Advanced Software Engineering with C++ Templates

Lecture 5: Separate Compilation and Templates III

Thomas Gschwind <thg at zurich dot ibm dot com>
Agenda

- Memory Organization & Management
  - Allocation &Deallocation
  - Stack
  - Variables (Pointers, Arrays, References)
  - Heap

- Classes – Fallacies and Pitfalls
Memory Management: Java vs. C++

- **In Java**
  - Built-in types are stored on the stack
  - User-defined types are always stored on the heap (only their references are stored on the stack)
  - Memory is automatically freed by the garbage collector

- **In C++**
  - Built-in types and objects can be stored on the stack and heap
  - C++ supports references and pointers; both can refer/point to objects and built-in types both on the stack and the heap
  - Stack memory is managed automatically
  - Heap memory needs to be freed explicitly (with delete and delete[])

Th. Gschwind. Fortgeschrittene Programmierung in C++. 3
Functions: Parameter Passing

- **Call by Value**
  - Argument to be passed copied from the caller’s scope into the callee’s scope
  - Callee operates on its own copy

- **Call by Reference/Pointer**
  - Callee receives a reference to the argument passed by the caller
  - Callee operates on caller’s copy

- **C++/C**
  - All parameters can be passed by value, by pointer, or in C++ also by reference

- **Java**
  - Primitive types are passed by value
  - Class instances are passed by reference

- **C#**
  - Value types (primitives and structs) by value or reference
  - Class instances by reference
Memory Organization

- The memory needs to store different things
  - The program
  - Global variables
  - Local variables (on the stack)
  - Etc.
The Stack

- Stores local variables, return addresses, etc.
- Right side shows the stack after fact(2) has been invoked

```cpp
int fact(int n) {
    int m = n - 1;
    if (n <= 2) return n;
    else return n * fact(m);
}

int main() {
    cout << "fact(4) = " << fact(4) << endl;
    return 0;
}
```

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
<th>Function State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0fd0</td>
<td>n == 2, m == 1</td>
<td>fact(2)</td>
</tr>
<tr>
<td>0x0fd8</td>
<td>&quot;line 4&quot;</td>
<td></td>
</tr>
<tr>
<td>0x0fdc</td>
<td>result</td>
<td></td>
</tr>
<tr>
<td>0x0fe4</td>
<td>n == 3, m == 2</td>
<td>fact(3)</td>
</tr>
<tr>
<td>0x0fe8</td>
<td>&quot;line 4&quot;</td>
<td></td>
</tr>
<tr>
<td>0x0fec</td>
<td>result</td>
<td></td>
</tr>
<tr>
<td>0x0ff0</td>
<td>n == 4, m == 3</td>
<td>fact(4)</td>
</tr>
<tr>
<td>0x0ff8</td>
<td>&quot;line 8&quot;</td>
<td></td>
</tr>
<tr>
<td>0x0ffc</td>
<td>result</td>
<td></td>
</tr>
<tr>
<td>0x1000</td>
<td>main() - code</td>
<td></td>
</tr>
</tbody>
</table>

Th. Gschwind. Fortgeschrittene Programmierung in C++. 6
Pointers (TYPE*)

- Pointers are a fundamental concept of C++
- However, pointers should be used
  - Sparsely and
  - Carefully
- A pointer points to a value or object stored anywhere in memory
- Since pointers can point to different TYPES of values, there are different types of pointers denoted by TYPE*
- A pointer is similar to an iterator iterating over a collection of elements of type TYPE
Pointers (cont’d)

- Pointers store an address
- This is visible to the developer
- The type of the pointer indicates the type of object stored at the address

```c
int main() {
    int a=3;
    int b=5;
    int *pa;
    pa=&b;
    pa=&a;  // ok
    *pa=9;  // ok
}
```

```
<table>
<thead>
<tr>
<th>Address-Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0ff4: pa==0x0ff8</td>
</tr>
<tr>
<td>0x0ff8: b==5</td>
</tr>
<tr>
<td>0x0ffc: a==9</td>
</tr>
<tr>
<td>0x1000: main()</td>
</tr>
</tbody>
</table>
```
Pointer Operations

- **&-Operator**
  - Obtain the address where a value is stored in memory (lvalue)
  - Returns a pointer to the type of the lvalue
  - & (lvalue)

- ***-Operator**
  - Dereference a pointer (i.e., manipulate the memory that the pointer points to)
  - Return the type of the value the pointer points to
  - *(expr)

An lvalue (left value) is an expression that can occur on the left or the right hand side of an assignment expression. An rvalue is an expression that can only occur on the right hand side.
Modifying Arguments with Pointers

- Implement a routine that exchanges the value of two arguments
- Arguments must be an lvalue
- Invoked with `c_swap(&var1, &var2);
- In Java, this is impossible, arguments need to be wrapped within an object

```c
void c_swap(int *a, int *b) {
    int c=*a;
    *a=*b;
    *b=c;
}
```

```c
int x=3, y=5;
c_swap(&x, &y);
```
Arrays

- Arrays provide memory for several values of the same type
- In C++ an array is typically equivalent to a pointer of the first element of the array

```c
void foo() {
    int buf[4];
    int *bufp=buf;    // ok
    buf[0]=3;         // ok
    *buf=3;          // same
    *bufp=*buf;      // same
    ++bufp;
    *bufp=*buf+1;
}
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0fe8:</td>
<td>bfp==0x0ff0</td>
</tr>
<tr>
<td>0x0fec:</td>
<td>buf[0]==3</td>
</tr>
<tr>
<td>0x0ff0:</td>
<td>buf[1]==4</td>
</tr>
<tr>
<td>0x0ff4:</td>
<td>buf[2]==?</td>
</tr>
<tr>
<td>0x0ff8:</td>
<td>buf[3]==?</td>
</tr>
<tr>
<td>0x0ffc:</td>
<td>“line 8”</td>
</tr>
<tr>
<td>0x1000:</td>
<td>main()</td>
</tr>
</tbody>
</table>
Arrays (Be Careful)

- Arrays are not range checked (buffer-overflow)
- C++ happily assigns a value to buf[4]
- Developer has to keep track of this
- There are several libraries providing a safe version of arrays (we’ll come back to this)
- Arrays “cannot” be returned as the result of a function
The Heap

- Stores non-local and non-global variables
- If memory needs to be allocated during runtime (e.g., linked lists, large arrays, etc)
- Memory needs to be explicitly allocated (with new like in Java)
- Useful for returning arrays from routines
- Useful for returning large user-defined types from routines
Memory Allocation & Deallocation

- "No" garbage collection in C/C++
  - Memory needs to be allocated explicitly (`new`, `new[]`) and freed (`delete`, `delete[]`)
  - Some lazy and/or old-fashioned C programmers still use `malloc` when they want uninitialized memory
  - The proper way of allocating uninitialized memory, however, is `::operator new (size_t bytes)`

- Error handling
  - `bad_alloc` will be thrown (`new ...`),
  - or `NULL` is returned (`new (nothrow) ...`)

- Initialization

"Unfortunately, overuse of new (and of pointers and references) seems to be an increasing problem."

Bjarne Stroustrup
Returning an Array

- An array per se is a pointer
  - Cannot return the entire array
  - Can only return the pointer

```cpp
int* foo2(int n) {
    int *buf=new int[n];
    for(int i=0; i<n; ++i) {
        buf[i]=i*i;
    }
    return buf;
}

void main() {
    int *buf=foo2(10);
    for(int i=0; i<10; ++i) {
        cout << buf[i] << endl;
    }
    delete[] buf;
}

Th. Gschwind. Fortgeschrittene Programmierung in C++.
Returning an Array (Style!)

- The example has a style problem
- Can you spot it?
  - Memory is allocated in foo2 and deallocated in main
  - If possible one should deallocate memory in the same method as it is allocated
- What are the alternatives?
  - Allocate the array in main and pass it into foo2
  - If that is not possible implement a foo2_cleanup routine that deallocates the memory returned by foo2
Returning an Array with Nice Memory Management

- Frequently, when a function \( f \) allocates memory for us, there is a sister function \( f' \) that releases the memory allocated by \( f \)

```cpp
int* new_foo2(int n) {
    int *buf=new int[n];
    for(int i=0; i<n; ++i) {
        buf[i]=i*i;
    }
    return buf;
}

void delete_foo2(int *buf) {
    delete[] buf;
}

void main() {
    int *buf=new_foo2(10);
    for(int i=0; i<10; ++i) {
        cout << buf[i] << endl;
    }
    delete_foo2(buf);
}
```
Garbage Collection

- C++ does not perform garbage collection
- BUT you can make it perform garbage collection for you
  - auto_ptr (in C++11: unique_ptr, same thing, a bit more efficient)  
    (simple solution – deallocate element when auto_ptr is deallocated)
  - smart_ptr (see exercises)

```cpp
void foo() {
    auto_ptr<int> pint(new int);
    *pint=17;
    cout << "*" << pint.get() << "=" << *pint << endl;

    auto_ptr<int> qint(pint); // pint points to NULL
    *qint=19;
    cout << "*" << pint.get() << endl; // displays 0
    cout << "*" << qint.get() << "=" << *qint << endl;
} // deallocate the integer
```
C++11: Memory Management

- Yes, memory management is your responsibility in C++
- The following “pointers” provided by the Standard Library help
  - `unique_ptr` (former `auto_ptr`)
    Delete object pointed to when `ptr` is destructed
  - `shared_ptr`
    Delete object pointed to by `shared_ptr` when this is the “last” pointer pointing to the object – works similar to the `smart_ptr` exercise
  - `weak_ptr` useful in combination with `shared_ptr` if cyclic structures are used
    Allows to break up cycles, need to be converted to `shared_ptr` before object may be accessed, see documentation for details
- Don’t use as all-round solution
  - For instance only if your object is really shared
C++11: Returning an Array

- unique_ptr and shared_ptr can be used to return objects
- The pointers will destruct the object when “they” no longer point to it

```cpp
unique_ptr<int[]> make_foo3(int n) {
    unique_ptr p{new int[n]};
    for(int i=0; i<n; ++i) {
        p[i]=i*i;
    }
    return p;
}

void main() {
    auto buf=make_foo3(10);
    for(int i=0; i<10; ++i) {
        cout << buf[i] << endl;
    }
}
```
References (TYPE&)

- Similar to pointers – sometimes more elegant
- However, like pointers they should be used
  - Sparsely and
  - Carefully
- A reference refers to a value or object stored in memory, it is another name (alias) for a given value or object stored in memory
- Unlike a pointer, it cannot be changed to refer to a different location
References (cont’d)

- References are similar to pointers
- Except their implementation is “invisible”

```c
int main() {
    int a=3
    int b=5;
    int &pa=a;
    pa=7;   // ok
    //&pa=b;   // error
    pa=b;   // ok
}
```

```
Address-Space

0x6f00:
0x6f04: b==5
0x6f08: pa/a==5

main()
```

Th. Gschwind. Fortgeschrittene Programmierung in C++.
Modifying an Argument with References

- Implement a routine that exchanges the value of two arguments
- Argument must be an lvalue
- Invoked with swap(var1,var2);
- Similar to VAR parameter in Pascal
- In Java, this is impossible, in C#, however, it is possible

```c
void swap(int &a, int &b) {
    int c=a;
    a=b;
    b=c;
}
```
Advantage, function can be used as lvalue
What does this mean?

Be careful! Don’t insist on using references

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

May also be used as return value of a function
Electric Fence / DUMA

- Bugs with pointers may cause a program to fail at a much later state.
- Tools have been developed to make programs with sloppy memory management fail early.
  - Electric Fence: The original library to do this.
  - DUMA: Detect Unintended Memory Access.
  - You will learn to love those programs.

How do you use it?
- Download from sourceforge, compile.
- Link your programs with -Lpath/to/duma -lduma -lpthread.

How does it work?
- Make sure that before and after every allocated block there is an unmapped memory block.
- After memory is freed, unmap the memory block.
- Accesses to unmapped memory trigger a segmentation fault.

Only use these tools for debugging/testing.

Th. Gschwind. Fortgeschrittene Programmierung in C++. 25
Agenda

- Memory Organization & Management
  - Allocation & Deallocation
  - Stack
  - Variables (Pointers, Arrays, References)
  - Heap

- Classes – Fallacies and Pitfalls
Classes – Fallacies and Pitfalls

- Example: Our own `String` class
- Represent strings without length-limit
  - => need to manage the string’s memory
- Want to be able to concatenate string
- Access a character at a given position x of the string
String: Naïve Solution

```cpp
class String {  // type declaration+definition
private:
    char *strg; unsigned int len;

public:
    String(const char *strg=NULL) {
        if (!strg) {
            this->strg=NULL; len=0;
        } else {
            len=strlen(strg);
            this->strg=(char*)malloc((len+1)*sizeof(char));
            memcpy(this->strg, strg, (len+1)*sizeof(char));
        }
    }
    ~String() { if(strg) free(strg); }

    // ...
};
```

Th. Gschwind. Fortgeschrittene Programmierung in C++.
**String: Naïve Solution**

```cpp
class String {
    // type declaration+definition

    // ... 

    String operator+(String b) {
        String r;
        r.len = this->len + b.len;
        r.strg = (char*)malloc((r.len+1)*sizeof(char));
        memcpy(r.strg, this->strg, this->len*sizeof(char));
        memcpy(r.strg+this->len, b.strg, (b.len+1)*sizeof(char));
        return r;
    }

    char &operator[](unsigned int idx) {
        return strg[idx];
    }
};
```

Th. Gschwind. Fortgeschrittene Programmierung in C++.
String: Naïve Solution

- A small test driver to test each function we implemented
  - Constructor by creating a string
  - Read and write (we return a reference) a character with operator[]
  - Concatenate two strings with operator+

```cpp
#include <iostream>
#include "String.h"

int main(int argc, char *argv[]) {
    String a("hello"), b(" "), c("World"), d("!"), e;
    a[0] = 'H';
    e = a + d;
    cout << e << endl;
    cout << "e[0]=" << e[0] << endl;
}
```

Th. Gschwind. Fortgeschrittene Programmierung in C++.
The Problem: \( e = a + d \);

1. The string given as parameter to operator+ is passed by value
   - A copy is created on the stack
   - Hence, the copy constructor is invoked creating a shallow copy

2. The string created by operator+ is returned by value
   - The return value is copied from \( r \) into the caller’s stack
   - Hence, the copy constructor is invoked creating a shallow copy

```cpp
String operator+(String b) {
    String r;
    r.len = this->len + b.len;
    r.strg = (char*)malloc((r.len+1)*sizeof(char));
    memcpy(r.strg, this->strg, this->len*sizeof(char));
    memcpy(r.strg+this->len, b.strg, (b.len+1)*sizeof(char));
    return r;
}
```

Th. Gschwind. Fortgeschrittene Programmierung in C++.
Constructors

- **Default Constructor** (`T()`)
  - Created by the compiler, if not defined by yourself
  - Initializes attributes with default constructor

- **Copy Constructor** (`T(const T&)`)
  - Created by the compiler, if not defined by yourself
  - Copies each attribute from the source to the target object ("shallow copy")
  - This constructor is executed frequently
    - Whenever a parameter is passed by value
    - Whenever an object is returned from a function
      (not if a pointer or reference to an object is returned)
String: Copy Constructor

- A simple constructor allocates memory and copies the string
- A more sophisticated `String` implementation
  - Uses reference counting and a copy-on-write strategy
  - A yet more sophisticated implementation copies short strings and uses reference counting for long strings

```cpp
String(const String &s) {
    if(!s.strg) {
        len=0; this->strg=NULL;
    } else {
        len=s.len;
        this->strg=(char*)malloc((len+1)*sizeof(char));
        memcpy(this->strg, s.strg, (len+1)*sizeof(char));
    }
}
```
Another Problem: \( e = a + d; \)

- The result of \( a + d \) is a String object
- The returned String is assigned to \( e \)
  - Hence, the assignment operator= is invoked
  - Like the copy constructor, the default assignment operator performs an element-wise or shallow copy
- The returned String is destructed after the assignment
  - Its destructor is called which frees the String’s memory
  - Due to the shallow copy from above this is the same memory as that of \( e \)
- The String \( e \) is destructed when main returns
  - Its destructor is called which frees the String’s memory
  - Now, most memory manager’s realize there is something wrong
String: Assignment Operator

- If this object stores already a string we need to deallocate it
- If the string to be assigned is not NULL we allocate memory
- Copy the string’s memory

```cpp
String &operator=(const String &s) {
    if (strg) free(strg);
    if (!s.strg) {
        len=0; this->strg=NULL;
    } else {
        ...
    }
    return *this;
}
```
Problems Solved?

- Our previous test driver now works perfectly fine
- Now let us extend our test driver with a redundant line of code

```cpp
int main(int argc, char *argv[]) {
    String a("hello"), b(" "), c("World"), d("!"), e;
    a[0]='H';
    e=a+d;
    e=e;
    cout << e << endl;
}
```

- Shouldn’t cause any problems?
- Well, what’s the first thing our assignment operator does?
String: Fixed Assignment Operator

- First, check if this is a self-assignment
- If this object stores already a string we need to deallocate it
- If the string to be assigned is not NULL we allocate memory
- Copy the string’s memory

```cpp
String &operator=(const String &s) {
    if(this==&s) return *this;  // don’t forget this!
    if (strg) free(strg);
    if (!s.strg) {
        len=0; this->strg=NULL;
    } else {
        ...
    }
    return *this;
}
```
Programming Style

- Repetition between
  - Constructor
  - Copy constructor
  - Assignment operator
  - They all assign a new value to the current object

- And between
  - Assignment operator
  - Destructor
  - Both can free a previously stored string
Helper Functions

```cpp
class String {
private:
    void init(const char *s) { /* allocate and copy string */ }
    void clear() { /* free memory occupied by string */ }

public:
    String(const char *strg) { init(strg); }
    String(const String &s) { init(s.strg); }
    ~String() { clear(); }

    String &operator=(const String &s) {
        if (this == &s) return *this; // don’t forget this!
        clear(); init(s.strg);
        return *this;
    }

    // ...

    Using helper functions makes the code tidy and easily readable.
}
```

Th. Gschwind. Fortgeschrittene Programmierung in C++.
Guidelines

- Don’t forget the Copy Constructor
- Don’t forget the Assignment Operator
  - Check whether source and target are the same
- Don’t forget the “Rule of Three”

- Avoid to unnecessarily copy large data types and structures
  - If possible, use “const T &foo” instead of “T foo” if T can be more than a couple of bytes/attributes
- Use helper functions
  - If you are concerned about performance, make them inline functions
C++11: Move constructor

- Remember our String example?

```cpp
#include <iostream>
#include "String.h"

int main(int argc, char *argv[]) {
    String a("Hello"), b(" "), c("World"), d("!"), e;
    e = a + d;
    cout << e << endl;
    cout << "e[0]=" << e[0] << endl;
}
```

- The Copy Constructor is invoked to pass the data from the operator+ scope into the caller’s scope
- Then the assignment operator is invoked
- With our String implementation we copy the string twice (Fortunately, most Standard Library String implementations avoid this by reference counting)
C++11: Move constructor (cont’d)

- **String(String &&s)**
  - Steals the resources of s
  - we know that s will be destroyed after the invocation of String(String &&s)
  - Good for return values
  - If not provided, the copy constructor will be invoked

```cpp
class String {
public:
    String(String &&s) {
        this.strg = s.strg;
        this.len = s.len;
        s.strg = NULL;
        s.len = 0;
    }

    String&&(operator+(String b) { ... })
};
```

Here, we steal the resources from s

Indication, that the resources of the return value may be stolen

Implementation is same as before
Summary

- Memory Organization & Management
  - Allocation & Deallocation
  - Stack
  - Variables (Pointers, Arrays, References)
  - Heap

- Classes – Fallacies and Pitfalls
Exercise 1: Emulating pointers

- Implement a dumb_pointer. Use the operator* and operator-> operators to implement a class that simulates a pointer.
- Implement another class that works like the dumb_pointer but is called smart_pointer and uses reference counting.
- No, using any of the C++ auto_ptr, unique_ptr, shared_ptr, weak_ptr classes is not an implementation option.
void print(smart_pointer<Object> p) { cout << p.counter() << ": " << *p << endl; }

void foo() {
    Object *o1=new Object(1); // let's create our 1st object
    Object *o2=new Object(2); // let's create our 2nd object

    smart_pointer<Object> p(o1); // ref counter is 1 for 1st object
    cout << p.counter() << endl; // displays 1

    smart_pointer<Object> q(p); // another smart pointer that points to o1 (overload copy constructor)
    cout << p.counter() << endl; // displays 2 (two smart pointers refer to o1)
    cout << q.counter() << endl; // displays 2 (two smart pointers refer to o1)

    smart_pointer<Object> r(o2); // ref counter is 1 for 2nd object
    cout << r.counter() << endl; // displays 1

    q=r; // decrease counter for 1st object and
    // increase counter for 2nd object (overload assignment operator)
    cout << p.counter() << endl; // displays 1
    cout << q.counter() << endl; // displays 2
    cout << r.counter() << endl; // displays 2

    print(p); // displays 2, and the 1st object, don't delete the object pointed to by p
    print(q); // displays 3, and the 2nd object, don't delete the object pointed to by q
    print(r); // displays 3, and the 2nd object, don't delete the object pointed to by r

    cout << *p << *r << endl; // display 1st and 2nd object (overload operator*)
    cout << p->method1() << q->method2() << r->method3() << endl;
    // invoke method1 on 1st object and
    // invoke method2 on 2nd object and
    // invoke method3 on 2nd object (overload operator->)
}

} // now the destructors of p, q, and r are called, make sure that 1st
// and 2nd object is each deleted once (i.e., when the counter reaches 0)
Exercise 2: Under the Hood

- Have a look at the two swap routines `swap(int&, int&)` and `c_swap(int*, int*)`
- Let the compiler compile the code but ask the compiler to stop at the assembly stage
  
  ```sh
  $ gcc -S -o source.s source.cc
  ```
- Compare the assembly code, what do you observe?
- How do you interpret the difference?

- Play around with inline functions such as swap and lcm
  - Implement a program with the swap and lcm functions declared inline
  - And another with those functions declared as normal functions
- Compile the two programs and generate assembly code (with and without optimization)
- How many instructions do the different programs have?
- How do you interpret the differences?
Exercise 3

- Improve the computer player of your “Connect 4” implementation and make sure you can accommodate computer players from your colleagues. Get at least 2 computer players from other colleagues and see who is better. In order to be able to do this you need to do the following (see the web for details):
  
  - You need to implement a class religiously implementing the playfield interface that gives the computer player access to the playfield through the stoneat(x,y) function.
  
  - Your computer player needs to implement the player interface that provides the play(playfield) method. In case the functionality provided by the playfield interface is not sufficient, your player needs to convert the playfield into your own representation.
  
  - Make sure that your program works in such a way that two computer players can play against each other.
  
  - Do not change the playfield or player interfaces or otherwise your implementation will not work with your colleagues’ implementations.
Next Lecture

- Inheritance
- Liskov Substitution Principle

Have a nice weekend, see you next week