Programming Design

Functions

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Functions

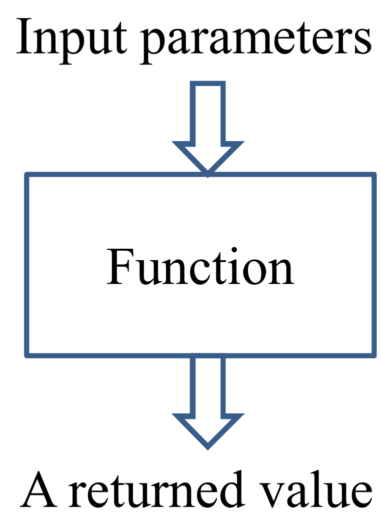
- In C++ and most modern programming languages, we may put statements into **functions** to be **invoked** in the future.
  - Also known as **procedures** in some languages.
- Why functions?
- We need **modules** instead of a huge main function.
  - Easier to divide the works: **modularization**.
  -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

Outline

- **Basics of functions**
- Scope of variables revisited
- More about functions

Structure of functions

- In C++, a function is composed of a **header** and a **body**.
- A header for **declaration**:
  - A function name (identifier).
  - A list of input parameters.
  - A return value.
- A body for **definition**:
  - Statements that define the task.
- Let's start with an example.



Function definition

- There is an **add()** function:
- In the main function we invoke (call) the **add()** function.
- Before the main function, there is a function **header/prototype** declaring the function.
- After the main function, there is a function **body** defining the function.

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

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 declaration

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

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

- In a function prototype, we declare its **appearance** and input/output **format**.

```
int add (int, int);
```

- The name of the function follows the same rule for naming variable.
- A list of (zero, one, or multiple) **parameters**:
  - The parameters passed into the function with their types.
  - We must declare their **types**. Declaring their names are optional.
- A **return type** indicates the type of the function return value.

Function declaration

- Some examples of function prototype:
  - A function receives two integers and returns an integer.
  - The parameter names may provide "hints" to what this function does.

```
int add (int num1, int num2);
int add (int, int);
```

  - A function receives two `double` and returns one `double`.

```
double divide (double, double);
double divide (double num, double den);
```

- For a function declaration, the **semicolon** is required.
- Every type can be the return type.
  - It may be "`void`" if the function returns nothing.

Creating 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.
- In a function prototype, we do not need to specify parameter **names**.
  - But in a function definition, we need!
- These parameters can be viewed as **variables** declared **inside** the function.
  - They can be accessed only in the function.

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

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
}
```

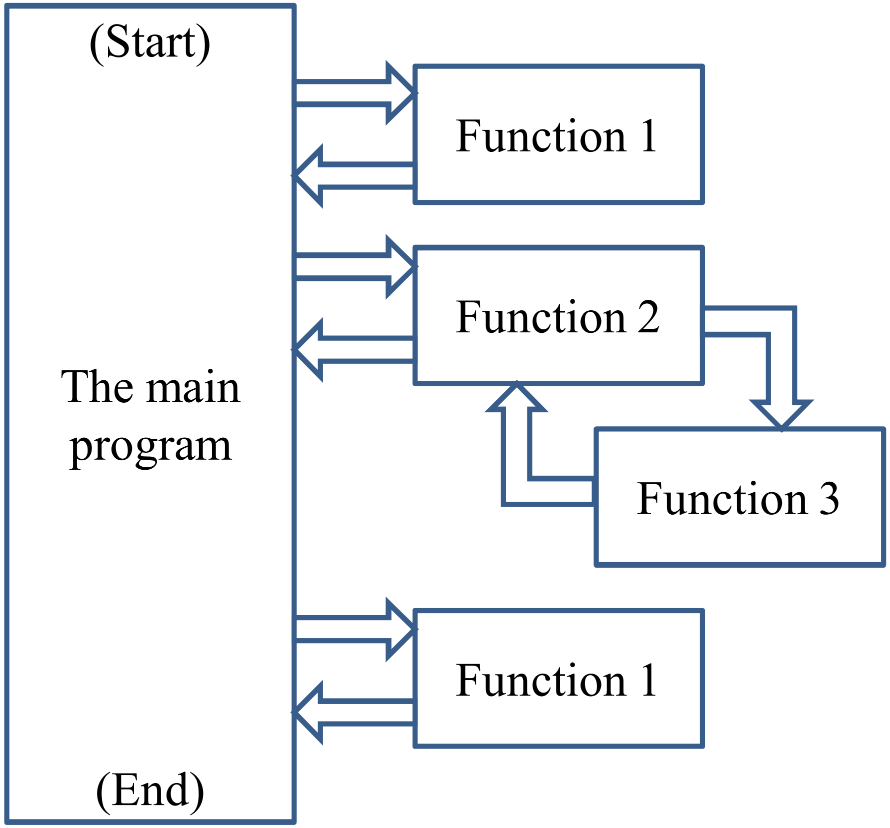
  - The first line, the function header, is almost identical to the prototype.
  - The parameter **names** must be specified.
  - Statements are then written for a specific task.

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

- The keyword **return** terminates the function execution and returns a value.

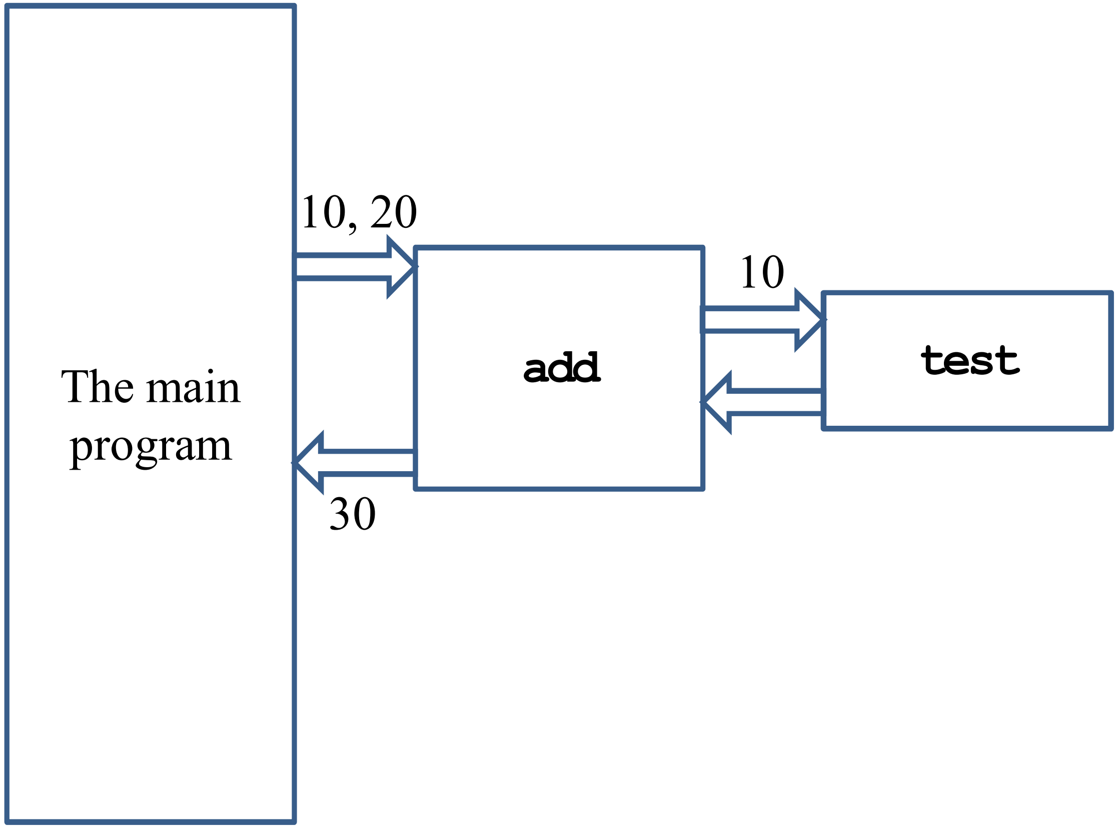
Function invocation

- 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?



Function invocation

```
int add (int, int);
void test (int);
int main ()
{
  int c = add(10, 20);
  cout << c << endl;
  return 0;
}
int add (int num1, int num2)
{
  test (num1);
  return num1 + num2;
}
void test (int toPrint)
{
  cout << toPrint << endl;
}
```



Function declaration and definition

- You may choose to define a function before the main function.
  - In this case, the function prototype can be omitted.
- In any case, you must declare a function **before** you use it.

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

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

```
void a()
{
  // error!
  b();
}
void b()
{
  ;
}

int main ()
{
  a();
  b();
  return 0;
}
```

Function declaration and definition

- In some cases, function prototypes must be used.

```
void a()
{
  // error!
  b();
}
void b()
{
  a();
}
int main ()
{
  a();
  b();
  return 0;
}
```

```
void a();
void b();
int main ()
{
  a();
  b();
  return 0;
}
void a()
{
  // fine!
  b();
}
void b()
{
  a();
}
```

  - Direct or indirect self-invocations are called **recursion** (a topic to be discussed in the next lecture).
- Using function prototypes also enhances communications and maintenance.

Function parameters vs. arguments

- When we invoke a function, we need to provide **arguments**.
  - **Parameters** are used inside the function.
  - **Arguments** are passed into the function.
- If a pair of parameter and argument are both variables, their names can be different.
- Let's visualize the memory events.

```
int add (int num1, int num2)
{
  return num1 + num2;
}
int main ()
{
  double q1 = 10.5;
  double q2 = 20.7;
  double c = add(q1, q2); // !
  cout << c << endl;
  return 0;
}
```

Function arguments

- Function arguments can be:
  - Literals.
  - Variables.
  - Constant variables.
  - Expressions.
- If an argument's type is different from the corresponding parameter's type, compiler will try to **cast** it.

```
int add (int, int);
int main ()
{
  const int C = 5;
  double d = 1.6;
  cout << add(10, 20) << endl;
  cout << add(C, d) << endl; // !
  cout << add(10 * C, 20) << endl;
  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.
  - Or we may write **`return;`**.
  - Otherwise, having no **`return`** statement results in a compilation error.

Function return value

- There can be multiple **`return`** statements.
- A function runs until the **first** **`return`** statement is met.
  - Or the end of the function for a function returning **`void`**.
- We need to ensure that at least one return will be executed!

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

```
int test (int);

int main()
{
  cout << test(-1);
  return 0;
}

int test (int a)
{
  if (a > 0)
    return 5;
}
```

Example

- What do these two functions do?

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

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

- Which one to choose?

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 comments.
  - Ideally, other programmers can understand what a function does without reading the definition.
- Declare all functions at the beginning of the program.

Outline

- Basics of functions
- **Scope of variables revisited**
- More about functions

Variable lifetime

- Four levels of variable lifetime (life scope) in C++ can be discussed now.
  - local, global, external, and static.
- We'll discuss more types of variables in this semester.

Local variables

- A **local** variable is 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; // 50
  return 0;
}
```

Global variables

- A **global** variable is declared **outside** any block (thus outside the main function)
  - From declaration to the end of the program.
- It will be **hidden** by any local variable with the same name.
  - To access a global variable, use the scope resolution operator `::`.
- There's no difference in the way you declare a local or global variable. The **locations** matter.
- We may add **`auto`** to declare a local or global variable, but since it is the default setting, almost no one adds this.

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

int i = 5;

int main()
{
  for(; i < 20; i++)
    cout << i << " "; // ?
  cout << endl;
  int i = 2;
  cout << i << endl; // ?
  cout << ::i << endl; // ?
  return 0;
}
```

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 will not need this now… actually you should try to **avoid** it.
  - It hurts modularization and makes the system hard to maintain.
  - Though it still exists in some old systems (e.g., some BBS sites).
- Note that global variables should be avoided for the same reason.

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;
}
```

- When do we use a static variable?

Good programming style

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

Outline

- Basics of functions
- Scope of variables revisited
- **More about functions**

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;
}
```

Passing an array as an argument (1/3)

- An array can also be passed into a function.
  - Declaration: need a `[]`.
  - Invocation: use the array name.
  - Definition: need a `[]` and a name for that array in the function.
- We do not need to indicate the size of the array!
  - An array variable stores an address.
  - "Passing an array" is actually telling the function how to access the array.
- Let's visualize the memory events.

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

Passing an array as an argument (2/3)

- It is fine if we indicate the array size.
  - But no new memory space will be allocated accordingly.
  - That number will just be ignored.
  - They can even be inconsistent.

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

Passing an array as an argument (3/3)

- We may also pass multi-dimensional arrays.
- The *k*th-dimensional array size must be specified for all $k \geq 2$!
  - Just like when we declare a multi-dimensional array.
- Now they must be consistent.

```
void printArray (int [][2], int);
int main()
{
  int num[5][2] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
  printArray(num, 5);
  return 0;
}
void printArray (int a[][2], int len)
{
  for (int i = 0; i < len; i++)
  {
    for (int j = 0; j < 2; j++)
      cout << a[i][j] << " ";
    cout << endl;
  }
}
```

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.