Karlsruhe Institute of Technology

Programmierparadigmen

Prof. Dr. Ralf Reussner

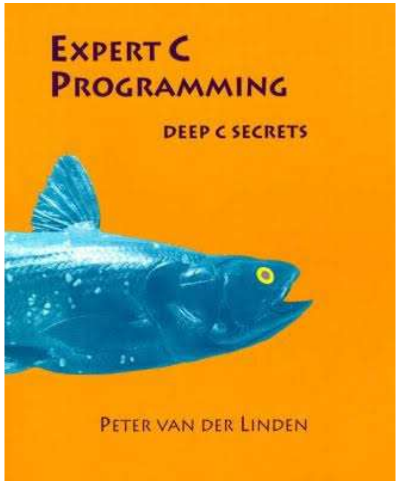
Topic 5.2

C/C++ Repetition

DEPENDABILITY OF SOFTWARE-INTENSIVE SYSTEMS
INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY, FACULTY OF INFORMATICS

dsis.kastel.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

- 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++
  - C with 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 defines 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

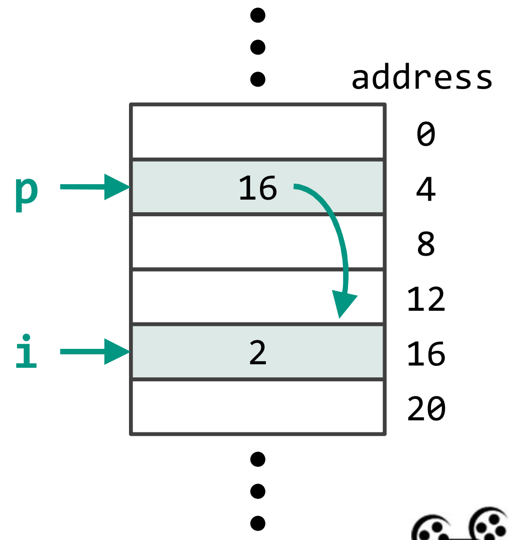
- Data types: `void`, `int`, `float`, `double`, `char`, `enum`
- Modifiers: `short`, `long`, `signed`, `unsigned`
- Actual sizes and ranges of data types are platform-dependent

- Arrays: indicated by `[]`
- Boolean values
  - C: `int = 0` for *false*; `int ≠ 0` for *true*
  - C++: `bool`
- Strings
  - C: `char[]` terminated with `'\0'`,
  - C++: `std::string`
- 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

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 (*)

```c
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 Arithmetic

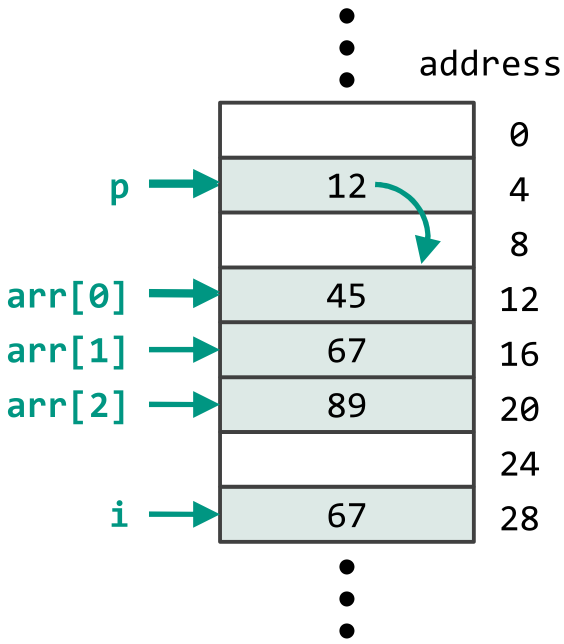
- Pointers can be *incremented* and *decremented*
  - by the (platform-specific) size of their data types
- They can also be *subscripted* like accessing an element of an array
  - Attention: risk of data errors due to direct memory access

```
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 (1)

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

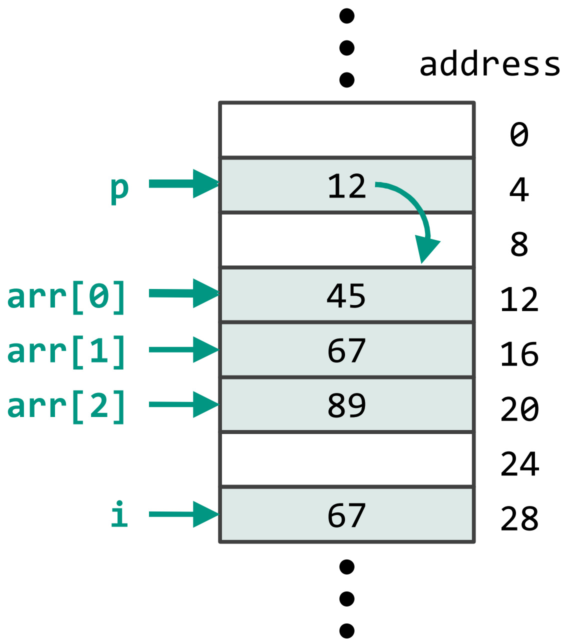
```
int arr[] = { 45, 67, 89 };
int *p, i;

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;
}
```



Pointers and Arrays (2)

- Notice: even though we write `p = arr`, `p` is not the same as `arr`:
  - `arr` (or `&arr[0]`) **is** the address of the first array element
  - `p` **points to** the address of the first array element
  - leads to **linking error** if in one file `int p[5]` and in other file `extern int *p` is defined / declared

```
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;
}
```



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;
}
```


Dependability of Software-intensive Systems
Institute of Information Security and Dependability

Function Pointers

- C/C++ supports pointers to functions
  - which in turn allows assigning functions to variables

- Declaration is similar to other languages
- Example: `void (*foo) (int);`
  - declares a variable `foo` that points to a function expecting an `int` and delivering `void`

- 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

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)

Dependability of Software-intensive Systems
Institute of Information Security and Dependability

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
  - Attention: risk of memory errors

- Explicit dynamic allocation/deallocation of heap memory through `malloc()` and `free()` possible
  - No garbage collection
    → Do not forget to free your memory once you do not need it anymore

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)

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: 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?

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

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`

The Precedence Rule [Linden1994]

```
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

```
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	`unsigned int`	"…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”
		• for `volatile` say “volatile”
		• for `*` say “pointer to”
		→ 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. “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
  - no fixed sizes for built-in data types
  - the lack of a garbage collection

Literature and References

[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

Appendix: 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

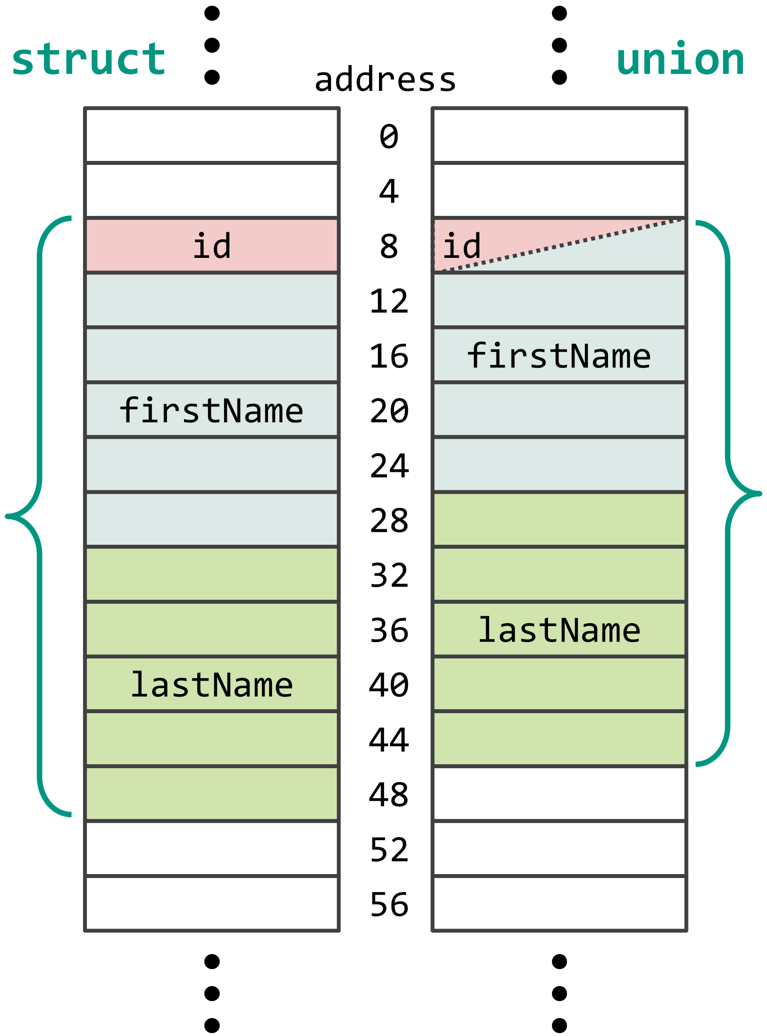
Appendix: Structs vs. Unions

- 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 XOR a name:

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



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

Appendix: Inheritance in C++

- C++ classes can have inheritance relationships just like in Java

like extends in Java

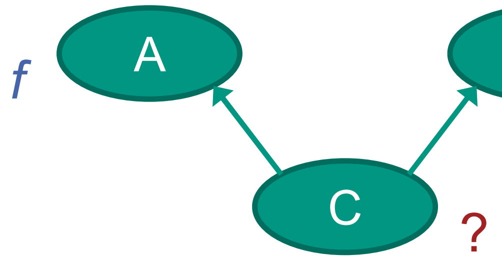
```
class A {
   public:
      int getData() {
         return 1;
      }
};
```

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

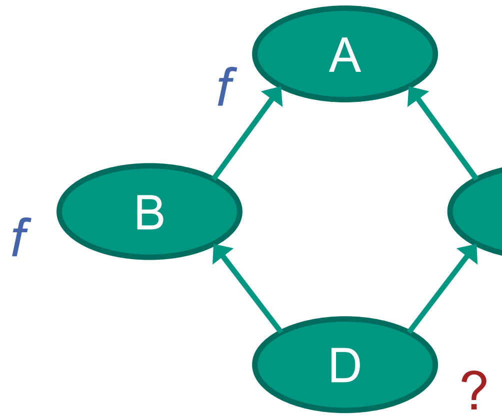
- C++ also allows for multiple inheritance
  - May cause ambiguities if methods of multiple inherited classes have the same signature
  - Is not relevant in this lecture (cf. Appendix for more detailed information)

Appendix: 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)*



Appendix: 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;
}
```

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: Default Implementations in Java

- Java does not allow multiple inheritance for classes
- Java 8 introduced default implementations of methods in interfaces
  - *Reason:* Extending an interface (i.e. adding a method) is problematic, as it breaks all classes that implement the interface
  - *Solution:* Allow to add a methods with default implementations, adding the `default`-Keyword and a method body

```java
interface A {
    int existingMethod();
    default int newMethod() {
        int i;
        // calculation of i
        return i;
    }
}
```

```java
class C implements A {
    public int existingMethod() {
        return 0;
    }
}
```

Appendix: Default Implementation Pitfalls

```java
interface A {
    int a();
}
```

```java
interface B {
    default int a() {
        return 1;
    }
}
```

```java
class C
    implements B {
}
```

Compiles

```java
interface A {
    int a();
}
```

```java
interface B {
    default int a() {
        return 1;
    }
}
```

This could also be an interface

```java
class C
    implements A, B {
}
```

Does not compile

Compiler Error: The default method *a()* inherited from *B* conflicts with another method inherited from *B*

```java
interface A {
    default int a() {
        return 0;
    }
}
```

```java
interface B {
    default int a() {
        return 1;
    }
}
```

```java
class C
    implements A, B {
}
```

Does not compile

Compiler Error: Duplicate default methods named *a* with the parameters *()* and *()* are inherited from the types *A* and *B*

Appendix: Default Implementation Pitfalls

```java
interface A {
    int a();
}

interface B {
    default int a() {
        return 1;
    }
}

class C
    implements B {
}
```

Compiles

```java
interface A {
    int a();
}

interface B {
    default int a() {
        return 1;
    }
}

class C
    implements A, B {
}
```

Does not compile

```java
interface A {
    default int a() {
        return 0;
    }
}

interface B {
    default int a() {
        return 1;
    }
}

class C
    implements A, B {
}
```

Does not compile

- If at least two implemented interfaces declare a method and at least one them defines it, it must be overwritten in the implementing class

Appendix: Multiple Inheritance in Java

- Resolving the compilation error:
  - Option 1: Manually overwrite method

```
class C implements A, B {
   @Override
   public int a() {
      return 0;
   }
}
```

```
interface A {
   default int a() {
      return 0;
   }
}
```

  - Option 2: Reuse existing method

```
class C implements A, B {
   @Override
   public int a() {
      return A.super.a();
   }
}
```

Use *a()* of interface *A*

```
interface B {
   default int a() {
      return 1;
   }
}
```

- Default Implementations can (but should not) be used to mimic multiple inheritance in Java

Appendix: Stack vs. Heap in C

- Stack
  - Realized as *Last In First Out* (LIFO)
  - Automatic allocation / release of memory upon entering / leaving functions and blocks
  - Attention: recursive function calls and / or „big“ function parameters can lead to stack overflow!

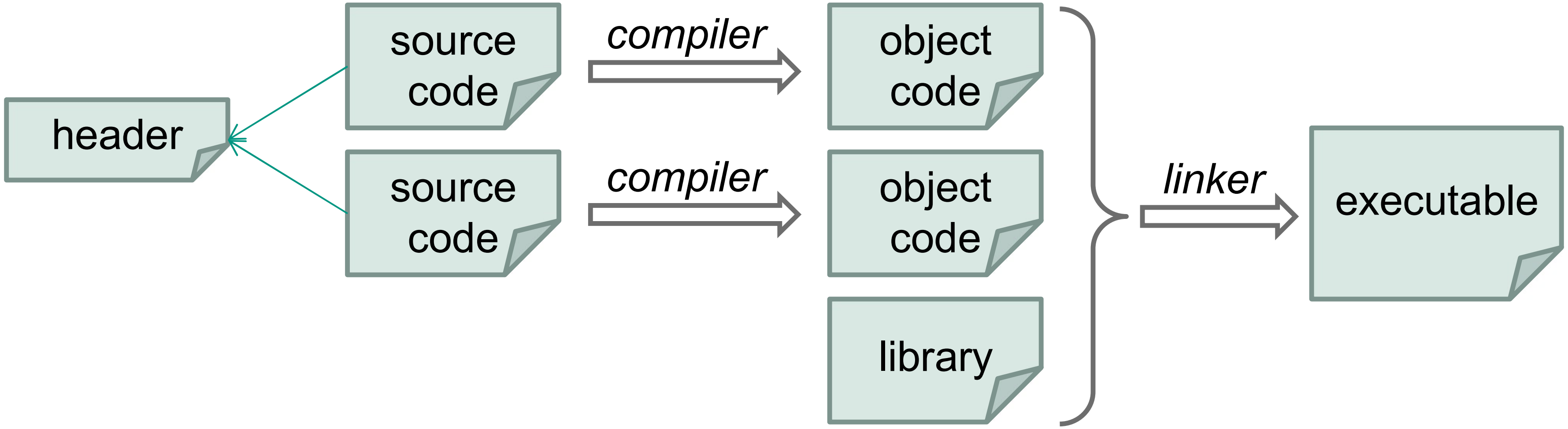
- Heap
  - Automatic allocation of global and static variables from their definition until end of program execution
  - Explicit allocation / release of arbitrary-sized memory blocks upon request
  - Attention: sequential memory requests do not necessarily lead to consecutively allocated memory blocks

Appendix: Memory Management Example

```c
#include <stdlib.h>

int i;           /* global variable – visible from here to end of file,
                    memory allocation on heap */
extern char c;  /* global variable, memory allocation at place of definition */
static float f; /* global static variable – visible ONLY in this file */

void incr(int *ptr) {
   static int delta = 2; /* local static variable – visible in incr */
   *ptr += delta;
}

int main() {
   double d = 3.7; /* local variable – visible in main, allocated on stack */
   int *ptr;       /* local pointer variable – same as above */
   ptr = (int *)malloc(sizeof(int)); /* dynamic memory allocation on heap */
   free(ptr);      /* release of allocated memory on heap */
   return 0;
}
```

Appendix: Compiling and Linking

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



Appendix: 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

Appendix: Makefiles for Compiling/Linking

- 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]

Appendix: C++ 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

Appendix: 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

Appendix: C++ Standard Template Library

- 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)