Simulative Evaluation of Internet Protocol Functions

Chapter 2 Java to C++ Primer

(Acknowledgement: this lecture is basically a shortened slide-version of [CS123])

© Prof. G. Schäfer

Introduction

- The C++ Programming Language:
  - Based on the C programming language
  - C++ adds object oriented features
  - Thus, programming style is completely different
  - Often said to be stricter than C and less strict than Java
  - Large and complicated language
  - Popular compilers incompatible (at best) or full of bugs (at worst). Most likely to find both!

"C lets you shoot yourself in the foot. C++ makes it harder, but when you do, you lose your whole leg"

- Java and C++ Similarities:
  - Java "looks like C++"
  - Class structure is very similar
  - Syntax and keywords are same or similar

(Acknowledgement: introductory slide compiled from [Pry, Agg00] with minor modifications)

© Prof. G. Schäfer

First Example: Hello World

```
[Hello.java]

// say that we are part of a package
// named hello:
package hello;

// declare a class called Hello:
public class Hello {

   // declare the function main
   // that takes an array of Strings:
   public static void main(String args[]) {

      // call the static method
      // println on the class System.out
      // with the parameter "Hello world!”:
     System.out.println("Hello world!"); }

} // end of <class hello>
```

```
[Hello.cc]

// include declarations for the standard
// I/O library where the printf function
// is specified:
#include <stdio.h>

// declare the function main that
// takes an int and an array of strings
// and returns an int as the exit code:
int main(int argc, char *argv[]) {

   // call the function printf with the
   // parameter "Hello world!\n",
   // where \n is a newline character:
   printf("Hello world!\n");

} // end of <main()>
```

Comparison of Hello World in Java and C++ (1)

- Key differences:
  - In C++, there can be global functions, that is functions which are not methods of a class, such as `printf` and `main`
  - In C++, we have a `#include` statement, that the compiler to read in a file that usually contains class or function declarations
  - The Java program includes a *package declaration*, in C++ has no analogous concept of packages (however, there are *namespaces...*)
- Further differences:
  - Program arguments:
    - In Java, main takes one parameter, an array of strings
    - If we want to find out how many elements are in a Java array, we can query it by accessing the array's length property
    - In C++, we use a `char*` instead of a String (will be explained later)
    - In C++, arrays have no properties or methods; they aren't pseudo-objects as in Java
    - For this reason, main takes a second parameter, `argc`, the number of arguments contained in the argument array `argv`

Comparison of Hello World in Java and C++ (2)

- Further differences (cont.):
  - Return value:
    - In Java the main method does not return a value
    - In C++ it returns an integer, known as the *exit code*
    - The exit code signifies whether or not the program terminated successfully:
      - A value of 0 indicates success
      - Any other value means the program failed
      - If no value is explicitly returned, it will automatically return a value indicating success (however, you should always explicitly return something!)

Classes (1)

```
[Foo.java]
public class Foo          // declare a class Foo
{
    protected int _num; // declare an instance variable of type int

    public Foo()          // declare and define a constructor for Foo
    {
        _num = 5;         // the constructor initializes the _num
                          // instance variable
    }
}
```

- This example shows the declaration of:
  - a class `Foo`,
  - with an instance variable of type `int`, and
  - the constructor function of `Foo` initializing the instance variable

Classes (2)

```
[Foo.h]
class Foo      // declare a class Foo
{
public:        // begin a public section
   Foo();      // declare a constructor for Foo
protected:     // begin a protected section
   int m_num;  // declare an instance variable of type int
};             // note the semicolon after the class declaration!
```

```
[Foo.cc]
#include "Foo.h"     // include the class declaration

Foo::Foo()           // definition for Foo's constructor
{
   m_num = 5;        // the constructor initializes the _num
                     // instance variable
}
```

Classes (3)

- In C++, the program is split into two files:
  - *Header file* (which we gave the extension .h), and
  - *Program file* (which we gave the extension .cc).
  - The header file contains the class declarations for one or more classes, and the program file contains method definitions
  - The program file includes the header file so that it knows about the declarations

- Advantages of splitting program declaration and definition:
  - One can easily look at a header file and see the interface for a particular class, without being having to see its implementation
  - Separating the header and program files can speed program compilation:
    - When the implementation of a class *A* changes, as other files containing classes that make use of *A* do not need to be recompiled, as long as the declaration of *A* has remained unchanged
    - This can be detected by comparing file dates and times

Classes (4)

- In Java, every member variable or method declaration in a class definition explicitly specifies its visibility by *access modifiers*
- In C++ this is done by the *access specifiers*:
  - `public`: general accessibility
  - `protected`: only accessible by instances of the same or derived classes
  - `private`: only accessible by instances of the same class
  - An access specifier defines the visibility of all members until the next access specifier
  - The default specifier is private for classes and public for structs
- Furthermore, you can declare another class *B* or global function *f* to be a *friend* of a class A:
  - In this case all instances of class *B* and the function *f* can access all private and protected members of class *A*
  - Friendship is not transitive (as in real life :o)

Classes (5)

- Naming convention for member variables:
  - In C++, "_" and "__" are reserved for the compiler and libraries
  - Using variables that *start* with those characters could lead to a crazy compiler error or worse
- The scope operator `::` is used when declaring methods:
  - If a class called Foo has a method called myMethod, when defining the function in the .cc file, it is called: `Foo::myMethod`
  - The scope operator is needed because a .cc file could contain method definitions for multiple classes, so we need to know for which class each method is being defined
- One minor but important syntax issue:
  - Remember to finish every class definition in a header file with a '`;`'
  - Otherwise you might get strange errors (sometimes even in other files)

Constructors and Initializer Lists (1)

- When an instance of a class is created, it is often required to initialize various instance variables, some of which are objects
- Java allows to initialize those variables and perform other startup tasks in the constructor
- C++ constructors can take a variety of parameters as in Java, plus there are some special constructors that will be discussed later
- In addition, instance variables can be initialized in a so-called *initializer list* before the rest of the constructor is called:
    - Whether to use initializer lists for this purpose is partially a matter of personal preference
    - However, knowing the syntax of initializer lists is sometimes needed, such as when calling superclass constructors

```
[Foo.h]   // Example Class declaration (see also next slide)
class Foo {
public:    Foo();
protected: int m_a, m_b;
private:   double m_x, m_y; };
```

© Prof. G. Schäfer

Constructors and Initializer Lists (2)

```
[Foo.cc] // with initializer list (functional equivalent)
#include "Foo.h"

Foo::Foo() : m_a(1), m_b(4), m_x(3.14), m_y(2.718)
{
    printf("The value of a is: %d", m_a);
}
```

```
[Foo.cc] // without initializer list (functional equivalent)
#include "Foo.h"

Foo::Foo()
{
    m_a = 1; m_b = 4; m_x = 3.14; m_y = 2.718;
    printf("The value of a is: %d", m_a);
}
```

- Don't use `this` inside an initializer list – it doesn't point to `this`

© Prof. G. Schäfer

Inline Declarations

- In Java, methods are declared and defined in the same place
- This can be done in C++ as well, by defining methods in the header file (→ *"inline declaration"*), but leads to a different compilation result:
  - Code declared inline is inserted for the method directly in the calling function, thus avoiding the large overhead of calling a function
  - Code specified in a .cc file leads to a function being called
- Inline declarations are typically used only for very short methods such as accessors and mutators, as well as other short, frequently called functions that do not require use of an external library

```
[Color.h]
class Color {
public:
   Color();
   int getRed() { return m_red; }        // inline declaration
   void setRed(int red) { m_red=red; }  // inline declaration
protected: int m_red, m_green, m_blue;
};
```

© Prof. G. Schäfer

Overloading (1)

- Both Java and C++ allow to declare more than one function with the same name:
  - This is called *overloading*
  - C++ uses the types of the parameters to determine which version of the function to call
  - There are rules about when C++ will do implicit casts, but making use of these rules requires a thorough understanding (see [Str00]).
  - Therefore, it is recommended for beginners to avoid ambiguity when overloading functions:
    - If possible, only overload on the number of parameters as opposed to the types of the parameters until you have learned all the rules
    - Or, if possible, call the functions by different names to avoid overloading entirely (OpenGL uses this method).

© Prof. G. Schäfer

Overloading (2)

```
[Foo.h]
class Foo
{
public:
   Foo();

   print(int a)    { printf("int a = %d \n",a); }
   print(double a) { printf("double a = %lf \n",a); }
};
```

- On an instance "foo" of type "Foo", calling:
  - `foo.print(5);` will output: `int a = 5`
  - `foo.print(5.5);` will output: `double a = 5.5`
- Attention:
  - When when overloading functions so that they take either a pointer type (see below) or an integer, the symbol NULL is actually an integer
  - The workaround is to explicitly cast NULL to the pointer type you want

Default parameters

- It is possible to declare default values for parameters of functions in the .h file:
  - If fewer parameters are passed than the function takes, it will use the default values
  - Using default values can sometimes help you avoid overloading functions or constructors
  - Note that parameters without default values must precede all the parameters with defaults

```
class Foo {
public:
   Foo();
   void setValues(int a, int b=5) { m_a = a; m_b = b; }
protected:  int m_a, m_b; };
```

- For an instance "foo" of class "Foo" the following two examples lead to the same result:
  - `foo.setValues(4);`
  - `foo.setValues(4,5);`

Inheritance (1)

- Inheritance is quite similar in Java and C++
- A class *B* can have a public superclass *A*:
  - There are types of inheritance other than public, but they are rarely used in real programs
- Furthermore, in C++ a class can inherit from more than one superclass:
  - This is called *multiple inheritance*
  - However, as multiple inheritance can lead to troubling issues and this course is not actually a programming course, we will not deal with it
- Initialization during instantiation:
  - Parameters to a superclass constructor can be passed in the initializer list

Inheritance (2)

```
[Foo.h]
class A
{ public: A(int something);
};

class B : public A
{ public:    B(int something);
};
```

```
[Foo.cc]
#include "Foo.h"
A::A(int something)
{ printf("Something = %d\n", something); }

B::B(int something) : A(something)
{ }
```

- In this example:
  - Class B inherits from class A
  - The parameter `something` is passed to the constructor of class A

Virtual Functions (1)

- Consider the following example in Java:

```
public void someMethod() {
   Object obj = new String("Hello");
   String output = obj.toString(); // calls String.toString(),
                                   // not Object.toString()
}
```

- The method `toString()` is defined in class `Object` and overridden in class `String`
- In the above example, Java knows that `obj` is really of type `String`, so at it calls the `String.toString()` method
- This is polymorphism at work
- It can resolve which method to call at run-time since in Java, all methods are virtual
- In a virtual method, the compiler and loader (or VM) make sure that the correct version of the method is called for each particular object.

Virtual Functions (2)

- In C++, functions are not virtual by default, as making everything virtual by default adds overhead to a program which would be against C++'s philosophy:
  - If you don't declare a function virtual and override it in a subclass, it will still compile even though the "correct" version of the method may not get called!
  - The compiler may give you a warning, but you should simply remember to do this for any function that you may override later.
  - In C++, the `virtual` keyword (the opposite of the Java keyword `final`), allows you to say that a function is virtual
  - If a function is declared virtual in a superclass, it is also virtual in all derived classes
    - However, you should always explicitly mark virtual functions in derived classes as virtual in order to make your code easier to read

Virtual Functions (3)

```
class A
{
public:
   A();
   virtual ~A();
   virtual void foo();
};

class B : public A
{
public:
   B();
   virtual ~B();
   virtual void foo();
};
```

- Destructor functions (explained later) should always be declared virtual:
  - Otherwise it might happen that not all memory gets released if an instance of a derived class is to be destroyed (see also memory allocation)

Overriding and Scope (1)

- Consider a class *A* and its subclass *B*
  - that both have a virtual function `foo`, and
  - B wants to call A's `foo`.
- In Java, you would use the `super` command to use A's `foo` from B's
- However, C++ has multiple inheritance, so another way is needed to specify which `foo` to call.
- This can be achieved with the scope operator `::`

```
[Foo.h]
class A {
public: A();
        virtual void foo();
};

class B : public A {
public: B();
        virtual void foo();
};
```

Overriding and Scope (2)

```
[Foo.cc]
#include "Foo.h"

A::foo()
{  printf("A::foo() called\n"); }

B::foo()
{  printf("B::foo() called\n");
   A::foo(); }
```

- For an instance *b* of class *B*, calling

  ```
  b.foo();
  ```

  will output

  ```
  B::foo() called
  A::foo() called
  ```

- Attention: Overriding an overloaded function in a derived class, also hides other overloaded versions of the superclass

C++ Variables and Memory Management (1)

- In C++, variables are declared in exactly the same way as in Java. Declaration of an integer variable would look like this under both languages:

  ```
  int myNumber;
  ```

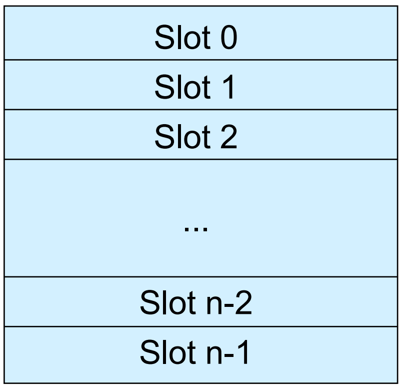
- You can also assign values to local variables at the time of declaration, just as in Java:

  ```
  int myNumber = 0;
  ```

  - Instance variables can not be assigned a value when declared in the header file; they are initialized in the constructor instead
- Thus, C++ and Java declare base type variables in basically the same way
- When it comes to declaring variables that can hold a class, however, things get a little more interesting
- In under to understand these issues, some basic knowledge about memory management is required

C++ Variables and Memory Management (2)

- Memory can be thought of as a very large number of "slots"
  - Each slot holds 8 bits, or one byte
  - Current PCs typically have 256 - 1024 megabytes of memory, meaning they have about as many memory "slots"
  - To organize all these slots, you can think of the computer as arranging them in a list – slot 0 is at the beginning, slot 1 follows it, and so on, until there are no more slots:



C++ Variables and Memory Management (3)

- All slots have a unique number associated with them:
  - Referring to "slot 5" is always talking about the same slot
  - Each of these slots can hold a single byte
  - It is possible to write code like: “add the byte in slot 7 to the byte in slot 20”
- Data occupying more than one byte are stored in consecutive bytes:
  - Integers, for example, take 32 bits to store (4 consecutive bytes)
  - Larger types, e.g. classes, are similarly stored in consecutive slots
  - The computer stores an instance of a class in memory by turning it into several numbers, containing the values of the instance variables of the class and other such information
  - Like with integers, it is possible to write code like:
    - "take the class starting at slot 5 and do something to it"
    - Since an instance of a class takes several slots, it is dealt with in terms of the first slot it occupies
    - The compiler keeps track of how large each class instance is so that it knows how many slots after the initial one are used.

Pointers (1)

- Two definitions:
  - A *memory address* is the number of one of the slots mentioned above
  - A *pointer* is a memory address (augmented with type information)
- So, a pointer to an integer `myInt` is:
  - "the number of the slot that stores `myInt`",
- To declare a pointer to an integer, we place the star symbol `*` between the data type and the variable name:

```
int* myIntegerPointer;
```

  - One of the uses of the `*` is to tell the compiler that we want something to be a pointer when we are declaring it
  - So the line above means "I want a pointer to an integer" and not just an integer
  - When you declare a pointer, it is pointing to nothing, or even worse, it often points to a random slot, until it is initialized
    - Attention: Using an uninitialized pointer, usually causes a program to crash with a segmentation fault or a bus error

Pointers (2)

- In order to make a pointer point to the memory address of a variable, the variable's memory address needs to be obtained
  - To get the memory address of something, we use the `&` symbol. One of its meanings is "address of"
- The following code declares an integer and lets a pointer point to it:

```
int* myIntegerPointer;
int myInteger = 1000;
myIntegerPointer = &myInteger;
```

- The following example programs make heavy use of the `printf` statement and special formatting characters, such as `%p` or `%d`
- These will be explained later in the section on the standard I/O library

Pointers (3)

```
[main.cc]
#include <stdio.h>
int main(int argc, char **argv) {
 // declare an integer with value 1000:
 int myInteger = 1000;

 // declare a pointer to an integer
 // and make it point to myInteger:
 int* myIntegerPointer = &myInteger;

 // print the value of the integer:
 printf("%d\n", myInteger);

// print the value of the pointer
printf("%p\n", myIntegerPointer);
}
```

- Output (second value is a guessed example value) :

```
1000
4026529420
```

Pointers (4)

- Changing the value to which a pointer points:
  - Consider a pointer to an integer and that the value of the integer to which it points needs to be changed
  - In order to make the compiler to produce code to "set the value of the integer at memory address x to 50," the compiler has to be told to use the integer at address x, not the address x itself
  - For instance, the code `myIntegerPointer = 50`
    - does not mean: "set the number that myIntegerPointer points to to 50"
    - but rather: "set the value of myIntegerPointer to 50"
    - This will change the memory address that myIntegerPointer actually points to; myIntegerPointer will now improperly point to "slot 50."
  - In order to modify the integer, the pointer needs to be dereferenced before using it. This is where the second use of the "`*`" comes in:
    - `myIntegerPointer` means "the memory address of <myInteger>."
    - `*myIntegerPointer` means "the integer at memory address <myIntegerPointer>."

Pointers (5)

```
// declare an integer and a pointer pointing to it:
int myInteger = 1000;
int *myIntegerPointer = &myInteger;

// print the value of the integer before changing it:
printf("%d\n", myInteger);

// dereference the pointer and add 5 to the integer
// it points to:
*myIntegerPointer += 5;

// print the value of the integer after changing it
// through the pointer:
printf("%d\n", myInteger);
```

- The output is:

```
1000
1005
```

Pointers (6)

```
// declare an integer and a pointer pointing to it:
int myInteger = 1000;
int *myIntegerPointer = &myInteger;

// declare another integer whose value is the
// same as the integer at memory address <myIntegerPointer>:
int mySecondInteger = *myIntegerPointer;

// print the integer and then change it through the pointer:
printf("%d\n", myInteger);
*myIntegerPointer += 5;

// print the value of the integer of both integers:
printf("%d\n", myInteger);
printf("%d\n", mySecondInteger);
```

- The output is:

```
1000
1005
1000
```

Pointers (7)

- Example on previous slide:
  - Experiment:
    - dereference a pointer `myIntegerPointer,` pointing to the memory address of a variable `myInteger ,`
    - store it in another variable `mySecondInteger`, and
    - change the value of `myInteger`
  - Question:
    - will the value of `mySecondInteger` also change?
  - Answer:
    - no it will not, as the second integer pointer contains another memory address, that contains a copy of the original value
- Can more than one pointer point to the same address?
  - It is possible to have multiple pointers point to the same address
  - In such a case, changing the value of the number at that address changes the values the other pointers are pointing to, since it is the same address

© Prof. G. Schäfer

Pointers to Pointers

- Since pointers are just numbers in memory (e.g. 32-bit integers on x86 machines), it's possible to have pointers to these pointers:

```
int myInteger = 1000;
int* myIntegerPointer = &myInteger;

int** myIntegerPointerPointer;
myIntegerPointerPointer = &myIntegerPointer;
```

- `myIntegerPointerPointer` is a pointer to a pointer to an integer:
  - A pointer to an integer is of type `int *`
  - So, a pointer to a pointer to an integer is of type `int **`.
  - Dereferencing `myIntegerPointerPointer` once gives a pointer to an integer:

    `(*myIntegerPointerPointer) == myIntegerPointer`
  - Dereferencing `myIntegerPointerPointer` twice gives an integer

    `(**myIntegerPointerPointer) == myInteger`

© Prof. G. Schäfer

Pointers To Objects (1)

- Consider the following simple class declaration:

```
[Foo.h]
class Foo {
public:
   Foo();              // default constructor
   Foo(int a, int b); // another constructor
   ~Foo();             // destructor

   void bar();         // random method

   int m_blah;         // random public instance variable
};
```

- Declaring a variable to this class and creating the class in Java:
  - `Foo myFooInstance = new Foo(0, 0);`
- In C++:
  - `Foo* myFooInstance = new Foo(0, 0);`
  - In C++, new operator returns a pointer to whatever follows it

Pointers To Objects (2)

- Calling the method bar on the instance in Java:
  - `myFooInstance.bar();`
- In C++, myFooInstance is a pointer, and pointers need to be dereferenced before being used:

  -
    ```
    myFooInstance->bar();  // dereference the pointer and
                           // call the method
    ```
- Public instance variables of instances of Foo could be accessed likewise (of course, you would never do that...):
  - `myFooInstance->m_blah = 5;`
- The arrow operator `->` does two things:
  - it dereferences the pointer, and then
  - it calls a method on the instance or accesses a member variable.
- This is shorthand for saying:
  - `(*myFooInstance).bar();`

Instances (1)

- In Java, the only way to create an object is to new one and store a reference to it in a variable
- In C++, it is possible to declare objects without “newing” them explicitly:
  - `Foo myFooInstance(0, 0);`
  - This line of code creates a variable of type Foo and passes the specified parameters to its constructor.
- If we wanted to create a Foo instance using the default constructor instead, we could say:
  - `Foo myFooInstance; // same as Foo myFooInstance();`
- In Java, myFooInstance would be a null reference
- In C++, it's an actual instance
- If we don't want to refer to the instance later, say, because it is being passed as a parameter, we can leave out the variable name:
  - `bar.setAFoo( Foo(5,3) ); // pass an instance of Foo`

Instances (2)

- Calling methods and accessing public instance variables of an instance has the same syntax that you're used to in Java:
  - `myFooInstance.bar();`
  - `myFooInstance.m_blah = 5;`
- Like pointers, instances may be local variables or member variables
  - If an instance is a member variable of a class, its constructor can be called in the class's constructor's initializer list (see next slide)

Instances (3)

```
[Bar.H]
#include "Foo.h" // Required since an instance of Foo is created

class Bar {
public:
    Bar(int a, int b);
protected:
    Foo m_foo; // declare an instance of Foo
};
```

```
[Bar.cc]
#include ”Bar.h”

// call Foo::Foo(int,int) and initialize m_foo:
Bar::Bar(int a, int b) : m_foo(a,b) {
 // create another instance of Foo, this time as a local var:
 Foo fooLocal;
 // do something with the two Foos, m_foo and fooLocal
}
```

References (1)

- Sometimes, it may be useful to refer to block of memory created for an object with more than one name:
  - This can, for example, be done with pointers, since multiple pointers can point to the same object
  - There is a second way to do it without using pointers: we can use something called references instead:

```
[main.C]
#include <stdio.h>

int main(int argc, char **argv) {
    int foo = 10;
    int& bar = foo;
    bar += 10;
    printf("foo is: %d\n", foo);
    printf("bar is: %d\n", bar);
    foo = 5;
    printf("foo is: %d\n", foo);
    printf("bar is: %d\n", bar);
}
```

References (2)

- In the above example, memory has been allocated to hold an integer and named it `foo`
- The `&` sign in the second line declares a reference to an integer
- By assigning `foo` to `bar`, `bar` does not become a copy of `foo`, but instead refers to the same memory location as `foo`.
- When you change the value of `bar`, it also changes the value of `foo` and vice versa
- References are essentially the same as pointers, except that they:
  - are dereferenced implicitly (code looks like code for an instance),
  - can never be NULL, and
  - can only be assigned to once, at creation
- This makes references "safer" than pointers

© Prof. G. Schäfer

References (3)

- The output of the above program should look like:
  - `foo is: 20`
  - `bar is: 20`
  - `foo is: 5`
  - `bar is: 5`
- Since references can be assigned to only at creation, references that are members of a class must be assigned to in the constructor's initializer list:

```
[Bar.H]
class Foo;

class Bar {
protected:
    Foo & m_foo; // declare a reference to a Foo
public:
    Bar(Foo & fooToStore) : m_foo(fooToStore) {}
};
```

© Prof. G. Schäfer

Converting between Pointers and Instances (1)

- The * and & operators “convert” between pointers and instances
- Example 1 (making a pointer from a local variable):

```
Foo myFooInstance(0, 0);
Foo* fooPointer;
// set the value of the pointer to the address of foo:
fooPointer = &myFooInstance;
```

  - The following two statements have the same effect:

```
myFooInstance.bar(); // call bar through the instance
fooPointer->bar();   // call bar through the pointer
```

- Example 2 (making a copied local variable from a pointer, don’t do it):

```
// create a pointer to Foo and give it a value:
Foo* fooPointer = new Foo(0, 0);
// dereference the pointer and assign it:
Foo myFooInstance = *fooPointer; // copy is made (!)
```

Converting between Pointers and Instances (2)

- The second example may not have the result that you expect:
  - If you remember from earlier, we had an example of a pointer to an integer that we dereferenced and stored in a second integer variable
  - We discovered that the second integer was actually a copy of the first
  - A similar thing is happening here: the instance that fooPointer points to and myFooInstance are actually *two separate instances*
  - The first line news an instance and assigns the address of that instance to the pointer
  - The second statement dereferences the pointer and assigns the instance to `myFooInstance`
    - Here, the compiler performs a bitwise copy of the instance pointed to by `fooPointer` and assigns it to `myFooInstance`, or, if you have defined an assignment operator, it is called instead
    - So, saying `fooPointer->m_blah = 5;` would not change the value of `blah` in `myFooInstance`
  - Doing things like this yields really confusing code and is a potential source of hard to find errors. For this reason, it is usually a bad idea to do this.

Copy Constructors

- A copy constructor is a constructor that is invoked when one instance of a class is initialized with another instance of the same type
- The syntax for a copy constructor is:

```
class Foo {
   Foo(const Foo &classToCopy); // copy constructor
};
```

- A copy constructor usually assigns all the values of instance variables in the class that is passed in to the instance variables that this constructor is called on
- Note: the keyword `const` will be explained later

Memory Management

- Declaring and using variables is a major aspect of programming
- The memory needed to store these variables varies with the type of the variable and where it is declared
- There are three major categories of storage:
  - *Local storage:* valid only within a certain scope; also known as *automatic memory* or *storage on the stack*
  - *Global storage:* valid throughout the execution of the program;
  - *Dynamic storage:* valid from being allocated to being deallocated; also called *free store, dynamic memory,* or *storage on the heap*

Local Storage

- The following block of code shows an integer and a instance of the class Bar being allocated in local storage:

```
{
  int myInteger; // memory for an integer allocated
  // ... myInteger is used here ...

  Bar bar; // memory for instance of class Bar allocated
  // ... bar is used here ...
}
```

- The { and } symbols mark the beginning and the end of a block:
  - When program flow enters the block, memory needed to store an integer is allocated for `myInteger`, and memory needed to store the class instance is allocated for the variable `bar`
  - When the end of the block is reached, this memory used to store `myInteger` and `bar` is freed up and those variables cease to exist
  - Trying to use the variables after the block is closed will yield compile errors, just as in Java

Allocating Memory with New (1)

- In the above example, if the variable `bar` is required outside of its block, it needs to be `bar` in global storage instead
- In C++, a block of memory can be requested for certain data types in dynamic storage by using `new`, and be returned by using `delete`
- The following C++ code shows how you can allocate memory and use it later (continued on next slide):

```
[Bar.h]
class Bar {
public:
    Bar() { m_a = 0; }
    Bar(int a) { m_a = a; }
    void myFunction(); // this method will be defined in Bar.cc
protected:
    int m_a;
};
```

Allocating Memory with New (2)

```
[main.cc]
#include "Bar.h"

int main(int argc, char *argv[])
{
 // declare a pointer to Bar; no memory for a Bar instance
 // is allocated now, so that p currently points to garbage:
 Bar * p = NULL;

 {
  // create a new instance of the class Bar (*p)
  // store pointer to this instance in p:
  p = new Bar();
 }

 // since Bar is in dynamic storage, we can still call
 // methods on it so that this method call will be successful:
    p->myFunction();
}
```

Deallocating Memory with Delete (1)

- In Java, memory for an object is allocated with `new`, and a garbage collector frees the memory automatically when no existing object references it
- In C++, whatever memory is allocated in dynamic storage, must explicitly be deallocated:
  - Otherwise the program will swell in size and contain what are called *memory leaks*
  - To avoid memory leaks, you need to keep track of all the memory you have newed and free it when you no longer need it
- In the example on the last slide, the following line should be added at the end of the function in order to free the memory allocated for p:

```
delete p; // memory pointed to by p is deallocated
p = NULL; // to avoid multiple deletes or later usage
          // that could corrupt the heap
```

Deallocating Memory with Delete (2)

- Attention:
  - Only objects created using `new` should be deleted with `delete`!
  - Instances created in local storage are automatically recycled and should not be deleted explicitly
- For example, the following code will make a program crash:

```
Bar bar;       // bar not created with new
               // ... use the instance of Bar ...
delete &bar;   // EEK! bar is in local storage...
```

- Arrays and single instances are deleted differently:
  - See the arrays section for more information on deleting arrays

Managing Memory in Classes: Destructors

- Class destructors have already been mentioned earlier, but their principal use in C++ has not yet been explained
- In Java, you don't have to deallocate memory, so you seldom need to fill in the `finalize()` method for an object.
- In C++, memory that is newed is not deallocated automatically, so you have to explicitly free it:
  - Since you can free memory at any time your program is running, the question is when to do it
- The following is a good rule of thumb:
  - Memory allocated in a constructor should be deallocated in a destructor
  - Memory allocated in a function should be deallocated before it exits
- Attention:
  - If you later (accidentally) access memory, that has already been deallocated, you can expect everything from strange behavior to crashes
  - Memory errors are among the hardest to find, as they often can stay undetected for a long time and often imply errors in other program parts

Managing Memory: Parameters (1)

- In Java, parameters are passed by reference:
  - When you pass a reference to an object in Java, you can change the actual object by calling methods on it or accessing its public instance variables
- In C++, parameters can be passed either by reference or by value:
  - Think of passing by value as passing a copy instead of the real variable
- Here's an example of passing by reference:
  - The function IncrementByTwo is defined to take a reference to an integer

```
void IncrementByTwo(int & foo) { foo += 2; }
```

  - Since the function has a reference, it can alter the integer that is passed in to it and you can increment an integer variable by calling:

```
int bar = 0;
IncrementByTwo(bar);
```

  - The variable bar will now have been increased by two

Managing Memory: Parameters (2)

- The following definition defines the parameter to be passed by value:

```
void IncrementByTwo(int fooVal) { fooVal += 2; }
```

  - If the function is called as shown on the preceding slide, `bar` will still be 0 after `IncrementByTwo` has returned
  - This is because the formal parameter `fooVal` contains a copy of `bar`
  - So, passing by value will not give the intended result
- A third way to pass parameters is to pass pointers to them:

```
void IncrementByTwo(int* fooPtr) { *fooPtr += 2; }
```

  - The function is then called by passing a pointer:

```
int bar = 0;
IncrementByTwo(&bar); // note the & sign;
```

  - Since we passed a pointer to `bar`, it will be incremented by two

Managing Memory: Parameters (3)

- How to pass objects as parameters in C++?
  - Objects can be passed by reference, by value, or by passing a pointer to the object
  - Since objects are often “newed”, meaning that you have a pointer to them, they are most commonly passed by a pointer
  - Of course, they can be passed by reference as well
  - However, objects are generally not passed by value, since that implies that a copy of the object is being made
    - For small types like integers, making a copy is not a big deal; but for objects, this can take up a lot of time (and memory).
    - If you're passing by value to make sure that the object you passed in won't be changed, instead make the input parameter `const` (see below for a description of `const` parameters.)

© Prof. G. Schäfer

Managing Memory: Return Values (1)

- Return values can be passed in all the ways discussed above
- However, a common C++ pitfall should be noted:
  - Passing a local variable outside of its scope
  - Variables in local storage are automatically destroyed when the block they are located in closes
  - Thus, if a pointer or reference to a variable declared in this manner is returned, and the variable leaves scope at some time, the pointer or reference will point to “nirvana”

```
Foo* FooFactory::createBadFoo(int a, int b) {
  // create a local instance of the class Foo:
  Foo aLocalFooInstance(a,b);

  // return a pointer to this instance:
  return &aLocalFooInstance;
} // EEK! aLocalFooInstance leaves scope and is destroyed!
```

© Prof. G. Schäfer

Managing Memory: Return Values (2)

- Returning a reference to a local variable also leads to errors:

```
Foo& FooFactory::createBadFoo(int a, int b) {
  // create a local instance of the class Foo:
  Foo aLocalFooInstance(a,b);
  // return a reference to this instance:
  return aLocalFooInstance;
} // EEK! aLocalFooInstance leaves scope and is destroyed!
```

- The solution to this problem is to either return a pointer to an instance in dynamic storage, or to return an actual instance:

```
Foo* FooFactory::createFoo(int a, int b) {
  // return a pointer to an instance of Foo:
  return new Foo(a,b);
}

Foo FooFactory::createFoo(int a, int b) {
    return Foo(a,b);  // return an instance of Foo
}
```

Managing Memory: Arrays (1)

- In C++, arrays behave like pointers but they are not exactly the same:
  - Pointers are variables which contain a changeable address
  - Arrays are "array-typed" addresses that can't be changed
- As with most types in C++, arrays can be allocated either in local storage or in free storage using new:

```
<type> *arrayName = new <type>[array size];
```

  - For example, declaring an array of 100 integers looks like this:

```
int* integerArray = new int[100];
```

- We can also declare an array in local storage as follows:

```
<type> arrayName[array size];
```

  - Declaring that 100 element integer array in local storage:

```
int integerArray[100];
```

- With both declarations, the value of `integerArray` is the memory address of the first element contained within it
  - However, there are subtle differences in the type of the variable!

Managing Memory: Arrays (2)

- The remainder of the array elements are stored in consecutive memory locations following the first element
- To access a particular element in the array, the syntax is:
  `arrayName[element]`
- Arrays are indexed from 0, just as in Java. So, to access the fourth element in the array, we say:
  `arrayName[3]`
- When you index into an array, it is dereferenced automatically
- Thus, even though `integerArray` is a pointer to integers, accessing one of its members returns an integer `(int)` not a pointer to one `(int *)`
- Therefore, the following straightforward code works as expected:

```
integerArray[10] = integerArray[42] + integerArray[0] - 5;
```

Managing Memory: Arrays (3)

- If we have the array of 100 integers integerArray, what happens when we access element -10, or element 200?
  - In Java, a java.lang.ArrayIndexOutOfBoundsException would be thrown.
  - In C++, there is no such luxury
- Using the indexing scheme mentioned above, `integerArray[-10]` translates to the memory address: `(integerArray - 10)`
- What does that point to? Who knows. It could point to some data of a class, a program instruction, there is really no way of knowing
- If you write to arbitrary memory addresses, your program may behave in a strange way, or you could get a "segmentation fault", a "bus error", and your program crashes
- So, when using arrays, you must be very, very careful not to index beyond their bounds, or your program will crash in general
- Attention:
  - If you declare, e.g., `int i[10];` then `i[10]` is beyond the array!

Managing Memory: Multidimensional Arrays (1)

- In C++, multidimensional arrays can not be declared in dynamic storage, but there you can emulate them with arrays of arrays
- So, where in Java you could declare a 10x10 array like this:

```
int twoDArray[][] = new int[10][10];
```

- In C++, you have to do it another way, by declaring a 10 element array, whose elements are 10 element arrays of integers
  - As you recall, an “array of int” is like a "int *". We want an array of arrays of integers, so prefixing another "*" will give us something of type "int **".
  - So, declaring the array would look like this:

    ```
    int** twoDArray;
    ```

  - To make twoDArray be an array of arrays, we just have to new 10 arrays of pointers to integers:

    ```
    twoDArray = new int*[10];
    ```

  - Ok, now we have an array of 10 pointers to int, each of which we want to point to an actual array:

    ```
    for(int i=0; i < 10; i++)  twoDArray[i] = new int[10];
    ```

Managing Memory: Multidimensional Arrays (2)

- In reality, our 2d (10x10) array does not really need to be 2d
- We have 100 integers, and we want to index the integer at location `(row, col)` at any given time:
  - To do this, we could declare a 100 element array of integers and index it ourselves:

    ```
    int* myArray = new int[100];
    ```

  - To index myArray, let's say row 0 of the 2d matrix is the first 10 elements in this array. Row 1 is the next 10, and so on. Given this scheme, we can find the correct element in the array using the formula:

    ```
    myArray[row * 10 + col]
    ```

  - Here, we multiply `row` by 10, since this is how much we have to move in the array to find the first element of row `row`. We add `col` to it to specify how far in the row of 10 elements we move to find the col'th element in that row.
- Advantage of indexing by yourself: the array is stored in one continuous block of memory leading to better cache hits in the CPU

Managing Memory: Deleting Arrays

- In Java, you did not need to delete arrays
- In C++, this has to be done explicitly for arrays declared in dynamic storage
- If you recall, deleting a single element looks like this:

```
delete <pointer>;
```

- To delete an array, we need to tell the compiler we are deleting an array, and not just an element. The syntax is:

```
delete [] arrayName;
```

- Deleting 2d arrays takes a little more work:
  - Since a 2d array is an array of arrays, we must first delete all the elements in each of the arrays, and then we can delete the array that contains them:

```
for(int i=0; i < (number of arrays); i++)
    delete [] 2dArray[i];
delete [] 2dArray;
```

Polymorphy (1)

- As you probably know from Java, you can use instances of subclasses as if they were instances of a (direct or indirect) base class
- This is called polymorphy
  - polymorphism only works with pointers and references to objects in C++, not with the objects as value
  - all methods that may be overridden in subclasses should be virtual

Dynamic cast

- Sometimes you have a pointer or reference of base class type but you know that it really points to an instance of a subclass
- If you want to access methods only present in the subclass you need to perform a dynamic cast
- A dynamic cast is checked at runtime. If the cast is not possible (the object is not of the given type) `NULL` will be returned (in the case of pointers) or an exception is thrown (in the case of references)

Polymorphy (2)

```
[AB.h]
class A {
public: A();
        virtual void foo();
};

class B : public A {
public: B();
        virtual void bar();
};
```

```
[x.cc]
#include "AB.h"
void someFunction() {
  A * a = new B();
  // a->bar(); // ERROR
  B * b = dynamic_cast<B*>(a);
  if (b != NULL) {
    b->bar();
  }
}
```

- You can assign a pointer of type `B*` (pointer to B) to a variable `a` of type `A*`, because `B` is a subclass of `A`
- As C++ has static type checking, you are only allowed to access fields of `A` through `a` (Generally, the compiler can not know that it contains a pointer to a `B` at the moment)
- To access fields of `B` you must dynamically cast `a` to `B*`
- Dynamic means that it is checked at **runtime** that `a` indeed points to a `B` as you claim in the cast, otherwise `NULL` is returned

C++ Strings: Character Arrays

- There is no string base type in C++, so arrays of characters, or char*s, are used instead
  - A "string" contains all of the characters that make it up, followed by `'\0'` to denote the end of the string.
  - `'\0'` is the ASCII NUL character which has a decimal value of 0, as opposed to the ASCII '0' character which has a decimal value of 48
  - Declaring a string:
    `const char* myString = "Hello, this is a string.";`
  - This creates an array of 25 chars: 24 for the characters above, and one at the end with the value of '\0' to denote the end of the string
  - Since strings are arrays, you cannot concatenate them using `+` or compare them using `==`
  - Instead, there are special functions to perform these tasks in the standard C library
- The C++ standard library has a string class (std::string) that can make your life a lot easier

Enumerated Types

- C++ allows to define enumerated types using the enum keyword
- Enumerated types are sometimes useful for expressing a value that has a limited range
- For example, the following code creates an enum for the life cycle of a caterpillar:

```
enum CatLifeCycleType
{
   LARVA,
   CATERPILLAR,
   PUPA,
   BUTTERFLY
};
```

- You can now create a variable of type CatLifeCycleType and assign to it values such as LARVA or PUPA

© Prof. G. Schäfer

The Keyword “const”

- In C++, the keyword `const` means different things according to its context:
  - When you add `const` in front of a variable, it means that variable is treated like a constant. You will not be able to change the value of a `const` variable once you assign it:

    ```
    const float PI = 3.14156;
    ```
  - If an object is declared as `const`, then only the `const` functions may be called
  - If `const` is used with a member function, that means the function can only be called for const objects
  - Parameters in a function may be declared const, which means that those parameters will not be changed during the function call:

    ```
    int multiply(const int a, const int b) { return a*b; }
    ```
  - This means that during this function, the parameters `a` and `b` will be treated as constants
  - Declaring things that should not change as `const` is good practice

© Prof. G. Schäfer

Standard I/O Library (1)

- C++ allows to make use of the stdio library known from C
- To use it: `#include <stdio.h>`
- The `printf` function is used to send output to your shell, called stdout
  - Unlike many other functions, `printf` takes an arbitrary number of parameters
  - It first takes a string that can contain special characters describing the format of the output
  - Next it takes the variables which are to be printed:

```
printf("Hello world\n");
printf("Hello %d.\n",42);
```

  - leads to:

```
Hello World
Hello 42
```

© Prof. G. Schäfer

Standard I/O Library (2)

- The function `scanf` is used for getting input from the user in a shell
- The `scanf` function works the same way as `printf`, except that you must pass the address of the variable you want it to copy the input into:

```
int x;
scanf("%d", &x ); // note the &x!
printf("Your number was %d\n",x);
```

- You can also get a line of input with the `gets()` function. You have to allocate a buffer in advance for it to copy the input into
- Attention:
  - There are many pitfalls and vulnerabilities resulting from the fact, that stdio functions accept a variable number of arguments
  - A good overview on this can be found in [Sch04]

© Prof. G. Schäfer

Flow of Control & Iteration

- Flow of control constructs (`if, switch`) are very similar in Java and in C++ with one difference:
  - Java has a `boolean` type, and
  - C++ has a `bool` type with implicit conversion that maps any non-zero value to `true` and the value zero to `false`
- Loops in Java and in C++ are practically identical:

  -
    ```
    for(<init counter>; <predicate>; <incr counter>)
       <statement>
    ```

  -
    ```
    while(<predicate>)
        <statement>
    ```

  -
    ```
    do
      <statement>
    while(<predicate>);
    ```
- A predicate is any `bool` expression
- As far as syntax goes, C++ and Java predicates are identical

The Preprocessor (1)

- Before the "real" compiler actually compiles your program, a program called the *preprocessor* processes it:
  - The job of the preprocessor is to do simple text substitutions and the like
  - Preprocessor commands all start with the `#` character
  - The preprocessor allows you to include the contents of one file in another file using the `#include` statement
    - If the name of the file to be included is enclosed in angle brackets `< >`, the compiler will search a standard list of directories
    - You can add entries to this list with the compiler flag `-I`
    - If the name of the file is in quotes `" "`, it will search the current directory for the files in addition to the other directories
  - The `#define` preprocessor directive will tell the preprocessor to do text substitution:
    - `#define PI 3.14159265`
    - The `#define` statement can also do substitution with parameters, allowing to write simple macros

The Preprocessor (2)

- You can use the preprocessor to conditionally compile code:
  - The statements used to do this are #if, #ifdef, or #ifndef as well as a #endif following the conditionally compiled section:

```
#define COMPILE_SECTION
 // ...
#ifdef COMPILE_SECTION
 // some code here
 // this code would be compiled
#endif
```

- Conditional compilation can be used for:
  - avoiding circular includes (see below),
  - writing code for multiple platforms, or
  - optionally showing debugging messages

Circular Include Directives

- In general, things can't be defined twice in C++
- This means that two files cannot include each other:
  - For example, if Foo.h does a `#include "Bar.h"` then Bar.h cannot `#include "Foo.h"`. Even if the restriction on multiple definitions didn't exist, this would clearly lead to infinite recursion
  - Therefore, all header files should contain something like this:

```
[Foo.h]
#ifndef Foo_Header
#define Foo_Header

// put all of the header file in here!!

// remember this endif or you will have big problems!
#endif
```

  - The first time the file is read, Foo_Header isn't defined, so it defines it and reads the code in the header file
  - The second time, Foo_Header will already be defined, so your code will be skipped, making your class declaration defined only once

Forward Declarations

- In order to deal with two classes that actually need to know about each other, forward declarations can be very useful:
  - It turns out that you don't need to know anything about a class other than its name in order to declare a pointer to it
  - So, if a class contains a pointer to another class, instead of including the first class' header in the second's header, simply make a forward declaration, and include the both header files in the .cc file:

```
[Foo.h]
#ifndef FOO_H_ALREADY_INCLUDED_
#define FOO_H_ALREADY_INCLUDED_
class Bar;  // forward declaration; says: "class Bar exists,
            // but we don't know anything about it"
class Foo { // ...
protected:
   Bar* m_bar; // We can declare a pointer to a bar. We can't
               // call any methods or declare a non-pointer
               // bar until we include its header file.
};
#endif
```

Build Process (1)

- For each .cc file in your project, the preprocessor copies in all the .h files that are #included and creates a large temporary files
- Each expanded temporary file (from the preprocessor) is compiled into an object (.o) file, which contains the methods and data in the .cc file in a format that can be executed
  - There is also a table of symbols (variables and functions) that are referenced but not defined in this particular .o file. (e.g. C Library functions, stuff from the support code, et cetera)
- In order to resolve all of these symbol references, the final step is to link the .o files together into an executable
  - The linker needs all of the .o files and any external libraries (e.g. libcs123.so) that have any referenced but unresolved symbols
  - The executable is basically a concatenation of your .o files with some information about external libraries

Build Process (2)

- When you run your program, any external libraries that were linked in dynamically will be dynamically loaded
- In order to properly build a program, you need to:
  - find out which files need to be recompiled,
  - build them, and then
  - rebuild your linked object (executable)

  every time you make any modifications to source
- To make this process easier, the tool `make` can be used
  - This tool needs a "description" of the program, called *Makefile,* listing the program's components together with the dependencies that exist between them, in order to automate the build process
  - In this course, the Makefiles will be provided to you (in fact, they will also be generated by yet another tool)

© Prof. G. Schäfer

Additional References

[Agg00] N. Aggarwal. *Java Compared to C++.* JAMM Consulting, Inc., Plano, Texas, USA, 2000. http://www.jammconsulting.com/ Presentations/JavaComparedToC++.ppt

[CS123] CS123 TA Staff. *Java to C++ Transition Tutorial.* Department of Computer Science, Brown University, Providence, Rhode Island, USA. http://www.cs.brown.edu/courses/cs123/javatoc.shtml

[Eck00] Bruce Eckel. *Thinking in C++.* electronic version available for free download at: http://mindview.net/Books/books.html#ThinkingInCPlusPlus

[Pry] N. Pryce. *Introduction to C++.* Distributed Software Engineering Group, Department of Computing, Imperial College, London, UK.

[Sch04] G. Schäfer. *Security Aware System Design & Implementation.* Chapter 4 of the course Protection of Communication Infrastructures, http://www.tkn.tu-berlin.de/curricula/Protection/Handouts/04_System_2Up.pdf

[Str00] B. Stroustrup. *The C++ Programming Language.* 3rd edition, Addison-Wesley, 2000.

[Wei03] M. A. Weiss. *C ++ for Java Programmers.* Addison-Wesley, 2003.

© Prof. G. Schäfer