2 Files and Functions

Introduction – Files, functions and namespaces

- Contents of this chapter

  - **Encapsulating algorithms** – functions

  - **Splitting your code into modules** – working with multiple files

  - **Decoupling symbols in modules** – namespaces

  - **Working with existing modules** – The Standard Library

Structured programming – Functions

- Functions group statements into logical units
  - Functions encapsulate algorithms

- Declaration

```
TYPE function_name(TYPE arg1, TYPE arg2, …, TYPE argN) ;
```

- Definition:

```
TYPE function_name(TYPE arg1, TYPE arg2, …, TYPE argN) {
   // body
   statements ;
   return arg ;
}
```

- Ability to declare function separate from definition important
  - Allows to separate implementation and interface
  - But also solves certain otherwise intractable problems

Forward declaration of functions

- Example of trouble using function definitions only

```
int g() {
  f() ; // g calls f – ERROR, f not known yet
}

int f() {
  g() ; // f calls g – OK g is defined
}
```

  – Reversing order of definition doesn’t solve problem
  – But **forward declaration** does solve it:

```
int f(int x) ;

int g() {
  f(x*2) ; // g calls f – OK f declared now
}

int f(int x) {
  g() ; // f calls g – OK g defined by now
}
```

Functions – recursion

- Recursive function calls are explicitly allowed in C++
  - Example

```
int factorial(int number) {

   if (number<=1) {
      return number ;
   }

   return number*factorial(number-1) ;
}
```

  - NB: Above example works only in pass-by-value implementation

- Attractive solution for inherently recursive algorithms
  - Recursive (directory) tree searches, etc…

Function arguments

- Function ***input and return*** arguments are ***both optional***
  - Function with no input arguments: `TYPE function() ;`
  - Function with no return argument: `void function(TYPE arg,…) ;`
  - Function with neither: `void function() ;`
- Pseudo type **`void`** is used as place holder when no argument is returned
- Returning a value
  - If a function is declared to return a value, a value must be returned using the '`return <value>`' statement
  - The return statement may occur anywhere in the function body, but every execution path must end in a return statement

```
int func() {
   if (special_condition) {
      return -1 ;
   }
   return 0;
}
```

```
void func() {
   if (special_condition) {
      return ;
   }
   return ; // optional
}
```

  - Void functions may terminate early using '`return ;`'

Function arguments – values

- By default all functions arguments are passed by value
  - Function is passed **copies** of input arguments

```
void swap(int a, int b) ;

int main() {
   int a=3, b=5 ;
   swap(a,b) ;
   cout << "a=" << a << ", b=" << b << endl ;
}

void swap(int a, int b) {
   int tmp ;
   tmp = a ;
   a = b ;
   b = tmp ;
}
// outputs: "a=3, b=5"
```

a and b in swap() are **copies** of a and b in main()

  - Allows function to freely modify inputs without consequences
  - Note: potentially expensive, because passing large objects (arrays) by value is expensive!

Function arguments – references

- You can change this behavior by passing **references** as input arguments

```
void swap(int& a, int& b) ;

int main() {
   int a=3, b=5 ;
   swap(a,b) ;
   cout << "a=" << a << ", b=" << b << endl ;
}

void swap(int& a, int& b) {
   int tmp ;
   tmp = a ;
   a = b ;
   b = tmp ;
}
// outputs: "a=5, b=3"
```

a and b in swap() are **references to** original a and b in main(). Any operation affects originals

  - Passing by reference is inexpensive, regardless of size of object
  - But allows functions to modify input arguments which may have potentially further consequences

Function arguments – const references

- Functions with 'const references' take references but promise not to change the object

```
void swap(const int& a, const int& b) {
   int tmp ;
   tmp = a ;   // OK – does not modify a
   a = b ;     // COMPILER ERROR – Not allowed
   b = tmp ;   // COMPILER ERROR – Not allowed
}
```

- Use const references instead of 'pass-by-value' when you are dealing with large objects that will not be changed
  - Low overhead (no copying of large objects)
  - Input value remains unchanged (thanks to const promise)

Function arguments – pointers

- You can of course also pass pointers as arguments

```
void swap(int* a, int* b) ;

int main() {
  int a=3, b=5 ;
  swap(&a,&b) ;
  cout << "a=" << a << ", b=" << b << endl ;
}

void swap(int* a, int* b) {
  int tmp ;
  tmp = *a ;
  *a = *b ;
  *b = tmp ;
}

// outputs: "a=5, b=3"
```

a and b in swap() are **pointers to** original a and b in main(). Any operation affects originals

  – Syntax more cumbersome, use references when you can, pointers only when you have to

Function arguments – references to pointers

- Passing a pointer by reference allows function to modify pointers
  - Often used to pass multiple pointer arguments to calling function

```
bool allocate(int*& a, int*& b) ;

int main() {
  int* a(0), *b(0) ;
  bool ok = allocate(a,b) ;
  cout << a << " " << b << endl ;
  // prints 0x4856735 0x4927847
  delete[] a;  delete[] b;
}

bool allocate(int*& a, int*& b) {
  a = new int[100] ;
  b = new int[100*100] ;
  return true ;
}
```

  - NB: reverse is not allowed – you can't make a pointer to a reference as a reference is not (necessarily) an object in memory

Function arguments – arrays

- Example of passing a 1D array as an argument

```
void square(int len, int array[]) ;

int main() {
   int array[3] = { 0, 1, 2 } ;
   square( sizeof(array)/sizeof(int), array) ;
   return 0 ;
}

void square (int len, int array[]) {
   while(--len>=0) {
      array[len] *= array[len] ;
   }
}
```

Remark:
Code exploits that len is a copy
Note prefix decrement use
*Note use of *= operator*

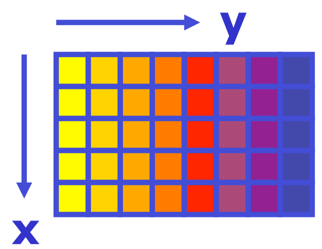
- Remember basic rule: array = pointer
  - Writing '`int* array`' is equivalent to '`int array[]`'
  - Arrays always passed 'by pointer'
  - Need to pass length of array as separate argument

Function arguments – multi-dimensional arrays

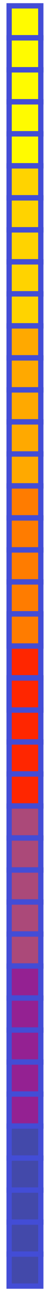
- Multi-dimensional array arguments more complicated
  - Reason: interpretation of array memory depends on dimension sizes (unlike 1D array)
- Must specify all dimensions except 1st one in declaration
  - Example

```
// arg of f declared to be N x 10 array
void f(int p[][10]) ;

// Pass 5 x 10 array
int a[5][10] ;
f(a) ;

void f(int p[][10]) {
  // inside f use 2-D array as usual
     … p[i][j]…
}
```

Logical layout



Memory layout



Function arguments – char[] (strings)

- Since char* strings are zero terminated by convention, no separate length argument needs to be passed
  - Example

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

int main(){
   const char* foo = “Hello World” ;
   print(foo) ;
   return 0 ;
}

void print (const char* str) {
   const char* ptr = str ;
   while (*ptr != 0) {
       cout << *ptr << endl ;
       ptr++ ;
   }
}
```

Function arguments – main() and the command line

- If you want to access command line arguments you can declare `main()` as follows

Array of (**char***)

```
int main(int argc, const char* argv[]) {
   int i ;
   for (i=0 ; i<argc ; i++) {
      // argv[i] is 'char *'
      cout << "arg #" << i << " = " << argv[i] << endl ;
   }
}
```

  – Second argument is array of pointers

- Output of example program

```
unix> cc -o foo foo.cc
unix> foo Hello World
arg #0 = foo
arg #1 = Hello
arg #2 = World
```

Functions – default arguments

- Often algorithms have optional parameters with default values
  - How to deal with these in your programs?
- Simple: in C++ functions, arguments can have default values

```
void f(double x = 5.0) ;
void g(double x, double y=3.0) ;
const int defval=3 ;
void h(int i=defval) ;

int main() {
  double x(0.) ;

  f() ;      // calls f(5.0) ;
  g(x) ;     // calls g(x,3.0) ;
  g(x,5.0) ; // calls g(x,5.0) ;
  h() ;      // calls h(3) ;
}
```

- Rules for arguments with default values
  - Default values can be literals, constants, `enum`erations or `static`s
  - Positional rule: all arguments without default values must appear to the left of all arguments with default values

Function overloading

- Often algorithms have different implementations with the same functionality

```
int minimum3_int(int a, int b, int c) {
  return (a < b ? ( a < c ? a : c ) : ( b < c ? b : c) ) ;
}

float minimum3_float(float a, float b, float c) {
  return (a < b ? ( a < c ? a : c ) : ( b < c ? b : c) ) ;
}

int main() {
  int a=3,b=5,c=1 ;
  float x=4.5,y=1.2,z=-3.0 ;

  int d = minimum3_int(a,b,c) ;
  float w = minimum3_float(x,y,z) ;
}
```

  - The `minimum3` algorithm would be easier to use if both implementations had the same name and the compiler would automatically select the proper implementation with each use

Function overloading

- C++ function overloading does exactly that
  - Reimplementation of example with function overloading

```
int minimum3(int a, int b, int c) {
  return (a < b ? ( a < c ? a : c )
                : ( b < c ? b : c) ) ;
}

float minimum3 (float a, float b, float c) {
  return (a < b ? ( a < c ? a : c )
                : ( b < c ? b : c) ) ;
}

int main() {
  int a=3,b=5,c=1 ;
  float x=4.5,y=1.2,z=-3.0 ;

  int d = minimum3(a,b,c) ;
  float w = minimum3(x,y,z) ;
}
```

Overloaded functions have same name, but different signature (list of arguments)

Code calls same function name twice. Compiler selects appropriate overloaded function based on argument list

Function overloading resolution

- How does the compiler match a list of arguments to an implementation of an overloaded function? It tries

Rank	Method	Example
1	Exact Match	
2	Trivial argument conversion	`int → int&`
3	Argument Promotion	`float → double`
4	Standard argument conversion	`int→bool, double→float`
5	User defined argument conversion	(we'll cover this later)

  – Example

```
void func(int i) ;
void func(double d) ;

int main() {
  int i ;
  float f ;

  func(i) ; // Exact Match
  func(f) ; // Promotion to double
}
```

Function overloading – some fine points

- Functions can not be overloaded on return type

```
int function(int x) ;
float function(int x) ; // ERROR – only return type is different
```

- A call to an overloaded function is only legal if there is exactly one way to match that call to an implementation

```
void func(bool i) ;
void func(double d) ;

int main() {
  bool b ;
  int i ;
  float f ;

  func(b) ;  // Exact match
  func(f) ; // Unique Promotion to double
  func(i) ;  // Ambiguous Std Conversion (int→bool or int→double)
}
```

  – Its gets more complicated if you have >1 arguments

Pointers to functions

- You can create pointers to functions too!
  - Declaration

```
TYPE (*pfname)(TYPE arg1, TYPE arg2,…) ;
```

  - Example

```
double square(double x) {
   return x*x ;
}

int main() {
   double (*funcptr)(double i) ;     // funcptr is function ptr
   funcptr = &square ;

   cout << square(5.0) << endl ;     // Direct call
   cout << (*funcptr)(5.0) << endl ; // Call via pointer
}
```

  - Allows to pass function as function argument, e.g. to be used as callback function

Pointers to functions – example use

- Example of pointer to function – call back function

```
void sqrtArray(double x[], int len, void (*handler)(double x)) {
  int i ;
  for (i=0 ; i<len ; i++) {
    if (x[i]<0) {
      handler(x[i]) ; // call handler function if x<0
    } else {
      cout << "sqrt(" << x[i] << ") = " << sqrt(x[i]) << endl ;
    }
  }
}

void errorHandler(double x) {
  cout << "something is wrong with input value " << x << endl ;
}

int main() {
  double x[5] = { 0, 1, 2, -3, -4 } ;
  sqrtArray(x, 5,  &errorHandler) ;
}
```

Organizing your code into modules

- For all but the most trivial programs it is not convenient to keep all C++ source code in a single file
  – Split source code into multiple files

- Module: unit of source code offered to the compiler
  – Usually module = file

- How to split your code into files and modules
  1. Group functions with related functionality into a single file
     - Follow guide line 'strong cohesion', 'loose coupling'
     - Example: a collection of char* string manipulation functions go together in a single module
  2. Separate declaration and definition in separate files
     - Declaration part to be used by other modules that interact with given module
     - Definition part only offered once to compiler for compilation

Typical layout of a module

- Declarations file

```
// capitalize.hh
void convertUpper(char* str) ;
void convertLower(char* str) ;
```

Declarations

- Definitions file

```
// capitalize.cc
#include “capitalize.hh”
void convertUpper(char* ptr) {
   while(*ptr) {
      if (*ptr>=‘a’&&*ptr<=‘z’) *ptr -= ‘a’-’A’ ;
      ptr++ ;
   }
}
void convertLower(char* ptr) {
   while(*ptr) {
      if (*ptr>=‘A’&&*ptr<=‘Z’) *ptr += ‘a’-’A’ ;
      ptr++ ;
   }
}
```

Definitions

Using the preprocessor to include declarations

- The C++ preprocessor `#include` directive can be used to include declarations from an external module

```
// demo.cc

#include "capitalize.hh"

int main(int argc, const char* argv[]) {
    if (argc!=2) return 0 ;
    convertUpper(argv[1]) ;
    cout << argv[1] << endl ;
}
```

- But watch out for multiple inclusion of same source file
  - Multiple inclusion can have unwanted effects or lead to errors
  - Preferred solution: add safeguard in `.hh` file that gracefully handles multiple inclusions
    - rather than rely on cumbersome bookkeeping by module programming

Safeguarding against multiple inclusion

- Automatic safeguard against multiple inclusion
  - Use preprocessor conditional inclusion feature

    ```
    #ifndef NAME
    (#else)
    #endif
    ```

  - NAME can be defined with `#define`

- Application in `capitalize.hh` example
  - If already included, CAPITALIZE_HH is set and future inclusion will be blank

```
// capitalize.hh
#ifndef CAPITALIZE_HH
#define CAPITALIZE_HH

void convertUpper(char* str) ;
void convertLower(char* str) ;

#endif
```

Namespaces

- Single global namespace often bad idea
  - Possibility for conflict: someone else (or even you inadvertently) may have used the name want you use in your new piece of code elsewhere → Linking and runtime errors may result
  - Solution: make separate 'namespaces' for unrelated modules of code
- The namespace feature in C++ allows you to explicitly control the scope of your symbols
  - Syntax:

```
namespace name {

  int global = 0 ;

  void func() {
    // code
    cout << global << endl ;
  }

}
```

Code can access symbols inside same namespace without further qualifications

Namespaces

- But code outside namespace must explicitly use scope operator with namespace name to resolve symbol

```
namespace foo {

  int global = 0 ;

  void func() {
     // code
     cout << global << endl ;
  }

}

void bar() {
   cout << foo::global << endl ;

   foo::func() ;
}
```

← Namespace applies to functions too!

Namespace rules

- Namespace declaration must occur at the global level

```
void function foo() {
   namespace bar {      ERROR!
      statements ;
   }
}
```

- Namespaces are extensible

```
namespace foo {
  int bar = 0 ;
}

// other code

namespace foo {   Legal
  int foobar = 0 ;
}
```

Namespace rules

- Namespaces can nest

```
namespace foo {
  int zap = 0 ;

   namespace bar {        Legal
     int foobar = 0 ;
   }

}

int main() {
   cout << foo::zap << endl ;
   cout << foo::bar::foobar << endl ;
}
```

Recursively use :: operator to resolve nested namespaces

Namespace rules

- Namespaces can be unnamed!
  - Primary purpose: to avoid ‘leakage’ of private global symbols from module of code

```
namespace {
  int bar = 0 ;
}

void func() {
  cout << bar << endl ;
}
```

Code in same module ***outside*** unnamed namespace can access symbols ***inside*** unnamed namespace

Namespaces and the Standard Library

- All symbols in the Standard library are wrapped in the namespace ‘std’
- The ‘Hello world’ program revisited:

```
// my first program in C++
#include <iostream>

int main () {
  std::cout << "Hello World!“ << std::endl;
  return 0;
}
```

Using namespaces conveniently

- It is possible to import symbols from a given namespace into the current scope
  - To avoid excessive typing and confusing due to repeated lengthy notation

```
// my first program in C++
#include <iostream>
using std::cout ;
using std::endl ;

int main () {
  cout << "Hello World!“ << endl;
  return 0;
}
```

Import selected symbols into global namespace

Imported symbols can now be used without qualification in this module

  - Can also import symbols in a local scope. In that case import valid only inside local scope

Using namespaces conveniently

- You can also import the symbol contents of an entire namespace

```
// my first program in C++
#include <iostream>
using namespace std ;

int main () {
  cout << "Hello World!“ << endl;
  return 0;
}
```

- Style tip: If possible only import symbols you need

Modules and namespaces

- Namespaces enhance encapsulation of modules
  - Improved capitalize module

```
// capitalize.hh
#ifndef CAPITALIZE_HH
#define CAPITALIZE_HH
namespace capitalize {
void convertUpper(char* str) ;
void convertLower(char* str) ;
}
#endif
```

```
// capitalize.cc
#include “capitalize.hh”
namespace capitalize {
  void convertUpper(char* ptr) {
     while(*ptr) {
        if (*ptr>=‘a’&&*ptr<=‘z’) *ptr -= ‘a’-’A’ ;
        ptr++ ;
     }
  }
  void convertLower(char* ptr) {
     while(*ptr) {
        if (*ptr>=‘A’&&*ptr<=‘Z’) *ptr += ‘a’-’A’ ;
        ptr++ ;
     }
  }
}
```

The standard library as example

- Each C++ compiler comes with a standard suite of libraries that provide additional functionality
  - `<math>` -- Math routines `sin(),cos(),exp(),pow(),…`
  - `<stdlib>` -- Standard utilities `strlen(),strcat(),…`
  - `<stdio>` -- File manipulation utilities `open(),write(),close(),…`

- Nice example of modularity and use of namespaces
  - All Standard Library routines are contained in `namespace std`

Compiling & linking code in multiple modules

- Compiling & linking code in a single module
  - `g++ -c demo.cc`
    - Converts demo.cc C++ code into demo.o (machine code)
  - `g++ -o demo demo.o`
    - Links demo.o with Standard Library code and makes standalone executable code
  - Alternatively, '`g++ -o demo demo.cc`' does all in one step

- Compiling & linking code in multiple modules
  - `g++ -c module1.cc`
  - `g++ -c module2.cc`
  - `g++ -c module3.cc`
  - `g++ -o demo module1.o module2.o module3.o`
    - Link module1,2,3 to each other and the Standard Library code

Intermezzo – Concept of ‘object libraries’

- Operating systems usually also support concept ‘object libraries’
  - A mechanism to store multiple compiled modules (.o files) into a single library file
  - Simplifies working with very large number of modules
  - Example: all Standard Library modules are often grouped together into a single ‘object library file’

- Creating an object library file from object modules
  - Unix: ‘`ar q libLibrary.a module1.o module2.o …`’
- Linking with a library of object modules
  - Unix: ‘`g++ -o demo demo.cc –L. –lLibrary`’
    - Above syntax looks for library name `libLibrary.a` in ‘standard locations’
    - To add directory to library search path, specify –L<path> in g++ command line. Typically the present directory is not by default in the search path; in that case a ‘-L.’ needs to be added.

Debugging tips – Crashes etc…

- Your program crashes – How do you analyze this
  - Recompile your program with the '-g' flag
    (i.e. `g++ -g -o blah blah.c`).
    - This will preserve source code line-number information in the executable
  - Rerun your program in the debugger:

```
unix> gdb blah
gdb> run <command line args for blah, if any, go here>
```

    (wait for crash)

```
gdb> where
```

    (shows line of code where crash occurred)

```
gdb> quit
```

    (exits the debugger)

Debugging tips – Memory leaks, corruption etc

- You want to check that no memory leaks occur, no memory corruption occurs (e.g. writing beyond boundaries of arrays etc...)
  - Recompile your program with the '-g' flag (i.e. `g++ -g –o blah blah.c`).
    - This will preserve source code line-number information in the executable
  - Rerun your problem with valgrind
    `unix> valgrind blah`
  - If memory corruption occurs, ERRORs will be printed in report (along with line numbers in code)
  - If memory leakage occurs, only total amount leaked is shown. To show report with details (where memory was allocated that was not deleted), rerun
    `unix> valgrind --leak-check=full blah`