CS3215

C++ briefing

Outline:

1. Introduction
2. C++ language features
3. C++ program organization

C++ versus Java

- Java is safer and simpler than C++
- C++ is faster, more powerful than Java:
  - lack of run-time checks in C++
  - direct use of pointers
  - dynamically allocated memory managed by the programmer (no garbage collection)
  - bit operations
  - static and dynamic binding (virtual functions)

Part 1:
C++ language features

C++ program elements

- classes and class instances – objects
  - data members
  - member functions
  - public/protected/private
- unattached functions
  - int max (int i, int j) { …}
  - function prototype: int max (int, int);
- global variables
- constants

Typical program modules

- main module : contains function main ()
  - unattached function main() is called first
- class module:
  - class interfaces in .h files and class implementations in .cpp files
- mixed module :
  - contains declarations, included files and collection of classes and/or unattached function implementations related to a single task
  - mixed modules in SPA : parser, query evaluator

C++ program files

- header files (stack.h) define module interfaces
  - services provided by a module (push(), pop(),…)
- implementation files (stack.cpp)
  - implementation of functions
- communication across files is established via declarations placed in header files:
  - in order to use stack (implemented in stack.cpp) in file f, file f must #include file stack.h

Class IntStack

header file stack.h:

```
class IntStack {
public:
  static const int s_MAX = 100;
  IntStack ();
  void push ();
  int top ();
  void pop ();
  bool empty ();
  bool full ();
private:
  int elem [MAX];
  int top; }
```

implementation file stack.cpp:

```
#include "stack.h"
IntStack::IntStack () {
     code for IntStack here }
void IntStack::push () {
     code for push () here }
   etc. }
```

#include "stack.h" directive causes pre-processor to include the contents of stack.h

Pre-processing

Macros embedded in code:

#include "file-name"

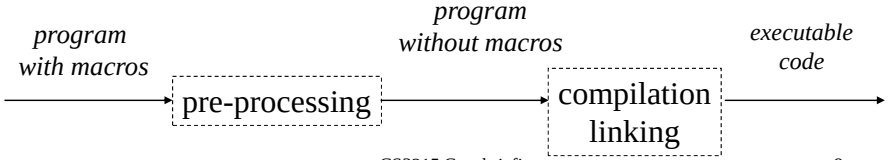
// the contents of the file-name is included here

#include <iostream>, <cassert>, <string>, <list>

ANSI C headers: <assert.h>, <cstring>

#define MAX 100

// occurrences of MAX in program are replaced with 100

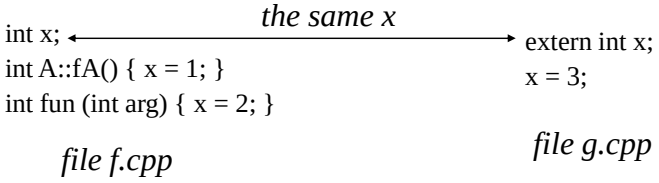


Definition vs. declaration

- there can be ONLY ONE definition of a program element in a given scope:
  - int x = 5; extern int y = 5; const int z = 5; static int t = 10;
    // compiler allocates memory for the above elements
  - implementation of a member or unattached function
- declaration allows you to use the element without knowing it's definition:
  - extern int x; // allows you to use x in a given program file
  - int fun (int arg); // function prototype
  - class A { } // class interface as in the header file
  - forward declaration of a class
    class A; //
    A* pA; // you may now declare a pointer to A

Name scoping rules

- block scope: { int x; }
  - you can use x within the block
- function and class scope
- program file scope
- external linkage: across files



Name spaces

```
namespace scope_1
{  // any constructs here
   int x; // this x is local to scope_1 }
namespace scope_2
{  int x; // this x is different from x in scope_1 }
references to x in different name spaces:
scope_1::x = 1;
scope_2::x = 2;
using namespace scope_1; // all the references to x
   will refer to x in scope_1
```


Static members

- defined for the whole class, not individual object

```
class TNode {
public
   static TNode * s_root_p;
   static int fTNode (); }
TNode *p;
p = TNode:: s_root_p; // use static without object
TNode:: fTNode(); // call to static member function
```

Arrays

- array is not a class
- definition of automatic array: int a [100];
  - a[0], a[1], …, a[99]
  - a[100] – runtime error
- array definitions without specified limits
  - limits are determined by a compiler
  - char text [] = “text here”;
  - int a [] = {1, 2, 3,}
  - foo (int a[]);

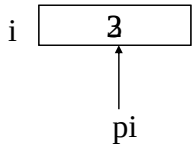
Pointers

- pointer contains address of memory location

```
int *pi; // pointer to the integer
class A { … };
A *pA = new(); // pointer to object of class A
```

- using pointers:

```
int i = 2;
int *pi;
pi = &i; // assigns pi address of i to pi
*pi = 3; // dereference operator *pi
// assigns 3 to location pointed to by pi, value of i now 3
pA -> fA() // invocation of A::fA()
(*pA).fA() // the same as above
```



Pointers, cont.

- pointers to pointers

int i = 2;

int *pi;

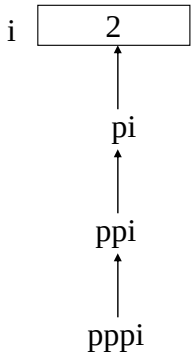
int **ppi, ***pppi;

pi = &i:

ppi = π

pppi = &ppi;

***pppi // the same as value of i



- universal pointer: void* p;

Constant pointers

int i = 7;

const int *p = &i; //can’t change value pointed to by p

*p = 8 // error

int * const q = &i; // can’t change q

q = p; // error

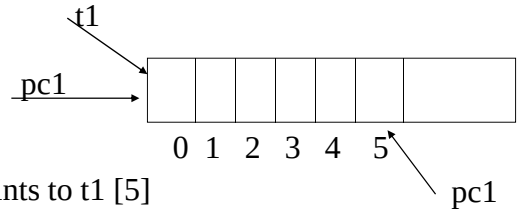
const int * const r = &i;

*r = 8 // error

r = p; // error

Arrays and pointers

- array name is constant pointer whose value points to the first array element




```
char t1 [10], t2 [10];
char *pc1 = t1; //ok
t1 = t2; // error
pc1 = t1 + 5 // pc1 points to t1 [5]
*t1  = *(t1 + 5) // assigns value of t1[5] to t1[0]
```

- you have to watch if array index does not go out of range; if it does – you will get a runtime error

```
char *cc = new char;
cc+1 // legal but points to nowhere – leads to a runtime error
```


Function argument passing

- arguments passed by value:

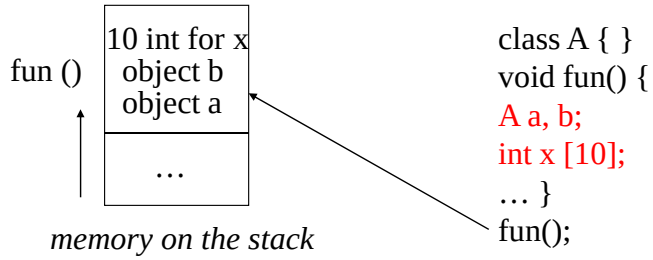
```
void fun (int arg) { arg = 2; }
int j = 1; fun (j); // value of j is still 1
// values of j is assigned to location arg which is used
// in fun(), value of j is not changed by function call
```

- arguments passed by reference:

```
void fun (int  &arg) { arg = 2; }
int j = 1; fun (j); // value of j is 2
// references can be also used as aliases:
int &x = i; // x is an alias for i
```

Memory allocation

- automatic memory allocation on the stack
  - when function is called, the runtime system allocates memory for all the objects defined in that function



  - when function execution is completed, the runtime system automatically frees the memory

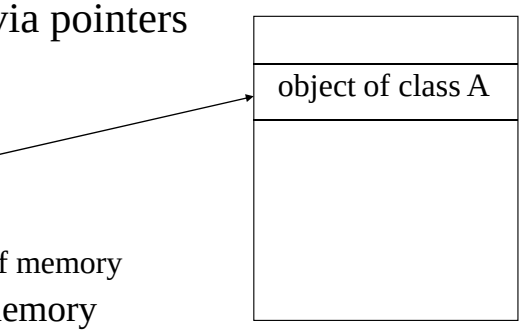
Automatic memory on the stack

```
char *fun() {
   char buf [100];
   get (buf);  // write values to 'buf'
   return buf; }

char *pc = fun();

                //memory for 'buf' has been freed!
```

Free memory allocation on the heap

- allocated with operator new () and freed with delete ()
- values accessed via pointers

```
class A { .. }
void fun () {
   A *pA = new A;
// no automatic deleting of memory
   delete pA; //frees memory
}
```



free memory on the heap

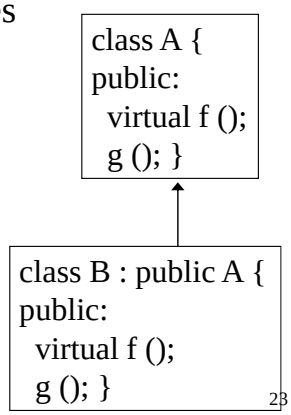
Bitwise operations

- how do you represent Modifies in SPA project?
  - bit vector with n'th bit corresponding to n'th variable
  - n'th bit 1 means that n'th variable is modified in a given procedure
  - n'th bit 0 means that n'th variable is NOT modified in a given procedure
- compact and easy to manipulate representation
- revisit bitwise operations when you plan data structures for Modifies and Uses relationships

Dynamic vs. static binding

- overriding of member functions defined in the parent class by those from derived classes
- dynamic binding for virtual member functions and static biding for non-virtual ones

```
B *pB = new B();
A *pA = pB;
pA -> f(); // dynamic: B::f() is called
 pA -> g(); // static: A::g() is called
```



Part 2:
C++ program organization

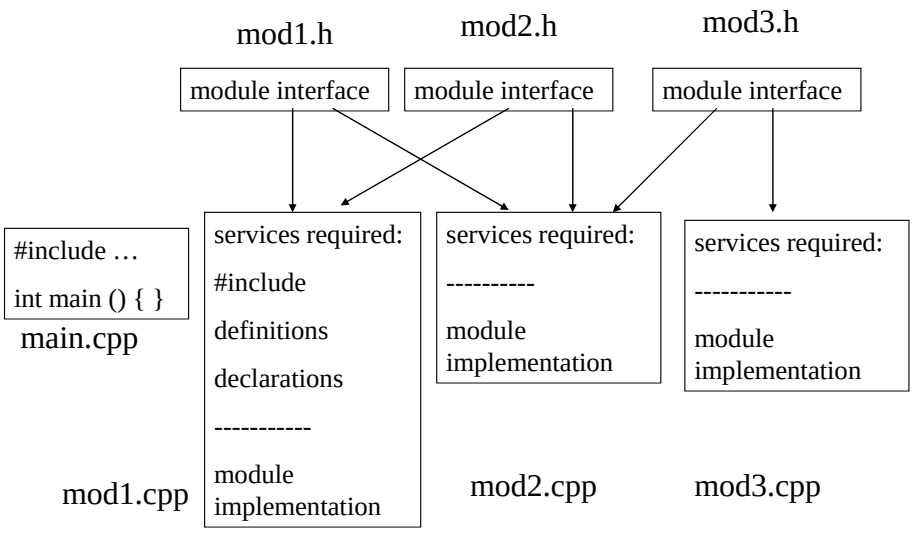
Physical structure of C++ programs

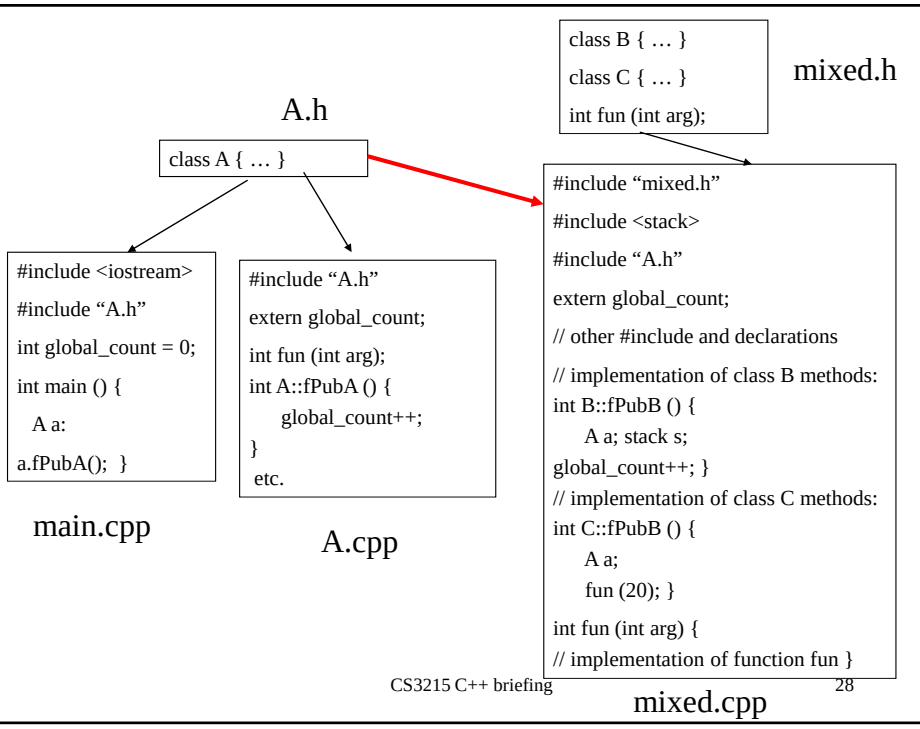
- once you have:
  - understood the problem (SPA)
  - completed architecture design
  - decided upon representation of SPA solution in C++ language
- **you will have to organize C++ program into modules implemented in many files**
- physical organization of program modules into files is a critical success factor in large-scale projects

Typical program modules

- main module file: contains function main ()
  - unattached function main() is called first
- class module files:
  - class interfaces in .h files and class implementations in .cpp files
- mixed module files:
  - contains declarations, included files and collection of classes and/or unattached function implementations related to a single task
  - mixed modules in SPA : parser, query evaluator

Typical program organization





About header files

- you can include many library classes/functions:

  #include <iostream>, #include <string>, #include <cassert>

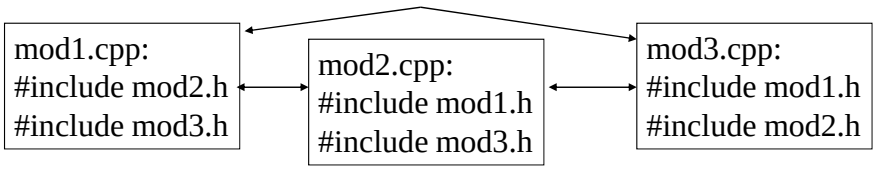
- use “include guards” to avoid multiple inclusions of the same header:

```
#ifndef HEAD_A
#define HEAD_A
#include A.h
/// possibly other included headers here
#endif  //HEAD_A
```

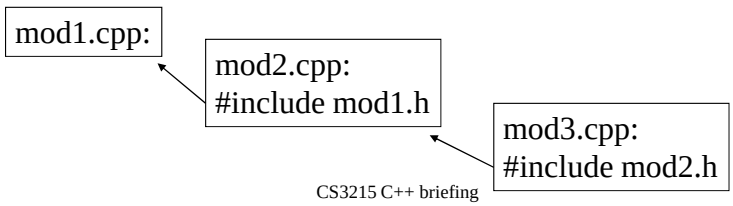
- do not put definitions (only declarations) in headers

Proper use of header files

- chaotic design will result in many dependencies:



- design program in layers to avoid circular inclusions of headers:



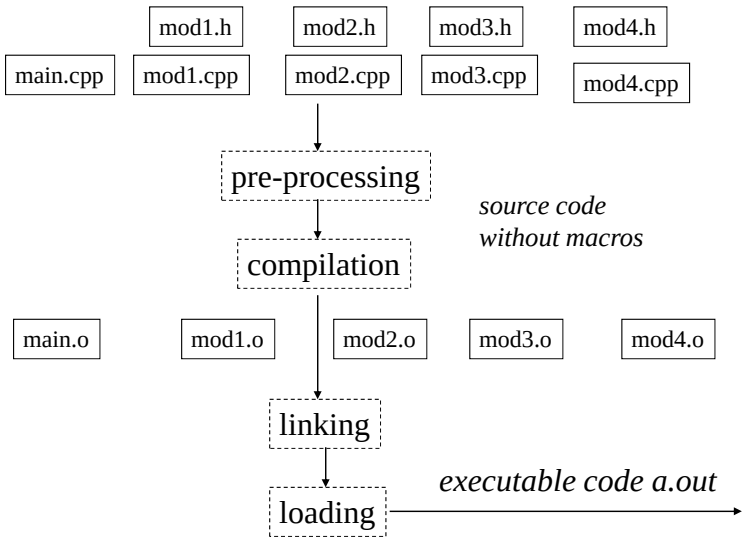
Proper use of header files, cont.

- too many header files complicate the program
  - sometimes it is better to use forward declarations rather than headers:

```
extern int x;
int  fun (int arg);
class A; // you may now declare a pointer to class A
pA* A;
```

- many .cpp files may include the same header
  - design program to minimize the amount of re-compilation required after modification of a header

How is executable code produced?



Adopt naming standards, e.g.:

- file names: query_eval.h, query-eval.cpp

class name: Cfg, TNode, (or TreeNode)

```
class TNode {
public:
   static TNode *s_root_p;
//static data member with prefix s_; pointers with suffix _p
   TNode *getParent ();
private:
   int d_nodeCount; // prefix d_ for data members  }
```

etc.

Documenting module interfaces

```
class TNode {
public:
// CONSTRUCTORS:
   TNode ();
// MODIFIERS (SETTERS):
   void setParent (TNode *parent);
// ACEESSORS (GETTERS):
   TNode *getParent (); }
```

- module query_evaluator.cpp does not need all the interface operations of class TNode (or TNode)
  - use comments to indicate which interface operations of class TNode are used in query_evaluator.cpp
- use comments to explain interface operations

Exceptions and assertions

- Use exceptions to handle abnormal program behavior
  - to handle anticipated errors
- Assertions express your intention as to what a correct program behavior should be:
  - check if program behaves according to program requirements
  - serve as documentation, a bridge between code and requirements
- Read Section 10.6 in Handbook: Error handling, exceptions and assertions in C++

Advanced C++ features

- pointers to functions (unattached and members)
- templates and related classes in STL
- overloading of operations, etc.

- start by using basic language features, as described in this briefing
- study and apply more advanced features only when you need them

--- The End ---