CPT-182 - Evening - Programming in C++

Lecture 4

User-Defined Functions

Dayu Wang

- **In computer programming, sometimes function and method can be used interchangeably.**

  If the function is defined outside classes, it is called function.

  If the function is part of a class, it is called a method of the class.

Syntax of defining a function in C++

```
return_type function_name(parameter_list) {
    function_statements
}
```

→ **The `return_type`**

Every C++ function must have a return type, except ~~class constructors~~ (in future lectures).

Return type is the data type of the return value of the function.

If a function does not ~~return a value~~, then its return type is `void`.

`void` is a return type, meaning that the function does not ~~return a value~~.

→ **The `parameter_list`**

Inside the parentheses, you need to list each parameter's data type and name, just like you are declaring variables.

Use comma to separate parameters.

→ [Example] Define a function that calculates the volume of a cuboid.

[Good Habit] Before writing code, you need to clearly know the following 3 things:

1) How many arguments (parameters) the function takes?

This function should take 3 arguments, length, width, and height of the cuboid.

2) What is the data type of each argument?

length: double, width: double, height: double

3) What is the return type of the function?

[Fact] Frequently, the return value of a function is the calculation result.

This function returns the calculated cuboid volume, which is a double.

Therefore, the return type of this function is double.

```
1  /** Calculates the volume of a cuboid.
2      @param length: length of the cuboid
3      @param width: width of the cuboid
4      @param height: height of the cuboid
5      @return: calculated volume of the cuboid
6  */
7  double cuboid_volume(double length, double width, double height) {
8      return length * width * height;
9  }
```

Function docstring is a must.

• [Exercise] Define a function that calculates the volume of a cylinder.

→ What shape is the end of a cylinder?

Circle

→ How many arguments that function cylinder_volume() takes?

It takes 2 arguments, which are the radius of the end circle and the length of the cylinder.

→ What is the data type of each argument?

radius: double, length: double.

→ What is the function return type?

double

```
1  /** Calculates the volume of a cylinder.
2      @param radius: radius of the end circle of the cylinder
3      @param length: length of the cylinder
4      @return: calculated volume of the cylinder
5  */
6  double cylinder_volume(double radius, double length) {
7      const double PI = std::atan(1) * 4;  // There is no pre-defined "π" in C++.
8      return PI * std::pow(radius, 2) * length;
9  }
```

What is this?

- **Preprocessor Directives**

→ **Preprocessor directives are lines included in the code of programs preceded by a hash sign (#).**

These lines are not ~~program statements~~ but directives for the preprocessor.

The preprocessor examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statements.

→ **These preprocessor directives extend only across a single line of code.**

As soon as a newline character is found, the preprocessor directive ends.

No ~~semicolon~~ (;) is expected at the end of a preprocessor directive.

The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

- **Macro Definitions (#define, #undef)**

Syntax of macro definition

```
#define identifier replacement
```

→ **When the preprocessor encounters this directive, it replaces any occurrence of `identifier` in the rest of the code by `replacement`.**

This `replacement` can be an expression, a statement, a block or simply anything.

Preprocessor cannot ~~understand C++ code~~, it simply replaces any occurrence of `identifier` by `replacement`.


```
1  #define PI std::atan(1) * 4
2
3  /** Calculates the volume of a cylinder.
4      @param radius: radius of the end circle of the cylinder
5      @param length: length of the cylinder
6      @return: calculated volume of the cylinder
7  */
8  double cylinder_volume(double radius, double length) {
9      return PI * std::pow(radius, 2) * length;
10 }
```

→ **Can we define the same entity multiple times?**

The redefinition will overwrite the previous definition.

```
1  int main() {
2  #define message "Hello"
3      cout << message << endl;
4  #define message "Welcome"
5      cout << message << endl;
6      system("pause");
7      return 0;
8  }
```

Console
```
Hello
Welcome
```

[Good Habit] Use `#undef` before redefine a macro.

[Good Habit] Define all the macros at the top of the file.

• **[Exercise] Define a function that shows message "Welcome" in the console.**

```
1  /** Shows message "Welcome" in the console. */
2  void welcome_message() {
3      std::cout << "Welcome" << std::endl;
4  }
```

Why this function returns void, not ~~string~~?

"Welcome" is the output value, not the ~~return value~~.

After we show "Welcome" in the console, there is no ~~return value~~ for the function.

[Common Mistake] Confused between return value and output value.

→ **Return value and output value**

A function could return something but output nothing.

A function could return nothing but output something.

A function could return something and output something.

A function could return nothing and output nothing.

Return value and output value are different concepts (independent to each other).

• **Can we call a function first in the main() function, and define it later?**

→ **Yes, as long as the function is declared before calling.**

Syntax to declare a function

```
return_type function_name(list_of_data_type_of_each_parameter);
```

```
1  // Functions to be used in the main() function
2  double rectangle_area(double, double);
3  void print_double(double);
4
5  int main() {
6      double width = 13, height = 10;
7      double area = rectangle_area(width, height);
8      print_double(area);
9      system("pause");
10     return 0;
11 }
12
13 double rectangle_area(double width, double height) { return width * height; }
14
15 void print_double(double area) { cout << "Area: " << setprecision(2) << fixed << area << endl; }
```

~~Argument names~~ can be absent.
Argument types cannot be absent.

```
Console  Area: 130.00
```

- **Can we define a function in one file, and call it in another file?**

  → **Normally, there are two types of files in a C++ project.**

  Header file (.h)

  Source file (.cpp)

  → **The `main()` function resides in a `.cpp` file.**

  → **A `.cpp` file can `#include` a header file, making all entities defined in the header file available in the `.cpp` file.**

File Data_Processor.h

```
1  #ifndef DATA_PROCESSOR_H
2  #define DATA_PROCESSOR_H
3
4  #include <algorithm>
5
6  // Finds the maximum value among 4 integers.
7  int max_value(int x1, int x2, int x3, int x4) {
8      return std::max(std::max(x1, x2), std::max(x3, x4));
9  }
10
11 #endif
```


File Program.cpp

```
1  #include "Data_Processor.h"
2
3  #include <iostream>
4
5  using namespace std;
6
7  int main() {
8      int a = 3, b = -3, c = -2, d = 0;
9      cout << max_value(a, b, c, d) << endl;
10     system("pause");
11     return 0;
12 }
```

Console

```
3
```

→ **When to use "", when to use <>?**

For user-defined header files, use "".

For system build-in headers, use <>.

→ **[Good Habit] The `.cpp` file containing the `main()` function should only contain the `main()` function.**

Functions should be written in separate files.

- **Conditional Inclusions** (#ifdef, #ifndef, #if, #endif, #else, and #elif)

→ These directives allow to include or discard part of the code of a program if a certain condition is met.

#ifdef allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter what its value is.

```
1  /** Calculates the volume of a cylinder.
2      @param radius: radius of the end circle of the cylinder
3      @param length: length of the cylinder
4      @return: calculated volume of the cylinder
5  */
6  double cylinder_volume(double radius, double length) {
7  #ifdef PI
8      return PI * pow(radius, 2) * length;
9  #else
10     cout << "[ERROR] \"π\" is not defined." << endl;
11     return -1;
12 #endif
13 }
```


→ The most frequently used conditional inclusion in this class is #ifndef.

In every header file, you must put the following.

```
1  #ifndef file_name(all_capital_letters_and_replace_dot_with_underscore)
2  #define file_name(all_capital_letters_and_replace_dot_with_underscore)
3
4  // All your code in the file
5
6  #endif
```

This is used to avoid ~~multiple inclusion~~.

File String_Processor.h

```
1  #include <iostream>
2  #include <string>
3
4  void to_uppercase(std::string& s) {
5      for (size_t i = 0; i < s.size(); i++) {
6          s[i] = std::toupper(s[i]);
7      }
8  }
9
10 void print_string(const std::string& s) {
11     std::cout << s << std::endl;
12 }
```

File String_Printer.h

```
1  #include "String_Processor.h"
2
3  /** Prints a string in uppercase.
4      @param s: string to be printed
5  */
6  void print_uppercase_string(std::string& s) {
7      to_uppercase(s);
8      print_string(s);
9  }
```

```cpp
#include "String_Printer.h"
#include "String_Processor.h"

#include <iostream>

using namespace std;

int main() {
    string s = "abcd";
    print_uppercase_string(s);
    system("pause");
    return 0;
}
```

Runtime Error

Console

```
function 'std::string to_uppercase(std::string &)' already has a body
function 'void print_string(const std::string &)' already has a body
```

We included `String_Processor.h` in `String_Printer.h`.

We included `String_Processor.h` in `Main.cpp`.

We ran the code in `String_Processor.h` twice.

All the functions defined in `String_Processor.h` will be defined twice, which is not allowed.

Using `#ifndef` can solve this issue perfectly.


```cpp
#ifndef STRING_PROCESSOR_H
#define STRING_PROCESSOR_H

#include <iostream>
#include <string>

/** Converts a string to uppercase.
    @param s: contains letters only
*/
void to_uppercase(std::string& s) {
    for (size_t i = 0; i < s.size(); i++) {
        s[i] = std::toupper(s[i]);
    }
}
/** Prints a string in the console.
    @param: string to be printed
*/
void print_string(const std::string& s) {
    std::cout << s << std::endl;
}

#endif
```

Console

```
ABCD
```

```cpp
#ifndef STRING_PRINTER_H
#define STRING_PRINTER_H

#include "String_Processor.h"

void print_uppercase_string(std::string& s) {
    to_uppercase(s);
    print_string(s);
}
#endif
```

```cpp
#include "String_Printer.h"
#include "String_Processor.h"

#include <iostream>

using namespace std;

int main() {
    string s = "abcd";
    print_uppercase_string(s);
    system("pause");
    return 0;
}
```

• **[Exercise] Define a function that returns the maximum value in a vector of integers.**

```cpp
// Finds the maximum value in a vector of integers.
int max_value(const std::vector<int>& vec) {
    int result = INT_MIN;
    for (int val : vec) {
        if (val > result) { result = val; }
    }
    return result;
}
```

• **Pass-by-Value and Pass-by-Reference**

→ **Define a function to swap two integers.**

```cpp
// Swaps two integers.
void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}
```

Can the function work as expected?

Expected output `a = 5, b = 3`

```cpp
int main() {
    int a = 3, b = 5;
    swap(a, b);
    cout << "a = " << a << ", b = " << b << endl;
    system("pause");  return 0;
}
```

There was no ~~error~~, but it seems like that the function did not do its job.

Was the swap algorithm correct?

The algorithm is correct.

Console `a = 3, b = 5`

→ **C++'s default: pass-by-value**

A function argument will be passed into the function by value.

It means that only the value of the argument will be used inside the function.

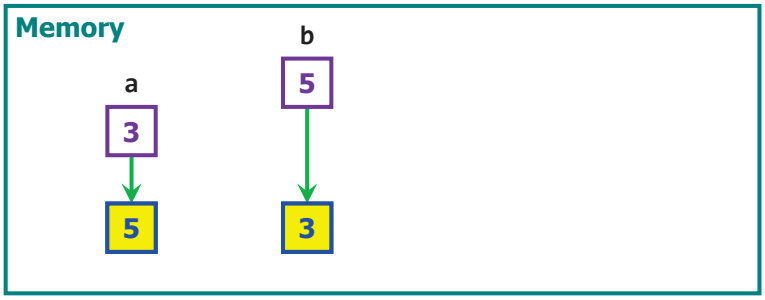
→ **What happens at the back end when `swap(a, b)` is being executed?**

Initially, `a` and `b` are stored in the memory.

When the system passes `a` and `b` to the `swap()` function...

1) **A copy of `a` is made.**
2) **A copy of `b` is made.**
3) **The two copies are used in the function.**
4) **The function swaps the two copies.**
5) **After executing the function, the two copies are "cleaned up" (out of scope).**
6) **So, the "original copies", `a` and `b`, are never ~~changed~~.**

```cpp
int main() {
    int a = 3, b = 5;
    swap(a, b);
    cout << "a = " << a << ", b = " << b << endl;
    system("pause");  return 0;
}
```



→ **Any changes to the argument made inside the C++ function will not be reflected outside the function.**

This is called pass-by-value.

→ **How to let `swap()` function really work? How can we let the change to an argument made inside the function be reflected outside the function?**

We need to pass the arguments by reference.

```
1  // Swaps two integers.
2  void swap(int& x1, int& x2) {
3      int temp = x1;
4      x1 = x2;
5      x2 = temp;
6  }
```

```
1  int main() {
2      int a = 3, b = 5;
3      swap(a, b);
4      cout << "a = " << a << ", b = " << b << endl;
5      system("pause");  return 0;
6  }
```

'&' tells the compiler "not to ~~make a copy~~" of the argument value.

The system will use the "original copy" directly.

```
Console  a = 5, b = 3
```


→ **[Exercises] What are the output values?**

```
1  // Swaps two integers.
2  void swap(int x1, int x2) {
3      int temp = x1;
4      x1 = x2;
5      x2 = temp;
6  }
```

```
1  int main() {
2      int a = 3, b = 5;
3      swap(a, b);
4      cout << "a = " << a << ", b = " << b << endl;
5      system("pause");  return 0;
6  }
```

```
Console  a = 3, b = 5
```

```
1  // Swaps two integers.
2  void swap(int& x1, int& x2) {
3      int temp = x1;
4      x1 = x2;
5      x2 = temp;
6  }
```

```
1  int main() {
2      int a = 3, b = 5;
3      swap(a, b);
4      cout << "a = " << a << ", b = " << b << endl;
5      system("pause");  return 0;
6  }
```

```
Console  a = 5, b = 3
```

```cpp
// Swaps two integers.
void swap(int& x1, int x2) {
    int temp = x1;
    x1 = x2;
    x2 = temp;
}
```

```cpp
int main() {
    int a = 3, b = 5;
    swap(a, b);
    cout << "a = " << a << ", b = " << b << endl;
    system("pause");  return 0;
}
```

```
Console  a = 5, b = 5
```

```cpp
// Swaps two integers.
void swap(int x1, int& x2) {
    int temp = x1;
    x1 = x2;
    x2 = temp;
}
```

```cpp
int main() {
    int a = 3, b = 5;
    swap(a, b);
    cout << "a = " << a << ", b = " << b << endl;
    system("pause");  return 0;
}
```

```
Console  a = 3, b = 3
```


→ **[Example] Swap two elements at index `i` and index `j` in an array of integers.**

```cpp
// Swaps two elements in an array.
void swap(int a[], size_t size, size_t i, size_t j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

```cpp
int main() {
    int arr[] = { 1, 2, 3, 4, 5 };
    swap(arr, 5, 0, 2);
    copy(begin(arr), end(arr),
         ostream_iterator<int>(cout, ", "));
    system("pause");
    return 0;
}
```

```
Console  3, 2, 1, 4, 5
```

→ **If an argument is a regular array (not ~~vector~~), you need not put '&' and it will always be passed by reference.**

• **[Exercise] Define a function to reverse a vector of integers in-place. The function returns `void`.**

```cpp
// Reverse a vector in-place.
void reverse(vector<int> vec) {
    size_t i = 0, j = vec.size() - 1;
    while (i < j) { swap(vec[i++], vec[j--]); }
}
```

```cpp
int main() {
    vector<int> vec = { 1, 2, 3, 4, 5 };
    reverse(vec);
    copy(vec.begin(), vec.end(),
         ostream_iterator<int>(cout, ", "));
    system("pause");
    return 0;
}
```

```
Console  1, 2, 3, 4, 5
```

What's wrong here?

→ The default is pass-by-value in C++, no matter the argument is value type or reference type.

→ A function argument may be huge (e.g., a vector that has 1,000,000 elements).

If passed by value (default), then a copy has to be made before passing into the function.

Making copies of huge arguments are very time-consuming and wasteful of precious memory.

```
1  // Function to reverse a vector in-place.
2  void reverse(vector<int>& vec) {
3      size_t i = 0, j = vec.size() - 1;
4      while (i < j) { swap(vec[i++], vec[j--]); }
5  }
```

→ Therefore, unless the argument is primitive data type, all arguments must be passed by reference.

Primitive data types: `int`, `unsigned int`, `long long`, `unsigned long long`, `double`, `char`, and `bool`.

→ If an argument is passed by reference, the function can directly change it.

This increases the danger of programming.

→ So, if you want an argument to be passed by reference but cannot be ~~changed~~ by the function, you need to pass it by `const` reference.

`const` means that any code inside the function cannot ~~change the argument~~; otherwise, the code will not ~~compile~~.

→ You will lose 1 point each time when you forget to pass argument of non-primitive data type by reference or `const` reference.

```
1  // Return the maximum value in a vector.
2  int max_value(const vector<int>& vec) {
3      int max_val = INT_MIN;
4      for (int val : vec) {
5          if (val > max_val) { max_val = val; }
6      }
7      return max_val;
8  }
```


• [Common Mistake] You could lose lots of points in your assignments/projects.

→ You should pass all arguments by reference, unless it is a primitive data type.

You will lose points if you pass a non-primitive data type argument by value.

```
1  string remove_spaces(string s) {
2      string result;
3      for (char ch : s) {
4          if (ch != ' ') {
5              result.push_back(ch);
6          }
7      }
8      return result;
9  }
```

Wrong!

- **Scope of variables and functions**

  → The name of a defined variable or function is only visible to part of a program, known as the variable's or function's scope.

  A variable declared in a function has scope limited to inside that function.

  In fact, because a compiler scans a program line-by-line from top-to-bottom, the scope starts after the declaration until the function's end.

  → A variable declared outside any function is called a global variable, in contrast to a local variable declared inside a function.

  A global variable's scope extends after the declaration to the file's end, and reaches into functions.

  Any function can access and change a global variable.

  → Just as `goto` statements, you are not allowed to ~~use global variables~~ in your program for whatever reason.

  However, you can use global `const` variables (global constants).

  → A function also has a scope, which extends from its declaration to the end of the file.

  → If your program contains only a single file...

  You need to put all the function declarations above the `main()` function.

  And put all the function definitions (implementation) below the `main()` function.

- **Default Values**

  → Sometimes a function's last parameter (or last few) should be optional.

```
1  void print_date(int year, int month, int date, const std::string& format);
```

  For example, if the format is not specified, then the functions uses "m/d/yyyy".

  → A function call could then omit the last argument, and instead the program would use a default value for that parameter.

  → A function can have a default parameter value for the last parameter(s), meaning a call can optionally omit a corresponding argument.

```
1  void print_date(int year, int month, int date, int format = 0) {
2      if (!format) {  // American
3          cout << month << '/' << date << '/' << year << endl;
4      } else if (format == 1) {  // European
5          cout << date << '/' << month << '/' << year << endl;
6      } else {
7          cout << "Invalid style" << endl;
8      }
9  }
```

```
1  int main() {
2      print_date(2022, 2, 15, 0);
3      print_date(2022, 2, 15, 1);
4      print_date(2022, 2, 15);
5      system("pause");
6      return 0;
7  }
```

Console
```
2/15/2022
15/2/2022
2/15/2022
```

You can only give default values to the last (or last few) parameters.

```
1  void print_date(int year = 2022, int month, int date, int format);
```

If you first declare the function and then implement it (separately), then you must give default values in declaration and not in ~~implementation~~.


```
1  void print_date(int, int, int, int = 1);
2
3  int main() {
4      print_date(2022, 2, 15, 0);
5      print_date(2022, 2, 15, 1);
6      print_date(2022, 2, 15);
7      system("pause");
8      return 0;
9  }
10
11 void print_date(int year, int month, int date, int format){
12     if (!format) {
13         cout << month << "/" << date << "/" << year << endl;
14     } else if (format == 1) {
15         cout << date << "/" << month << "/" << year << endl;
16     } else {
17         cout << "Invalid Style" << endl;
18     }
19 }
```

Default value only given in the declaration

Not in the ~~implementation~~

- **Function Overloading**

→ **Can we define two functions with the same name?**

This is called function overloading.

```
1   // Swaps two integers.
2   void swap(int& x1, int& x2) {
3       int temp = x1;
4       x1 = x2;
5       x2 = temp;
6   }
7
8   // Swaps two elements at index i and index j in a vector of integers.
9   void swap(vector<int>& arr, int i, int j) {
10      int temp = arr[i];
11      arr[i] = arr[j];
12      arr[j] = temp;
13  }
```

→ **If we call function `swap()` in `main()` function, how can the compiler know which version to use?**

The compiler can see the argument list.

1) **If the argument list contains 2 integers, then it will use the top version.**
2) **If the argument list contains a vector of integers and other 2 integers, then it will use the bottom version.**

→ **The function name can be the same, but the parameter list must be unique.**

```
1   void foo(int a);
2   int foo(int a, int b);
3   double foo(const std::string& s);
4   bool foo(const std::string& s, int a);
5   bool foo(int a, const std::string& s);
6   // All above are allowed.
7   // You can overload the function as many times as you want.
```

→ **You cannot overload the function in the following ways:**

```
1   // There is already a function defined as below:
2   void foo(int a, int b);
```

```
1   void foo(int& a, int& b);  // Compiler cannot distinguish "int" and "int&".
2   unsigned int foo(int a, int b);  // Only return type is different.
```

- **Why we would like function overloading?**

→ **Same function name works for different data types.**

Based on different data types, the function behaves differently.

→ **Extend the build-in functions, letting them work for user-defined data types.**

Overloading operators (in future lectures)

- **Functions with I/O**

```
1  ostream& print_str(const string& s, ostream& out) {
2      out << s << endl;
3      return out;
4  }
```

→ ostream is "output stream".

cout is an ostream. ofstream is an ostream.

→ print_str() can output the string to any kind of output stream.

ostream must be passed by reference (not ~~const reference~~).

→ Do we need to open the output stream in the function?

When you define a function, you assume that the arguments are already available.

When you write a recipe, you assume that all ingredients are already available.

```
1  int main() {
2      print_str("abcd", cout);
3      system("pause");
4      return 0;
5  }
```

Console
```
abcd
```

```
1  int main() {
2      ofstream fout("output.txt");
3      print_str("abcd", fout);
4      fout.close();
5      return 0;
6  }
```

output.txt
```
abcd
```

→ Do we need to close the output stream at the end of the function?

We will do this in the main() function, not ~~the user-defined function~~.

- **How about input streams?**

```
1  istream& read_str(istream& in, string& s) {
2      in >> s;
3      return in;
4  }
```

→ istream is "input stream".

cin is an istream. ifstream is an istream.

→ read_str() can read the string from any kind of input stream.

istream must be passed by reference (not ~~const reference~~).

→ Do we need to open the input stream in the function?

When you define a function, you assume that the arguments are already available.

When you write a recipe, you assume that all ingredients are already available.

```
1  int main() {
2      string s;
3      cout << "Enter a string: ";
4      read_str(cin, s);
5      cout << "String you entered: ";
6      cout << s << endl;
7      system("pause");
8      return 0;
9  }
```

```
1  int main() {
2      ifstream fin("input.txt");
3      string s;
4      read_str(fin, s);
5      cout << "String read: ";
6      cout << s << endl;
7      fin.close();
8      return 0;
9  }
```

→ Do we need to close the input stream at the end of the function?

We will do this in the main() function, not ~~the user-defined function~~.

- **[Common Mistakes] You could lose lots of points in your assignments/projects.**

  → **You should pass all arguments by reference, unless it is a primitive data type.**

  You will lose points if you pass a non-primitive data type argument by value.

```
1  string remove_spaces(string s) {
2      string result;
3      for (char ch : s) {
4          if (ch != ' ') {
5              result.push_back(ch);
6          }
7      }
8      return result;
9  }
```

Wrong!

```
1  void print(const string& s, ofstream& out) {
2      out << s << endl;
3  }
```

Wrong!

→ **When defining functions with I/O, you must use `istream` or `ostream`, not ~~`ifstream`~~ or ~~`ofstream`~~!**

→ **The `istream` or `ostream` must be passed by reference (not `const` reference).**

→ **Do not forget the return statement.**