are four levels of variable lifetime (life scope) in C++ that we are ready to understand.
  - local, global, external, static.
- We'll discuss more types of variables in the lectures for classes and objects.

Local variables

- A variable declared in a **block**.
- It lives from the declaration to the end of block.
- In the block, it will **hide** other variables with same name.

```
int main()
{
  int i = 50; // it will be hidden
  for(int i = 0; i < 20; i++)
  {
    cout << i << " "; // print 0 1 2 … 19
  }
  cout << i << endl; // ?
  return 0;
}
```

Global variables

- A variable declared **outside** any block (thus outside the main function)
- Its lives from declaration to the end of program execution.
- it will be **hidden** by any local variable with the same name.

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

int i = 5;

int main()
{
  for(; i < 20; i++)
    cout << i << " "; // ?
  return 0;
}
```

Global variables: Using "::"

- To access a global variable, use the scope resolution operator `::`.

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

int i = 5;

int main()
{
  for(int i = 0; i < 20; i++)
    cout << ::i << " ";  // 5 ... 5
  return 0;
}
```

Local and global variables

- We may add **`auto`** to declare a local or global variable, but since it is the default setting, almost no one adds this.
- There's no difference in the way you declare a local or global variable. The **place** differs.

External variables

- In a large-scale system, many programs run together.
- If a program wants to access a variable **defined in another program**, it can declare the variable with the key word **`extern`**.
  - **`extern int a;`**
  - **`a`** must has been defined in another program.
- These programs must run together.
- You won't need this now… maybe neither in the future.

Static variables

- The memory space allocated to a **static** variable will not be released until the program terminates.
- Once a static variable is declared, all other declaration statements will not be executed.
- A static global variable cannot be declared as external in other programs.

Static variables

```
int test();
int main()
{
  for (int a = 0; a < 10; a++)
    cout << test() << " ";
  return 0; // 1, 1, ..., 1
}
int test()
{
  int a = 0;
  a++;
  return a;
}
```

```
int test();
int main()
{
  for (int a = 0; a < 10; a++)
    cout << test() << " ";
  return 0; // 1, 2, ..., 10
}
int test()
{
  static int a = 0;
  a++;
  return a;
}
```

Summary and good programming style

- You have to distinguish local and global variables.
  - Try to **avoid** global variables!
  - One particular situation to use global variables is to define **constants**.
  - Try to use local variables to replace global variables.
- You may not need static and external variables now or even in the future.
- At least we need to know these things exist.

Outline

- Basics of functions
- More about functions
  - Function overloading
  - Default function arguments
  - Inline functions
- Variable lifetime
- **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.
  - What are other applications of random numbers?
- In C++, randomization can be done with two functions, **`srand()`** and **`rand()`**.
- They are defined in **`<cstdlib>`**.

Random Numbers: `rand()`

- **`int rand();`**
- It will return a **pseudo-random integer** between 0 and 32767.
- Example "**06_03_random**":

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

- What will happen if we execute it for multiple times?

Random Numbers: 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.

Random Numbers: srand()

- **void srand(unsigned int);**
- It will produce a **seed** for the pseudo-random function.

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

- Why still all the same?

Random Numbers: srand()

- **srand(x)** will create a seed by input **x** into another complicated function.
- Thus when you do **srand(0)**, you still obtains the same sequence for each execution.
- To solve this, try to give **srand()** **different arguments**.
- In most cases, we may 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 cannot be explained now.

Random Numbers: srand() and time()

```
int rn;
srand(time(0));
for (int i = 0; i < 10; i++)
{
  rn = rand();
  cout << rn << " ";
} // OK~ :>
```

```
int rn;
for (int i = 0; i < 10; i++)
{
  srand(time(0));
  rn = rand();
  cout << rn << " ";
} // not ok... / \
```

- In a computer, do the **for** loop for 10 times requires a very short time, and **time()** returns a count of seconds, thus all the parameters of **srand()** are (almost always) the same.

Random Numbers: In a Range

- If you want to produce random numbers in a specific range, use %.

```
srand(time(0));
int rn;
for (int i = 0; i < 10; i++)
{
  rn = ((rand() % 10)) + 100;
  cout << rn << " ";
} // what is the range?
```

An example

- Write a program to produce 10 random numbers, which are rational numbers that uniformly distributed between 0 and 5.

```
srand(time(0));
double rn;
for(int i = 0; i < 10; i++)
{
  rn = (static_cast<double>(rand() % 501)) / 100;
  cout << rn << " ";
} // 0 <= rn <= 5
```

  – Do not forget casting!