Functions II

DM2111
C++ Programming

Introduction

Introduction	Break
Problem solving	Array and Strings
Basic elements of C++	Array and Strings
Basic elements of C++	Pointers
Statements	Pointers
Repetition	I/O operations
Functions	Structs
Functions	Others

Agenda

- Variable Scope
- Variable Lifespan
- Function Overloading
- Default Parameters
- Recursive Functions
- Using Multiple Files

Variable Scope refers to where it is accessible

- Global
  - Declared outside of any function
  - Available everywhere
- Local
  - Declared within a block / function
  - Available only within the block / function

```
int global;

void coolFunction()
{
    int local;
    local = global; // allowed
    //more code here
}
global = local; // error: local is not accessible here.
```

Global variable

- Name conflict with local variable is resolved with the scope resolution operator (::)
  - If it is used before it is declared, it needs to be declared extern (similar to function prototype)

Variable Scope : Global Scope examples

```
int secret = 1234;

void fn (void)
{
    cout << secret;  // 1234
}

void main (void)
{
    cout << secret;  // 1234
}
```

This is declared as global

In this example, both `fn()` & `main()` can access this variable

```
int secret = 1234;

void main (void)
{
    int secret = 5678;

    cout << secret;    // 5678
    cout << ::secret;  // 1234
}
```

With the scope call here, we tell compiler to use the global version, not local version of this variable name

Extern

Useful when programs are split into separate files.

Know the difference between declaration and definition

- Declaration makes the entity known
- Definition creates the entity

file01.cpp

```
// This is a declaration
extern int chickens;

~~int chickens;~~ // can’t do this

++chickens;
```

file02.cpp

```
// This is a definition
int chickens;

~~int chickens;~~ // can’t do this

++chickens;
```

Global variable - Try not to use them

Avoid global variables as far as possible

- Global variable is usually a lazy workaround
- Poor design
- Variables declared global will stay memory resident for the lifespan of program
- Difficult to debug

Local Variable exists only within its block

```
void main (void)
{
    int ant= 4;
    {
        int bear = ant;     // allowed
    }
    {
        int cow = bear;     // not allowed
        cow = ant;          // allowed
        {
            int deer = cow; // allowed
            deer = ant;     // allowed
        }
        int emu = cow;      // allowed
        emu = deer;         // not allowed
    }
    int fox = deer;         // not allowed
    fox = emu;              // not allowed
}
```

Static variables

- Global variables remain allocated throughout the lifetime of the program
- Automatic variables are allocated at block entry and de-allocated at block exit
- Static variables remain allocated throughout the lifetime of the program

```
int fn (void)
{
    int val = 5;
    return val++;
}

void main (void)
{
    cout << fn();   // 5
    cout << fn();   // 5
}
```

```
int fn (void)
{
    static int val = 5;
    return val++;
}

void main (void)
{
    cout << fn();   // 5
    cout << fn();   // 6
}
```

Variable Lifespan

Never return reference or pointer to local objects

```
int& multiply (int& rabbit)
{
    rabbit *= 2;
    return rabbit;
}

int& add (int& rabbit)
{
    int kit = rabbit + 1;
    return kit; // kit only exists in this function!
}
```

Function Overloading

Functions that perform the same general actions, but apply to different parameter types

```
void printInt(int i)
{
     cout << "int has a value of " << i;
}
void printFloat(float f)
{
    cout << "float has a value of " << f;
}
void printChar(char c)
{
    cout << "char has a value of " << c;
}

printInt(2);
printFloat(2.0f);
printChar('2');
```

```
void print(int i)
{
     cout << "int has a value of " << i;
}
void print(float f)
{
    cout << "float has a value of " << f;
}
void print(char c)
{
    cout << "char has a value of " << c;
}

print(2);
print(2.0f);
print('2');
```

Function Overloading with different number of parameters

```
void kahWei();
void kahWei(long);
void kahWei(long, int);
```

Function Overloading with different data types

```
void jax(int);
void jax(char);
void jax(string);
```

Function Overloading does NOT work for return types

```
void gerald();
short gerald();  // NOPE!!!
char gerald();   // NOPE!!!
```

Function Overloading

```
void glenn();
void glenn(void);          // NOPE!!!
short glenn();             // NOPE!!!
void glenn(int);
void glenn(char);
void glenn(unsigned int);

void glenn(int, int);
void glenn(float, int);
void glenn(int, float);
int glenn(int, int);       // NOPE!!!
```

Function matching

```
void timLin(float);
void timLin(double);
void timLin(long double);


timLin(1);        // No match!
timLin(1u);       // No match!
timLin(1ul);      // No match!
timLin(1.0);      // timLin(double)
timLin(1.0f);     // timLin(float)
timLin(1.0l);     // timLin(long double)
timLin(‘1’);      // No match!
```

Default Parameters

- The number of parameters in a function call must match the number of parameters of the function
- Default Parameters allow you to set the most common value for a parameter

```cpp
int increment(int n, int i)
{
    return n + i;
}

void main (void)
{
    increment(4, 2); // 6
}
```

Example of a default parameter value

```cpp
int increment(int n, int i = 1)
{
    return n + i;
}

void main (void)
{
    increment(4, 2); // 6
    increment(4);    // 5
}
```

Default Parameters are defined the declaration

- If prototype is used, default parameters are defined in the prototype only

```cpp
int increment (int n, int i = 1)
{
    return n + i;
}

void main (void)
{
    increment(4, 2); // 6
    increment(4);    // 5
}
```

```cpp
int increment (int n, int i = 1);

int increment (int n, int i)
{
    return n + i;
}

void main (void)
{
    increment(4, 2); // 6
    increment(4);    // 5
}
```

Default Parameters

- Once a default parameter starts, all subsequent parameters must have defaults
- In a function call, if a default value is used, all subsequent default values must be used

```
void fn (int a, int b = 2, int c, int d = 4);              ✗

void fn (int a, int b = 2, int c = 3, int d = 4);          ✓
```

```
void fn (int a, int b = 2, int c = 3, int d = 4);

fn (10, , 30, 40);     ✗

fn (10, 20);     // a = 10, b = 20, c = 3, d = 4           ✓
```

Recursive Functions

To understand recursion, you must first understand recursion.

Recursive Functions call themselves

```
n! = 1 x 2 x 3 x ... x (n-2) x (n-1) x n
```

```
int factorial (int num)
{
    int result = 1;

    for (int i = 1; i <= num; ++i)
    {
        result *= i;
    }
    return result;
}
```

Recursive Functions call themselves

n! = 1 x 2 x 3 x ... x (n-2) x (n-1) x n

```
int factorial (int num)
{
    if (num <= 1)
    {
        return 1;
    }
    else
    {
        return num * factorial(num – 1);
    }
}
```

Recursion functions must have a terminating condition
Otherwise your computer will explode

Hey girl,

I want our relationship to be like poor recursion, so that it never terminates.

Using Multiple Files

- Good programming practice
  - Makes code cleaner and easier to maintain
  - Different people can work on different parts of the code at the same time
  - E.g. working with a concurrent versioning system (svn)
- Each functionality group should have its own .cpp and .h
- Other parts of the code that uses those functions need to include the .h file

Using Multiple Files

- Function declaration tells the compiler how the function looks like (name, parameters and return types)

```
int sum (int x, int y);
```

- Function definition gives the actual code of the function

```
int sum (int x, int y)
{
    return x + y;
}
```

Using Multiple Files

- Suppose function A() calls function B(); when function A() is compiled, the compiler needs to know the declaration of function B()
- Function B() (the definition) is compiled in a similar manner
- All compiled functions will reside in some .obj file
- During linking, all relevant files are combined to form the final .exe

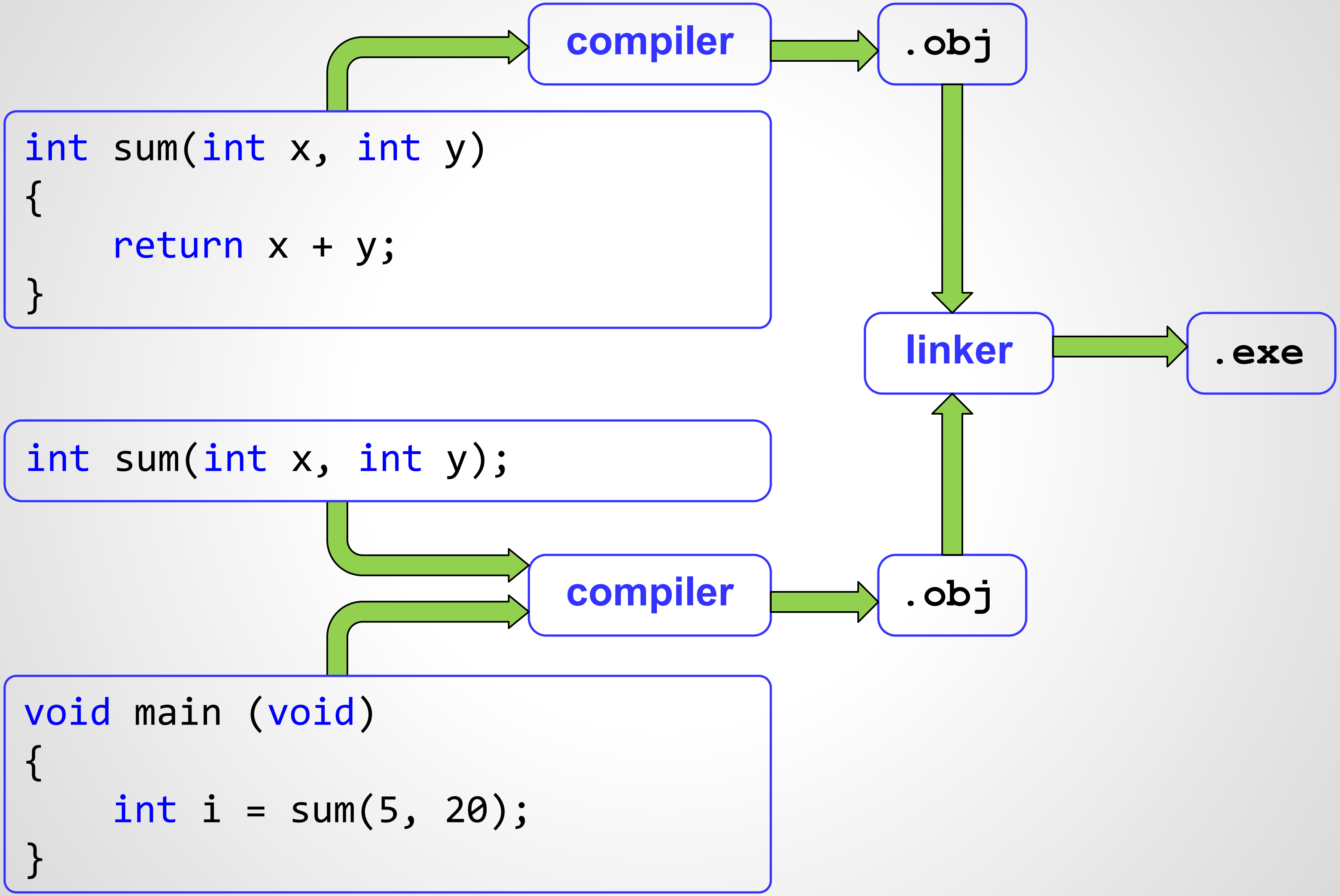
Using Multiple Files

```cpp
int sum(int x, int y)
{
    return x + y;
}
```

```cpp
int sum(int x, int y);
```

```cpp
void main (void)
{
    int i = sum(5, 20);
}
```

Using Multiple Files



```
int sum(int x, int y)
{
    return x + y;
}
```

```
int sum(int x, int y);
```

```
void main (void)
{
    int i = sum(5, 20);
}
```

compiler

.obj

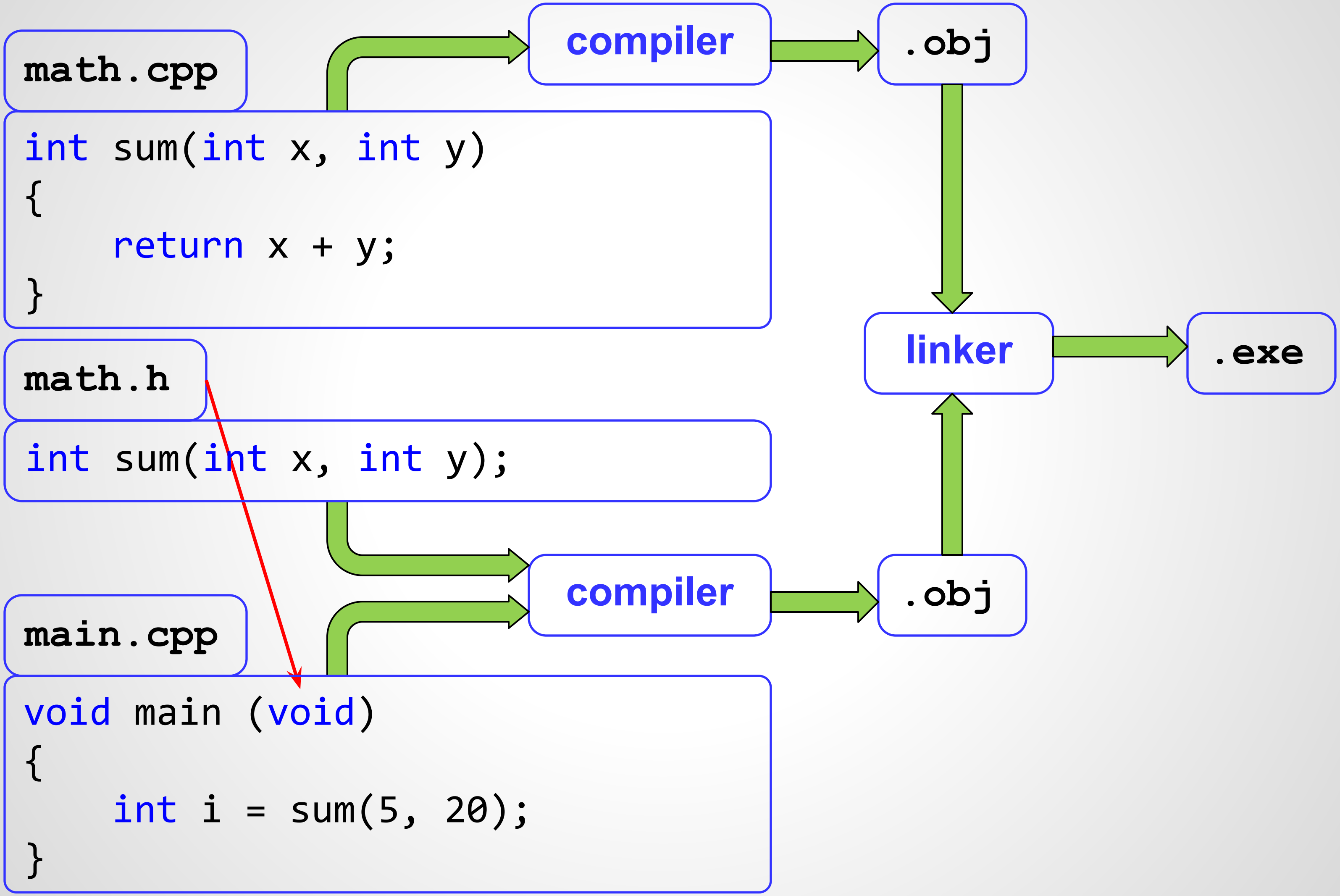
compiler

.obj

linker

.exe

Using Multiple Files



math.cpp

```
int sum(int x, int y)
{
    return x + y;
}
```

math.h

```
int sum(int x, int y);
```

main.cpp

```
void main (void)
{
    int i = sum(5, 20);
}
```

Header guards prevent multiple inclusion

math.h

```
#ifndef _MATH_H
#define _MATH_H
int sum(int x, int y);

#endif // _MATH_H
```

main.cpp

```
#include "math.h"
void main (void)
{
    int i = sum(5, 20);
}
```