Karlsruhe Institute of Technology

# Programmierparadigmen

Oliver Hummel, IPD

## Topic 4

**C/C++ Repetition** *as a Prerequisite for MPI*

SOFTWARE DESIGN AND QUALITY GROUP
INSTITUTE FOR PROGRAM STRUCTURES AND DATA ORGANIZATION, FACULTY OF INFORMATICS

sdq.ipd.kit.edu

KIT – University of the State of Baden-Wuerttemberg and
National Research Center of the Helmholtz Association

www.kit.edu

# Overview on Today's Lecture

- Content
  - C/C++ Basics
    - Built-in Types
    - Structs, Unions, Classes
    - Pointers, Arrays, References
  - Memory Management
  - Declarations in C

- Learning Goals, participants –
  - refresh the most important basics of the C programming language
    - in order to be able to apply them within MPI
  - are able to read and understand C declarations

# Programming Languages Compared

- Programming in C
  - no object-oriented language constructs; no classes, only structs
  - no (direct) multithreading support until recently (2011)
  - program execution flow is determined by a set of functions
    - starting in function `main()`
- C++
  - adds object-orientation
  - still allows for functions and variables outside classes
  - can still be used for procedural programming without object orientation
- Java
  - strictly object-oriented (but still allows primitive data types)
    - everything resides inside a class (including `static void main()`)
  - source code files are organized along class boundaries
  - explicit multithreading support

# C/C++ Built-in Data Types

- Data types: `void`, `int`, `float`, `double`, `char`, `enum`
- Modifiers: `short`, `long`, `signed`, `unsigned`

➔ *Actual sizes and ranges are platform-dependent!*

- **Arrays**: indicated by `[]`
- Handling of **boolean values**
  - C: `int = 0` for *false*; `int ≠ 0` for *true*
  - C++: `bool`
- Handling of **strings**
  - C: `char[]` terminated with '`\0`',
  - C++: `std::string`
- Handling of **enumerations**
  - C: list of aliases for integer numbers
  - C++: `enum` is still an alias for integers
    - declaration has become simpler keyword `enum` can be omitted

# Structs and Unions

- C/C++ user-defined data type
- Groups variables together
- Members are accessed through the point operator (`.`)

```
struct/union myPoint {int dimX; int dimY; int dimZ;};

int main() {
   struct myPoint p;
   p.dimX = 10;
   p.dimY = 20;
   p.dimZ = 30;
   printf("%d\n", p.dimX)    /* output? */

   return 0;
}
```

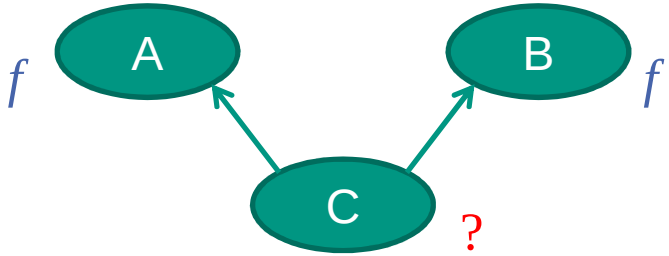
- `union` is similar to `struct` but with storage shared across members
  - may be used to save space, or to give multiple interpretations of the same data
  - *attention: risk of memory errors!*

# Structs vs. Classes

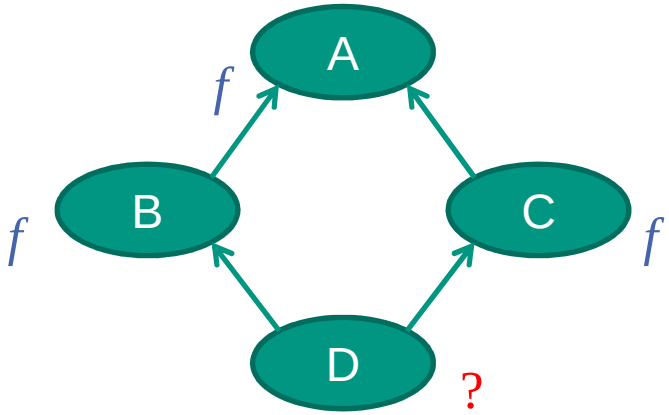
- `struct` in C:
  - only data, no behaviour, no inheritance
- `struct` in C++:
  - may include behaviour (*methods*)
  - even inheritance works
  - only difference to classes is basically the default access to and inheritance of members
    - public for structs
    - private for classes
- `class` in C++/Java:
  - data (*fields*) and behaviour (*methods*)
  - blueprint from which instances (*objects*) can be created at run-time
  - offers inheritance and polymorphism

# Multiple Inheritance

- C++ classes allow for multiple inheritance
  - ➔ *this may cause ambiguities*
  - inheritance of two features with the same name




  - multiple inheritance of the same feature
    - diamond inheritance



# Inheritance of Identical Features

- Two features with equal name: use *scope resolution operator* (`::`)

*like extends in Java*

```
class A {
   public:
      int getData() {
         return 1;
      }
};
```

```
class B {
   public:
      char getData() {
         return 'a';
      }
};
```

```
class C : public A, public B {
   public:
      double getDataC() {
         return 3.14;
      }
};
```

*automatically calls default constructor*

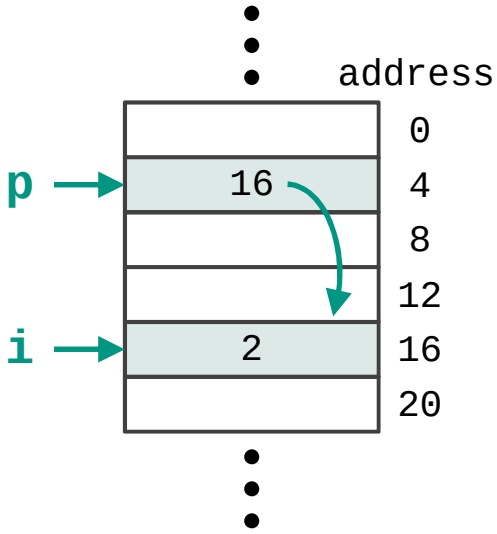
```
int main() {
   C myC;
   int i = myC.A::getData();  /* identify feature using scope
                                 resolution operator */
   char c = myC.B::getData(); /* identify feature using scope
                                 resolution operator */
   double d = myC.getDataC();
   return 0;
}
```

# Pointers

- C/C++ variable that contains an *address* of another variable
- Pointer handling syntax:
  - pointer declaration: asterisk (*)
  - variable address retrieval: reference operator (&)
  - target value retrieval: dereference operator (*)

```
int i, j;
int *p;      /* pointer to int */

int main() {
   i = 2;
   p = &i;  /* p points to address of i */
   j = *p;  /* j is assigned the value of i */
   return 0;
}
```



# Pointers: Fundamental Properties

- Common means to pass parameters to functions
  - avoids copying data structures in spite of „*call by value*“
  - enables data processing in a function without loss of changes upon leaving the function
- Can point to…
  - …any data type, including structs, classes (C++), and `void`
  - …*functions*
  - …other pointers
- Suited for working with arrays
- Can be used to build and manipulate data structures like linked lists

# Pointer Arithmetics

- Pointers can be *incremented* and *decremented*
  - by the (platform-specific) size of their data types
- They can also be *subscripted* like accessing an element of an array

➔ *Attention: risk of data errors due to direct memory access!*

```
char c1, c2;
char *p;              /* pointer to char */

int main() {
   c1 = 'A';
   p = &c1;     /* p is assigned the address of c1 */
   p += 5;      /* p is incremented by 5 * sizeof(char) */
   c2 = p[2];   /* c2 is assigned the value at address p + 2 *
                   sizeof(char)*/
   return 0;
}
```

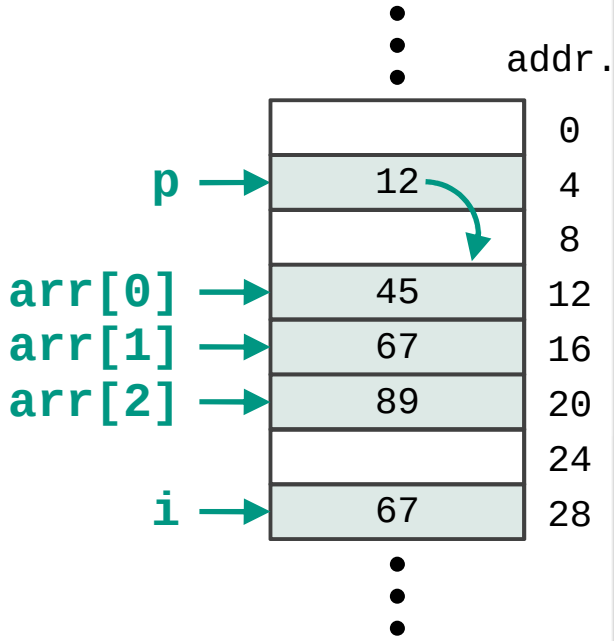
# Pointers and Arrays

- Arrays are allocated as continuous areas in memory with a consecutive layout of their elements
- Using pointer arithmetic, pointers can address arbitrary elements of arrays
- Array names *decay* to the address of the first array element

```
int arr[] = { 45, 67, 89 };
int *p, i;

int main() {
   p = arr;      /* p is assigned the address of
                    the first element of arr */
   p = &arr[0];  /* same effect as above */

   i = arr[1];   /* i is assigned the value 67 */
   i = p[1];     /* same effect as above */
   i = *(p + 1); /* same effect as above */
   return 0;
}
```



# Function Pointers

- C/C++ allows pointers to functions
  - which in turn allows assigning functions to variables
    - similar to Scala and X10
- Declaration is similar to other languages
  - declares a variable foo that points to a function
    - expecting an int and delivering void

```
void (*foo) (int);
```

- A function's address can be retrieved as shown below

```
void my_int_func(int x) {
    printf( "%d\n", x );
}

…

void (*foo)(int);
foo = &my_int_func;
foo(2);
```

*fetch address of my_int_function*

cf. e.g. http://www.cprogramming.com/tutorial/function-pointers.html

# Parameter Passing in C

- Default: pass by value
- Exception: array parameters
  - syntax suggests copy of whole array
  - but only the pointer to the first address is passed

```
void incr(int arr[], int length) { /* as a function parameter, int arr[]
                                      implicitly means int *arr */
   int i = 0;
   for(i = 0; i < length; i++) {
      arr[i]++;                    /* array elements can be accessed by
                                      standard pointer subscription */
   }
}

int main() {
   int arr[] = { 45, 67, 89 };
   incr(arr, 3);                   /* arr decays to &arr[0] */
   return 0;
}
```

# References

- Only available in C++
- Represent an *alias* for a variable
- Declared using the reference modifier (&)
  - in contrast to a pointer it does not need to be dereferenced
- Must be defined together with the declaration
  - except in –
    - function parameters
    - return types of function calls
    - declarations with `extern` specifier
  - in other words, once a reference is created, all assignments only change the referenced value

# References vs. Pointers

- References are used in C++ to realize *pass by reference* for function parameters
- C only knows *pass by value* (except for arrays) and thus requires pointers to avoid parameter copies

```
void incr(int *i) {
   (*i)++;
}

int main() {
   int i = 1;
   incr(&i);
   return 0;
}
```

Pass by value (C/C++)

```
void incr(int &i) {
   i++;
}

int main() {
   int i = 1;
   incr(i);
   return 0;
}
```

Pass by reference (C++ only)

# Memory Management in C

- Each program is allocated into three areas (segments) of memory:
  - *text segment* (or *code segment*): contains the program instructions
  - *stack segment*: automatic variables within functions
  - *heap segment*: global and static variables, dynamically allocated memory
- Direct memory manipulation through pointer variables possible
  - ➔ risk of memory errors!
- Explicit dynamic allocation of heap memory through `malloc()` and `free()` possible
  - do not forget to free your memory once you do not need it anymore
    - no garbage collection!

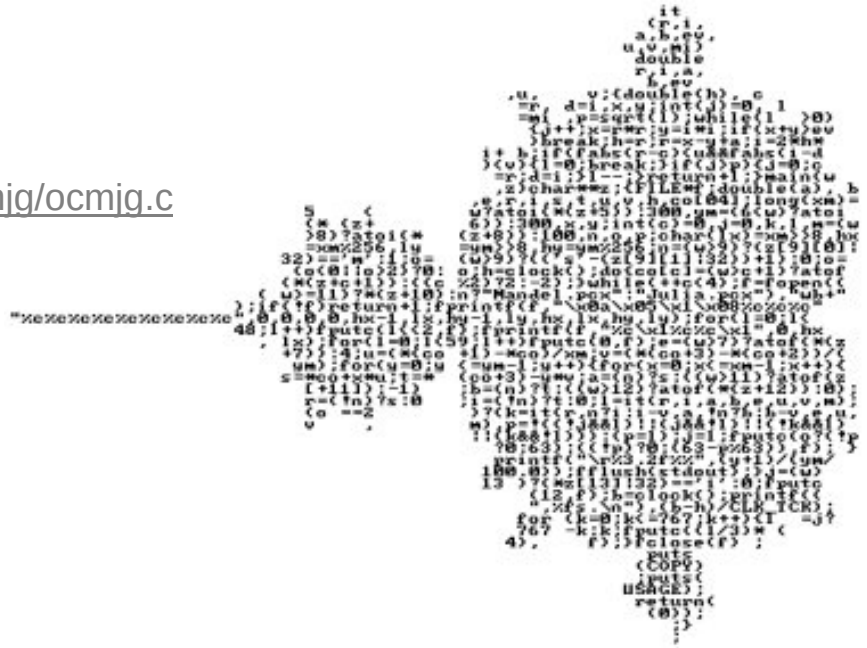
# Memory Management in C++

- C++ provides operators `new` and `delete`
  - instead of `malloc()` and `free()`
  - ➔ includes constructor / destructor execution
    - the required memory size is automatically calculated
- Like in C, any data type can be dynamically allocated on the heap
- Explicit memory release through `delete` is still necessary
  - no garbage collection

# Understanding C

- Are you good at it?

http://www.funiter.org/ocmjg/ocmjg.c



```
long long n,u,m,b;main(e,r)char **r;{f\ or(;n++||(e=getchar()|32)>=0;b="ynwtsflrabg"[n%=11]-e?b:b*8+ n)
for(r=b%64-25;e<47&&b;b/=8)for(n=19;n;n["1+DIY/.K430x9\ G(kC["]-42&255^b||(m+=n>15?n:n>9?m%u*~-u:
~(int)r?n+ !(int)r*16:n*16,b=0))u=1ll<<6177%n--*4;printf("%llx\n",m);}
```

http://www.ioccc.org/2012/kang/kang.c

- Still too easy?
  - go and practice at ioccc.org ☺

# Declarations in C

- *Forward declaration* principle
  - all entities (variables, types, functions) must be declared before use
- Multiple declarations of the same identifier may exist
  - e.g., in multiple files
- The actual definition of the identifier occurs in exactly one place
  - and may be integrated with a declaration

# Declarations in C: Modifiers

- `const`
  - read-only after definition
  - attention: through pointers and direct memory access, changing read-only data is still possible!
- `volatile`
  - always fetch value from main memory
  - no registers, no optimization
  - useful if variable is accessed outside the user program control (e.g., I/O buffers)
- Other modifiers…
  - type-specifiers: `void, char, short, int, long, signed, unsigned, float, double`
  - storage-class: `extern, static, register, auto, typedef`

# Declarations in C: Challenges

- C declarations are sometimes hard to read:
  - no simple reading from left to right

```
int * arr[]; /* arr is an array of
                pointer to int */
```

  - potentially nested declarations

```
int *(*p)(); /* p is a pointer to a
                function returning a
                pointer to an int */
```

  - modifiers `const` and `volatile`

```
volatile int * const i; /* i is a
                           read-only pointer to
                           a volatile int */
```

- What does the following declaration mean?

```
static unsigned int* const *(*next)();
```

# The Precedence Rule [Linden1994]

```
static unsigned int* const *(*next)();
```

A „[name] is a…“

B Follow the precedence order:

- B.1 parentheses ( )
- B.2 postfix operators:
  - B.2.1 ( ) „…function returning…“
  - B.2.2 [ ] „…array of…“
- B.3 prefix operator: * „…pointer to…“
- B.4 prefix operator * and `const` / `volatile` modifier: „…[modifier] pointer to…“
- B.5 `const` / `volatile` modifier next to type specifier: „…[modifier] [specifier]“
- B.6 type specifier: „…[specifier]“

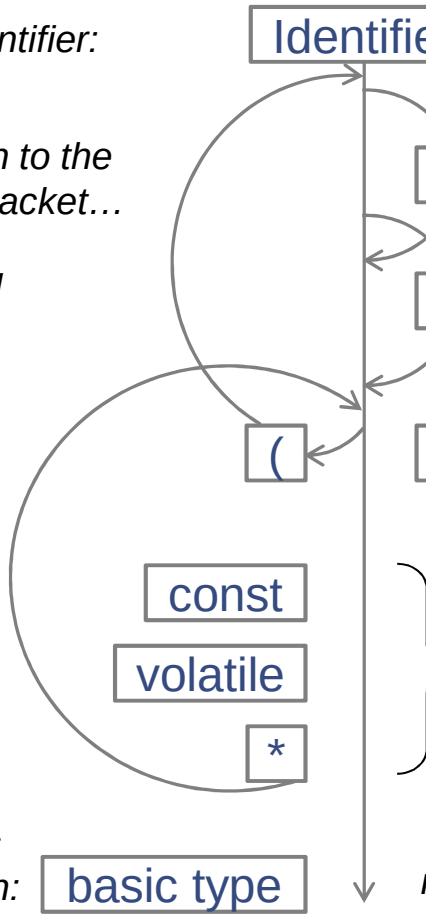
# The Precedence Rule: Example

```
static unsigned int* const *(*next)();
```

| | | | |
|---|---|---|---|
| 1. | A | `next` | „next is a …“ |
| 2. | B.3 | `*` | „…pointer to…“ |
| 3. | B.1 | `()` | „…“ |
| 4. | B.2.1 | `()` | „…a function returning…“ |
| 5. | B.3 | `*` | „…a pointer to…“ |
| 6. | B.4 | `*const` | „…a read-only pointer to…“ |
| 7. | B.6 | `static unsigned int` | „…static unsigned int.“ |

# “Decoder Ring” for C Declarations [Linden1994]

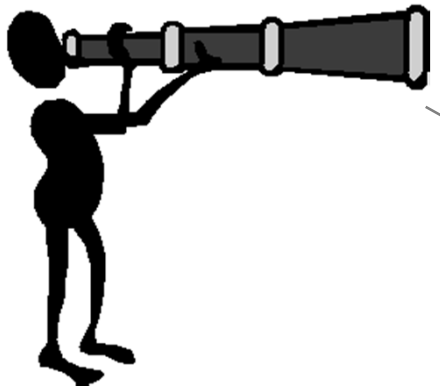
| Step number | Token to match | How to read |
|---|---|---|
| 1. Go to the leftmost identifier: | Identifier | say “identifier is…” |
| 2. Look at the next token to the right if it’s a square bracket… | [ possible-size ]… | for each pair, say “array of |
| 3. …or if it is an opening parenthesis | ( possible-parameters ) | read up to closing parenthesis, say “function returning…” |
| 4. If the token to the left is an opening parenthesis: | ( stuff-already-dealt-with ) | read up to balancing parenthesis, start again at step 2 |
| 5. If the token to the left is any of const / volatile / asterisk: | const, volatile, * | • for const say “read-only…”<br>• for volatile say “volatile…”<br>• for asterisk say “pointer to…”<br>→ keep reading tokens to the left, until it’s not one of these three, then goto step 4 |
| 6. The tokens that remain from the basic type of the declaration: | basic type | read off the remaining tokens, e.g. “static unsigned int” |

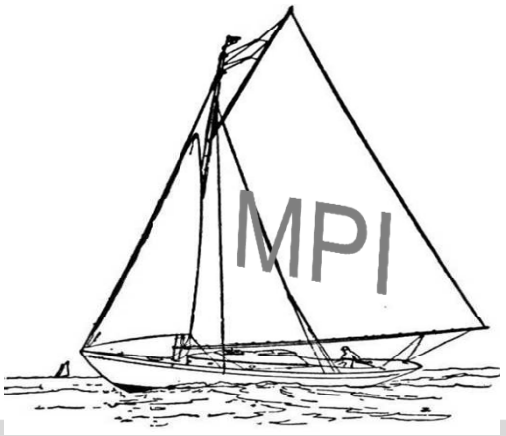


# Conclusion

- C and C++ are still widely used languages
  - especially in the embedded and high-performance domains
    - due to more degrees of freedom (manual memory management etc.)
- Although they are syntactically similar to Java, there are some subtle differences, such as –
  - the use of pointers
  - the possibility to implement multiple inheritance
  - or no fixed sizes for built-in data types
- Thank you for listening!

Scala
X10
C++





# Literature


[Linden1994] Peter van der Linden, „Expert C Programming“, Prentice Hall, 1994

[Meyer1997] Bertrand Meyer, „Object-oriented Software Construction“, 2nd Edition, Prentice Hall, 1997

[SGI1994] SGI Standard Template Library Programmer’s Guide, 1994, http://www.sgi.com/tech/stl

[Ullenboom2004] Christian Ullenboom, „Java ist auch eine Insel“, 4th Edition, Galileo Computing, 2004

[Wilhalm2004] Thomas Willhalm, „Von Java nach C++“, Internal Report, KIT, 2004, http://digbib.ubka.uni-karlsruhe.de/volltexte/1000001246

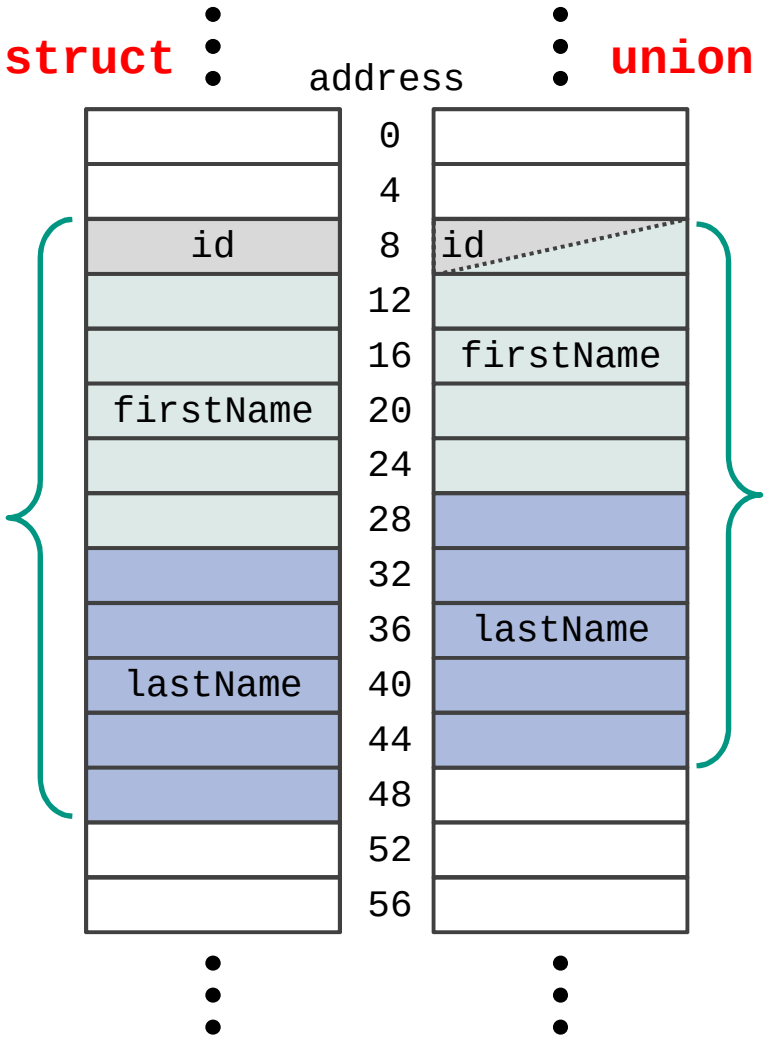
# Appendix: Structs vs. Unions

- A student with an id AND a name:

```
struct student {
   int id;
   struct {
      char firstName[20];
      char lastName[20];
   } name;
};
```

- A student with an id OR a name:

```
union student {
   int id;
   struct {
      char firstName[20];
      char lastName[20];
   } name;
};
```



# Appendix: Diamond Inheritance

- Diamond inheritance must use virtual base classes
  - avoids two sub objects in inherited class (D)
  - must already be considered when designing intermediate classes
    - i.e. B and C

*basically allows overriding at runtime*

```
class A {
   public:
      virtual int getData() {return 0;};
};
```

```
class B : virtual public A {
   public:
      int getData() {return 1;}
};
```

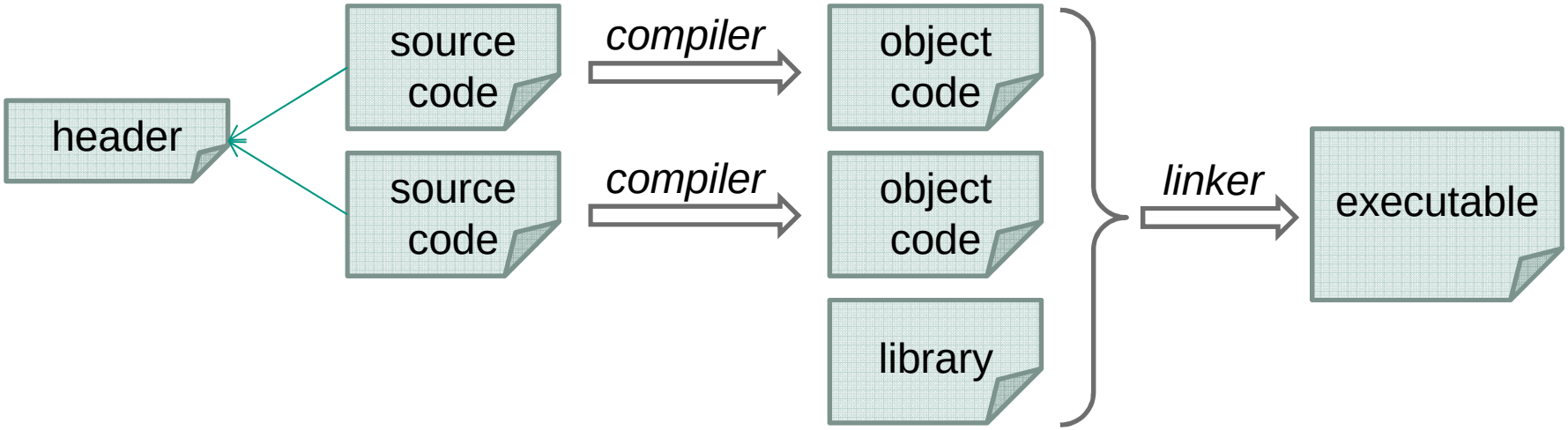
```
class D : public B, public C {
   public:
      int getData() {return B::getData();}
};
```

```
class C : virtual public A {
   public:
      int getData() {return 2;}
};
```

```
int main() {
   A *myA;
   D myD;
   myA = &myD; /* causes compile error without virtual base classes */
   return (*myA).getData(); /* which member function is executed? */
}
```

# Appendix: Compiling and Linking

- Artefacts
  - Source code files (including header files)
  - Object code files (including libraries)
  - Executable file
- Tools
  - Compiler
  - Linker



# Compiler and Linker Details

- Compiler
  - Compiles one source file into an object file:
    - `extern` declarations → undefined symbols
    - Global definitions → defined symbols
    - Local definitions → local symbols
  - Includes header files to allow for reusable forward declarations

- Linker
  - Combines object files (including libraries) into an executable
  - Resolves undefined symbols of individual object files
  - *Dynamic linking* allows for keeping undefined symbols in the executable and loading corresponding DLLs at run-time

# Compiling and Linking: Makefiles

- Allow for automation of the build process
- Define targets for compiling and linking
- Keep track of dependencies between artefacts

```
CC = g++
FLAGS = -Wall –g
hafas.o: hafas.cc
   $(CC) –c hafas.cc $(FLAGS) –o hafas.o
dijkstra.o: dijkstra.cc
   $(CC) –c dijkstra.cc $(FLAGS) –o dijkstra.o
hafas: hafas.o dijkstra.o
   $(CC) hafas.o dijkstra.o $(FLAGS) –o hafas
```

[Wilhalm2004]

# C++ Special Keywords

- `asm`
  - C++ inline assembler
- `explicit`
  - prohibits automatic conversions
- `friend`
  - grants access to private and protected class members
- `inline`
  - function is directly inserted into calling code
- `mutable`
  - allows a data member of a `const` object to be modified
- `operator`
  - creates overloaded operator functions
- `virtual`
  - allows member functions to be overridden by a derived class

# C++ Standard Template Library (STL)

[SGI1994]

- Contains basic data structures and algorithms
- Generic programming, parameterized classes
- Based on *concepts* and *refinements*

- Concrete Contents –
  - containers (vector, list, set, map, …)
  - iterators (istream_iterator, insert_iterator, sequence_buffer, …)
  - algorithms (find, count, search, copy, swap, replace, remove, sort, …)
  - other contents (function objects, utilities, memory allocation)