Object-Oriented Programming: Function and Reference

Lectured by Ming-Te Chi 紀明德

Computer Science Department
National Chengchi University

First Semester, 2022

Slides credited from 李蔡彥 and 廖峻鋒

Outline

- Functions
  - Prototype conventions
  - Default arguments
  - Inline functions
  - References
  - Overloading
- Odds and ends
  - **`new`** and **`delete`**
  - name mangling

Function: Prototype Conversion

- Function prototypes are REQUIRED in C++.
- In C, `void` is an argument.

```
int foo(void);   /* function with no arguments */
int foo();       /* function with any number of arguments */
```

- In C++, the above two are synonymous.
  - But, C++ style prefers the prototype *without* void.
- `main()` in C++
  - The compiler *must* support the following two and may support more.

```
int main();
int main(int argc, char *argv[]);
```

Function: Default Arguments (I)

- Function arguments can be given default (optional) values.

```
int sum(int a, int b=6, int c=7);

int main() {
   sum(5);         // the same as sum(5,6,7)
   sum(5,10);      // the same as sum(5,10,7)
   sum(5,10,20);// the same as sum(5,10,20)
}

int sum(int a, int b, int c) {
   return a+b+c;
}
```

Function: Default Arguments (II)

- Rules:
  - A function can have any number of default arguments
  - The default arguments must all come after the non-default ones.
  - The default values are indicated in the function prototype.
- Style
  - Using one or two default arguments is clever. More than two makes it a confusing function.

Function: What is Function Macro

- What is macro?

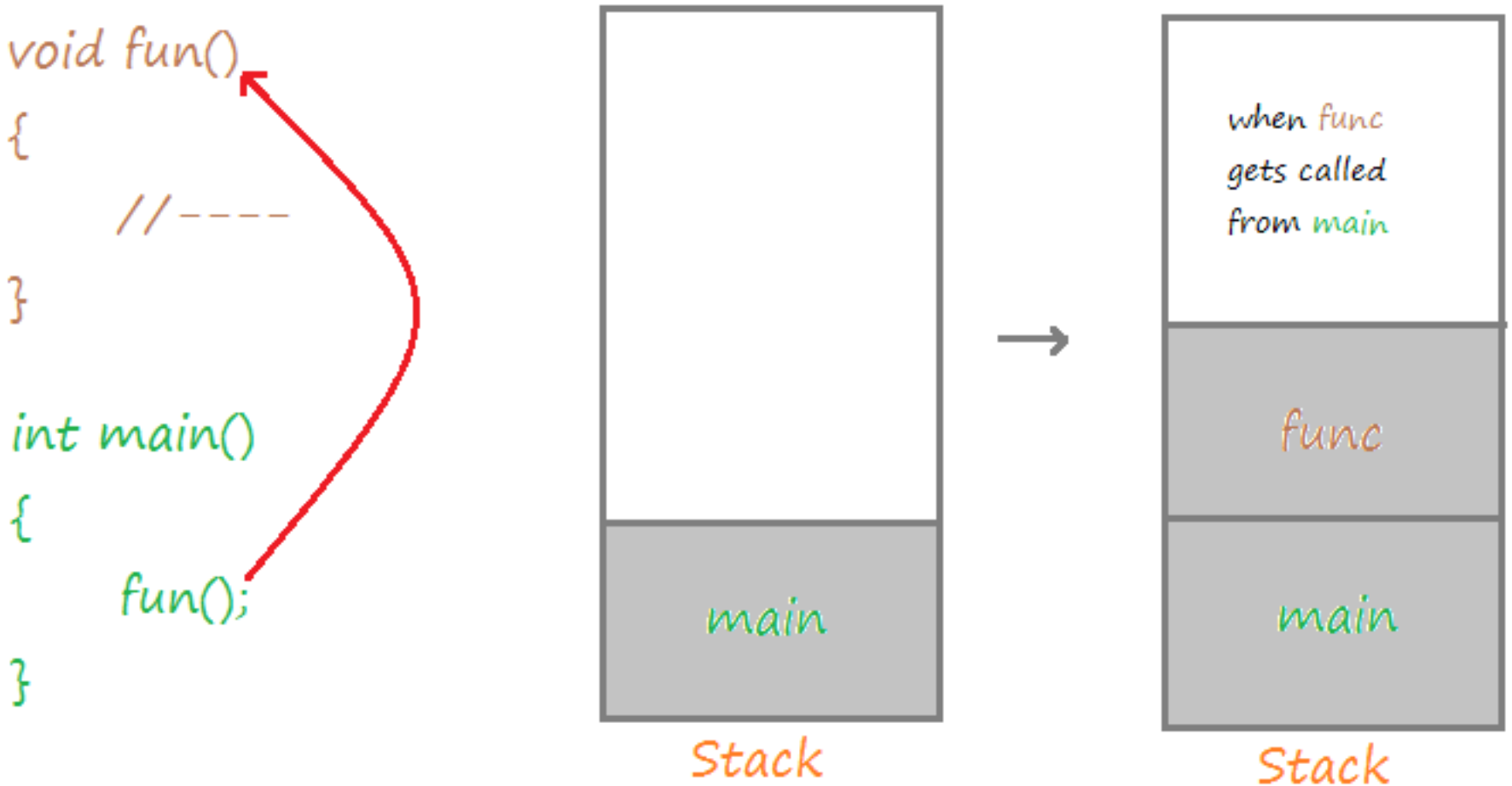
```
#define marco_identifier(parameters) subst_text
```

- Why C programmer use macros?
  - Avoid the overhead of function calls

```
#define AREA(l,w) (l)*(w)   // parameterized macro

a=AREA(length, width);
```

- Disadvantages of macros:
  - You expand the code every time you call the macro.
  - You can't debug a macro function.
  - Macros may have side effects.



https://pencilprogrammer.com/cpp-tutorials/inline-function/

Function: Problems of Function Macro

```
#define square(x) (x*x)
int x=5, y=6, z;
z = square(x+y);
cout << z;
```

Output: 41
Why?

```
#define square(x) ((x)*(x))
int x=5, y;
y = square(--x);
cout << x < " " < y;
```

Output: 3 9
Why?

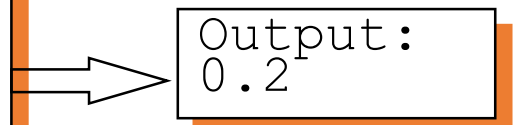
```
#define inverse(x) (1/(x))
int x=5; double y;
y = inverse(x);
cout << y << "\n";
```

Output: 0
Why?

Function: Inline Functions

- Inline functions are functions that are equivalent to macros without the drawbacks.

```
inline double Inverse(double x);
int main() {
   int x=5; double y;
   y = Inverse(x);
   cout << y << "\n";
}
inline double Inverse(double x) {
   return 1/x;
}
```



```
Output:
0.2
```

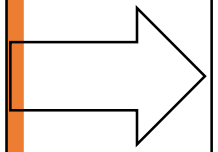
- Rules:
  - The compiler only inlines *simple* functions.
  - Inline functions, like macros, duplicate code. Use sparingly.
  - Never use local `static` variables in `inline` functions.

Introduction to References

- C simulates **call-by-reference** through pointers;
- C++ has *references*.

```
void ReferenceTest(int &inputX)
{
   inputX = 10; // not *x
}

int main() {
   int x=0;
   ReferenceTest(x); // no &
   cout << x;
}
```



Output:

10

Why Use References

- Main reasons for using reference arguments in functions:
  - To alter a data object
  - To speed up a program
- The reference variable acts as an *alias* to the first variable and must be *initialized* in its declaration.

```
int x=5, y = 6;
int &alias = x; //OK

int &alias2;    //error: 'alias2' declared as reference but
not initialized
int &alias3 = y;
alias3 = x;  // equal y=5, but not recommend in the same
scope
```

Reference Is a Resolved Pointer

- Comparing two references is comparing their *values*, not addresses.
- A reference cannot be initialized to another variable. But, *assignment* is different from initialization.

```
int main() {
   int x=5, y=5;
   int &aliasX = x;
   int &aliasX = y; // ERROR
   int &aliasY = y; // OK
   if (aliasX == aliasY)
      cout << "Identical\n";
   else
      cout << "Not identical\n";
   // reference variable assignment
   y = 10;
   aliasX = y;
   x = 15;
   cout << aliasX;
}
```

Output:

Identical

15

Constant Reference Arguments

- There are NO type conversions with references since it references to a temporary variable. *Types must match,* unless a `const` modifier is used.
- Function can return a reference used as an lvalue.

```
struct point {
   int x, y;
};
point &translate(point &p, const point &offset) {
   p.x += offset.x;
   p.y += offset.y;
   return p;
}
int main() {
   point p1 = {1, 2};
   point shift = {1, 1};
   translate(p1, shift).x += 2;
   cout << p1.x << "\n";
}
```

`translate` does not modify offset

Output: 4

Summary on References

- References should be used instead of pointers whenever possible in C++.
- The primary uses for references are the following (listed in order of importance):
  - To achieve call-by-reference with parameters.
  - To avoid copying data for parameters or return types.
  - References can be used to make functions work like lvalues.
  - References can serve as aliases to other variables.

Common Errors on References

- Forgetting that references are not pointers.
- Forgetting that references are not really variables in themselves.
- Using a reference to data that is going out of scope.

Function Overloading

- Functions with the same function name but different number of parameters or different parameter types.

```
void print(const char *str, int width);  // #1
void print(double d,        int width);  // #2
void print(long l,          int width);  // #3
void print(int i,           int width);  // #4
void print(const char *str);             // #5
```

```
print("test", 15);          // use #1
print("Hi");                // use #5
print(1996.0, 10);          // use #2
print(1996,   12);          // use #4
print(1996L,  15);          // use #3
```

```
void print(char *str);
void print(const char *str);
```

⇨ different functions

```
unsigned int year = 1995;
print(year, b); // ambiguous
```

⇨ error !

```
void print(char *str);
int print(char *str);
```

⇨ error !

Name Mangling

- Name mangling:

function prototype → **function signature**

```
int func(char *x, unsigned y);
long func(unsigned x, long y);
```

```
_func_FPScUi
_func_FUiSl
```

- Disabling name mangling for C functions:

```
extern "C" {
   #include <stdlib.h>
   long print(int x);
}
```

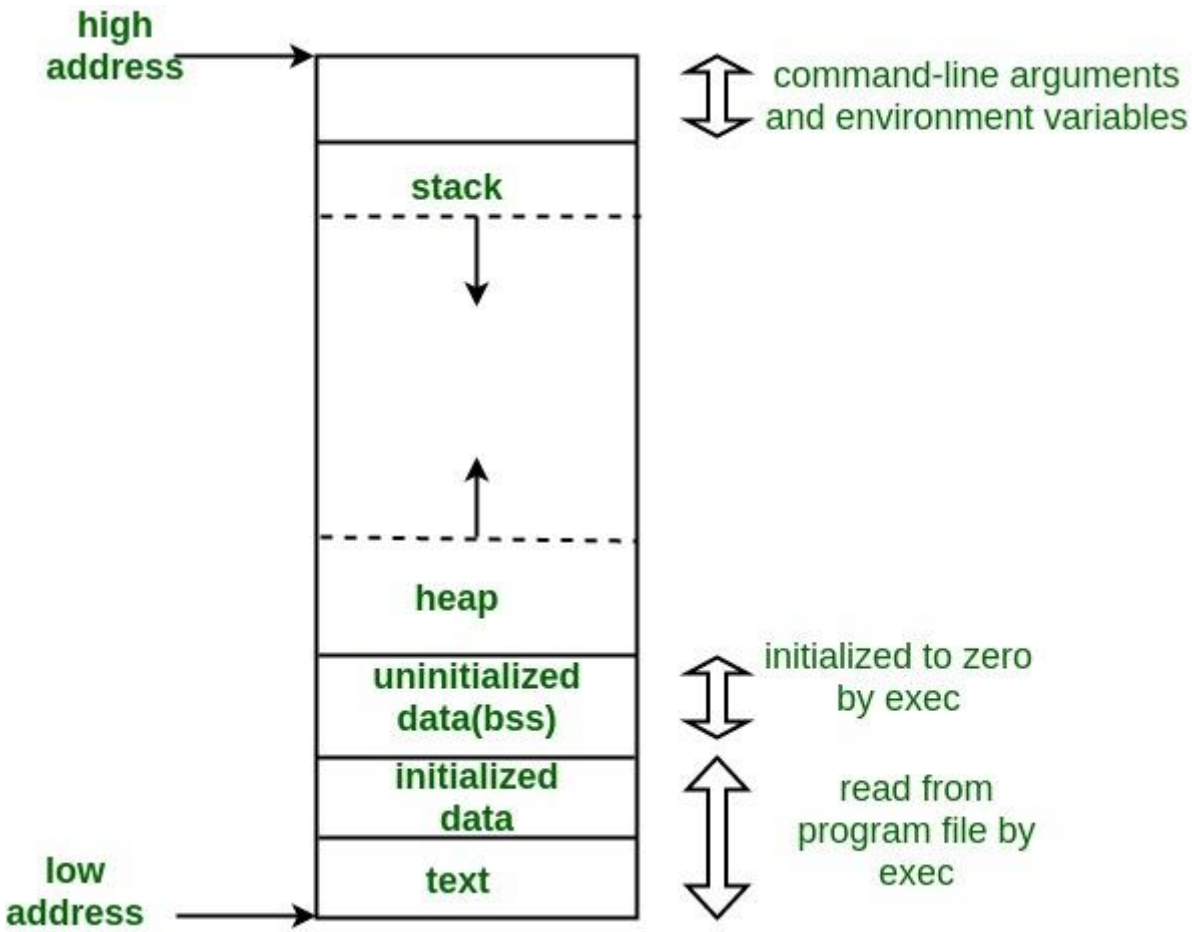
New Way to Allocate Memory: `new` and `delete` (I)

- C++ has better ways to allocate memory.

C	`malloc()`	`free()`
C++	`new`	`delete,` `delete[]`

```
int *x;
float *array;
x = new int;  // allocating a int variable
array = new float[100]; // allocate a float array
delete x;  // free the memory for x
delete [] array; // free the memory for an array

char *c = new char(64);   // initialization
double *d = NULL;         // nullptr(c++11)
delete d;  // deleting a NULL pointer is ok.
```



https://www.geeksforgeeks.org/memory-layout-of-c-program/

New Way to Allocate Memory: `new` and `delete` (II)

- `new` returns NULL if no memory is available.
- Do not combine `malloc` with `delete`, `new` with `free`.
- Advantages of `new` and `delete`:
  - Simplicity.
  - Initialization capability.
  - Compatibility with other aspect of C++. (More on this later)

Summary of Extensions in C++ (I)

- C++ tries to simplify C
  - // comments
  - `cin / cout`
  - `new / delete`
  - Reference
- C++ aspires to make C less error prone
  - // Comments
  - Strict function prototypes
  - `cin / cout`
  - Inline functions instead of macros
  - `new / delete`

Summary of Extensions in C++ (II)

- C++ reduces the role of preprocessor.
  - `const`'s instead of `#define`'s
  - Inline functions instead of macros
- C++ adds new functionality, more power.
  - Default arguments / function overloading
  - Variable declarations at any points