C++ Programming

Lecture 6

Software Engineering Group

Philipp D. Schubert
Common mistakes

- Do not confuse between (de)allocation operators

  ```cpp
  void* operator new ( std::size_t count ); // for objects
  void operator delete ( void* ptr ); // for objects
  void* operator new[]( std::size_t count ); // for arrays
  void operator delete[]( void* ptr ); // for arrays
  ```

- Do not introduce variables before needed

  ```cpp
  int i; for (i = 0; i < x; ++i) versus for (int i = 0; i < x; ++i)
  ```

- Index out of bounds

  ```cpp
  vector<int> v = {1, 2, 3};
  for (int i = 0; i < v.size(); ++i) {
      if (v[i] < v[i+1]) { ... }
  }
  .size() versus sizeof()
  ```

- Pointer arithmetic

  ```cpp
  int *array = new int[10]; array[0] = 42; *(array + 1) = 42; delete[] array;
  ```

- Copy/move assignment operator

  ```cpp
  Variables have already been allocated
  ```
Contents

1. C/C++ preprocessor
2. Templates
3. Variadic function arguments
4. Functors
5. Lambda functions
The C/C++ preprocessor: `cpp`

- Remember the `#` directives
- Preprocessor directives are executed by the preprocessor
- `cpp` is just a text processing program
- Pre because it processes the program text before the compiler gets the text
  - Great power
  - Great opportunity to introduce really subtle bugs
- Do not overuse/misuse the preprocessor
  - Only use in rare cases
  - C++ has better mechanisms to replace its uses
- In modern C++ it rarely has to be used anymore
  - Except for organizing files: header & implementation files

[Figure taken from “A Tour of C++”, Stroustrup 2013]
The C/C++ preprocessor

- Remember the language processing system
  - In C++ we use module-wise compilation of implementation files (.cpp)
  - Header files (.h) are not compiled at all
    - Header files are only included in implementation files
    - Each implementation file becomes preprocessed and then compiled
- Preprocessor can be considered a second language → performs simple text processing/replacement
- Problem: a programmer has to deal with 2 languages!
  - Template metaprogramming is a 3rd language (inside C++)
  - Template metaprogramming is even Turing complete
    - TMP next time
- Why use such a complicated system for modularity?
  - Compatibility: C had this system, so C++ adopted it
  - (Java-like) Module-System planned for C++20

[Figures taken from "A Tour of C++", Stroustrup 2013]
CPP directives

- Important CPP directives
  - `#include`
  - `#define`
  - `#if`
  - `#elif`
  - `#endif`
  - `#else`
  - `#ifdef`
  - `#ifndef`
  - `##` // a dark corner of CPP
  - Variadic function arguments (macros) // another dark corner of CPP

[Figure taken from https://c1.staticflickr.com/3/2117/1792490622_5de0a1d609_b.jpg]
CPP directives

- `#include` header files

  Example

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Hello!\n";
    return 0;
}
```

- `#define` symbols

  Example

```cpp
#include <iostream>
#define VALUE 10
using namespace std;

int main() {
    cout << VALUE << '\n';
    return 0;
}
```
CPP directives

- **#ifdef example**

```cpp
#include <iostream>
// this is one way to define a symbol
#define LINUX
using namespace std;
int main() {
  #ifdef LINUX
    cout << "is linux\n";
  #elif WINDOWS
    cout << "is windows\n";
  #elif MAC
    cout << "is mac\n";
  #else
    cout << "is something else\n";
  #endif
  return 0;
}
```

- Another way to define a symbol
  - Tell the compiler directly
    - g++ -DMAC test.cpp -o test
  - Program produces:
    - "is mac"
  - g++ test.cpp -o test
  - Program produces:
    - "is linux"
  - Use option –U to undefined a symbol
  - Remember preprocessing-only action can be done by calling the **cpp** program
    - Use option –E or –P (or both)
CPP directives

- #if example

```c++
#include <iostream>
#define VALUE 10
using namespace std;

int main() {
    #if VALUE == 10
        cout << "VALUE is 10\n";
    #else
        cout << "VALUE is not 10\n";
    #endif
    return 0;
}
```
CPP: Software product lines

- Using \#if, \#ifdef, \#elif, \#else, \#endif ...
  - one can build software product lines
  - maintain huge software product lines within same codebase
  - Linux kernel has many \#if directives
    - Can be compiled for many platforms
  - Almost all big software projects use the preprocessor to establish SPL’s
  - Configuration for CPP symbols are usually maintained in a huge (documented) table
    - Symbols are looked up for product needed
    - Car system software for model XY of brand Z
      - Use corresponding CPP symbols for XY and compile
      - If software for another model is needed
        - Use the appropriate CPP symbols
... out of few components
Feature model restricts valid configurations
Finding bugs & testing SPL’s

- It is like hell!
- It is just a program?
  - No!
  - Every use of `#if` or `#ifdef` leads to two different programs
    - One program that contains the code “inside” `#if`
    - One program that does not contain the code “inside” `#if`
- There are $2^n$ different versions of your program
  - $n$ is the number of preprocessor `#if`s used
- What does that mean?
#ifdef occurrences in OpenSSL

391 different Features $\Rightarrow 5 \cdot 10^{117}$ Combinations
Observable Universe: roughly $10^{80}$ Atoms

OpenSSL: $5 \cdot 10^{117}$ Combinations
CPP macros

* Writing a macro

```cpp
#include <iostream>
#define FACTORIAL(X,R) {
    R = 1;
    while (X > 0) {
        R *= X;
        --X;
    }
}

using namespace std;
int main() {
    int i = 5;
    int result;
    FACTORIAL(i, result);
    cout << result << '
';
    return 0;
}
```

* Do not abuse macros for function inlining!
* This leads to subtle bugs
  * Compiler only sees the expanded macro
  * Programmer sees the unexpanded one
  * Type information is missing
* Use C++ keyword `inline` instead
* Macro expands to:

```cpp
// Expanded code
using namespace std;
int main() {
    int i = 5;
    int result;
    // Code inside FACTORIAL
    cout << result << '
';
    return 0;
}
```
Macros gone wrong

- Very subtle bugs

```cpp
#include <iostream>
using namespace std;
#define ABS(X) ((X) < 0) ? -(X) : X;

int main() {
    int i = ABS(-42);
    int j = ABS(42);
    int t = 10;
    int k = ABS(--t);
    cout << i << 'n';
    cout << j << 'n';
    cout << k << 'n';
    return 0;
}
```

- What will be printed?

```cpp
using namespace std;
int main() {
    int i = (-42 < 0) ? -42 : -42;;
    int j = (42 < 0) ? -42 : 42;;
    int t = 10;
    int k = (--t < 0) ? --t : --t;;
    cout << i << 'n';
    cout << j << 'n';
    cout << k << 'n';
    return 0;
}
```

- Output: 42, 42, 8
Predefined CPP macros

- Predefined macros

```cpp
#include <iostream>
using namespace std;

int main() {
    #define UNRECERROR(BOOL_EXPR, MESSAGE) {
        if (BOOL_EXPR) {
            cout << "Detected unrecoverable error: " << MESSAGE << 'n';
            cout << "File : " << __FILE__ << 'n';
            cout << "Date : " << __DATE__ << 'n';
            cout << "Time : " << __TIME__ << 'n';
            cout << "Line : " << __LINE__ << 'n';
            cout << "ANSI : " << __STDC__ << 'n';
            exit(-1);
        }
    }

    UNRECERROR(1, "uups");
    return 0;
}
```

- Output

```plaintext
File : cpp.cpp
Date : Nov 25 2016
Time : 09:09:16
Line : 8
ANSI : 1
```
Notes on the preprocessor

- Include mechanism is fine (essential even)
  - Header file
    ```
    #ifndef MODULE_H_
    #define MODULE_H_
    // ... some code
    #endif
    ```
  - Implementation file
    ```
    #include <iostream>
    #include "MyFile.h"
    ```
- Other than that, try to avoid using CPP's
  ```
  #define
  #if
  ```
  as much as possible
Templates

- Templates allow abstraction from a concrete type
- Templates are handled by the compiler, not the CPP
  - Compiler can use the type system
- “A template is a C++ entity that defines one of the following:
  1. a family of classes/structs (class template), which may be nested classes
  2. a family of functions (function template), which may be member functions
  3. an alias to a family of types (alias template)
  4. a family of variables (variable template)
- Templates are parameterized by one or more template parameters of three kinds:
  1. type template parameters
  2. non-type template parameters

- If template parameters are specified (template instantiation) one obtains a template specialization
Templates

- Templates are code generators (compare to C’s abs implementation: \texttt{fabs}, \texttt{fabsf}, \texttt{fabsl}, \texttt{abs}, \texttt{labs}, \texttt{llabs})

- Caution template code cannot be compiled up-front
  - A concrete template instantiation is required
  - Put template code into header files
  - Include them where you need the template code

- Always prefer templates over some macro directives

- Use templates …
  - if you do not know information up-front
  - if a class/function/variable can be used more general (like sorting – in the exercises)

- Can be abused to compute values at compile time: \textit{template meta-programming} (next lecture)
  - Since C++11/C++14 prefer \texttt{constexpr} functions
  - But not everything can be expressed using \texttt{constexpr} functions
# include <iostream>
# include <type_traits>
using namespace std;

template<typedef A, typedef B>
struct Tuple {
  A first;
  B second;
  Tuple(A a, B b) : first(a),
          second(b) {} }

template<typename T>
T add(T a, T b) {
  static_assert(is_arithmetic<T>::value,
            "wrong");
  return a + b;
}

int main() {
  Tuple<int, string> t(5, "Hello!");
  cout << t.first << '
';
  cout << t.second << '
';
  int result_int = add<int>(5, 5);
  double result_double = add<double>(1.11,
                            2.22);
  double result_double_deduced = add(3.33,
                    4.44);
  cout << result_int << '
';
  cout << result_double << '
';
  cout << result_double_deduced << '
';
  return 0;
}
Variable template

- Implement your own safe static array type (carrying size)

```cpp
#include <iostream>
#include <algorithm>
#include <cassert>
using namespace std;
template<typename T, size_t S>
class MyArray {
private:
    static constexpr size_t elements = S;
    T data[S];

public:
    MyArray() {}
    MyArray(T ival) { fill(&data[0], &data[elements], ival); }
    size_t size() const { return elements; }
    T& operator[](size_t idx) {
        assert(idx < elements);
        return data[idx];
    }
};
```

const T& operator[](size_t idx) const {
    assert(idx < elements);
    return data[idx];
}

friend ostream& operator<<(ostream& os, const MyArray& a){
    for (size_t i = 0; i < a.elements; ++i)
        os << a.data[i] << " ";
    return os;
}

int main()
{
    MyArray<int, 10> a;
    MyArray<double, 5> b(1.11);
    cout << a << 'n';
    cout << b << 'n';
    return 0;
}
```

- Note: only integer types are allowed for variable templates
C++’s oddities

```cpp
template<typename T>
class MyClassTemplate {
    // ...
};

versus

template<class T>
class MyClassTemplate {
    // ...
};
```

“Summary: Stroustrup originally used class to specify types in templates to avoid introducing a new keyword. Some in the committee worried that this overloading of the keyword led to confusion. Later, the committee introduced a new keyword typename to resolve syntactic ambiguity, and decided to let it also be used to specify template types to reduce confusion, but for backward compatibility, class kept its overloaded meaning.”

[http://stackoverflow.com/questions/213121/use-class-or-typename-for-template-parameters]
Variadic function arguments

- More flexibility for function parameters
- How can we pass arbitrary many parameters to a function?
  ```cpp
  void print_ints(int i) { cout << i << '\n'; }
  void print_ints(int i, int j) { cout << i << " " << j << '\n'; }
  ```
- Do not use C-style vararg macros!
- Better solution
  - Use `std::initializer_list`
    ```cpp
    void print_ints(initializer_list<int> varargs);
    ```
  - The user can pass arbitrary many integers to `print_ints()`
  - But caution
    - You have to pass arguments using the `initializer_list`'s curly braces `{ }`
Variadic function arguments

```cpp
#include <iostream>
#include <initializer_list>
using namespace std;

void print_ints(
    initializer_list<int> args) {
    for (int i : args) {
        cout << i << " ";
    }
}

template<typename T>
void print_generic(
    initializer_list<T> args) {
    for (const T &arg : args) {
        cout << arg << " ";
    }
}

int main() {
    print_ints({1, 2, 3, 4});
    print_ints({1, 2});
    print_generic({1.111, 2.222, 3.333});
    return 0;
}
```

- Use `initializer_list` for variable argument list
  - Cleanest way to achieve flexibility
- Another way is possible
  - C-style varargs (involves preprocessor macros)
    - Hard to read
    - Hard to understand → NO!
- Next time: variadic arguments of different types
  → involves template meta programming
Function object or functor

- A functor is a class or struct that implements `operator()`
  - The (function) call operator
- A variable of that type can be called like a function
  - A functor is a function that can store variables and data
    - It has state
    - You can wrap tasks into functors
    - Fits perfectly into object oriented programming and multi-threading (later on)
Functor

Example function object

```cpp
#include <iostream>
using namespace std;

class Task {
private:
    int i;
    int j;
    // perform extensive task of adding
    // two numbers
    int do_hard_work() { return i + j; }

public:
    Task(int i, int j) : i(i), j(j) {}
    int operator() () {
        return do_hard_work();
    }
};

int main() {
    // set up a task
    Task t(100, 200);
    // start solving the task
    int result = t();
    cout << result << '\n';
    return 0;
}
```
Lambda functions

- “A lambda function is an unnamed function object capable of capturing variables in scope”

- Syntax
  1. `[ capture-list ] ( params ) -> ret { body }
  2. `[ capture-list ] ( params ) { body }
  3. `[ capture-list ] { body }

- What is a capture-list?
  - A comma-separated list of zero or more captures
    - `[a,&b]` where `a` is captured by value and `b` is captured by reference.
    - `[this]` captures the `this pointer` by value
    - `[&]` captures all `automatic` variables `odr-used` in the body of the lambda by reference
    - `[]=` captures all `automatic` variables `odr-used` in the body of the lambda by value
    - `[]` captures nothing
Lambda functions

```cpp
#include <iostream>
#include <functional>
using namespace std;

int main() {
    int result = bin_operation(2, 3, [](int a, int b) {
        return a + b;
    });
    int other = bin_operation(2, 3, [](int a, int b) {
        return a * b;
    });
    function<int(int)> f_sq = [](int a) {
        return a * a;
    };
    // or use auto to be shorter
    auto f_sub = [](int a, int b) { return a - b; };
    cout << f_sq(10) << '
';
    cout << f_sub(10, 5) << '
';
    cout << result << '
';
    cout << other << '
';
    return 0;
}

int bin_operation(
    int a,
    int b,
    function<int(int)> f) {
    return f(a, b);
}
```

#include <iostream>
#include <functional>
using namespace std;

int main() {
    int result = bin_operation(2, 3, [](int a, int b) {
        return a + b;
    });
    int other = bin_operation(2, 3, [](int a, int b) {
        return a * b;
    });
    function<int(int)> f_sq = [](int a) {
        return a * a;
    };
    // or use auto to be shorter
    auto f_sub = [](int a, int b) { return a - b; };
    cout << f_sq(10) << '
';
    cout << f_sub(10, 5) << '
';
    cout << result << '
';
    cout << other << '
';
    return 0;
}

int bin_operation(
    int a,
    int b,
    function<int(int)> f) {
    return f(a, b);
}
Lambda functions

- Example using `for_each()` algorithm (which is a fully generic function)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;

int main() {
    vector<int> v = {1, 2, 3, 4, 5, 6};
    for_each(v.begin(), v.end(), [](int &i) -> void { i *= i; });
    for_each(v.begin(), v.end(), [](int i) { cout << i << " "; });
    cout << '\n';
    return 0;
}
```
Lambda functions

- Now we can introduce easy-to-use predicates
- Example binary predicate

```cpp
#include <iostream>
#include <vector>
#include <functional>
using namespace std;

bool check_pred(
    vector<int>& v1,
    vector<int>& v2,
    function<bool(int,int)> pred){
    for (size_t i = 0; i < v1.size(); ++i)
        if (!pred(v1[i],v2[i]))
            return false;
    return true;
}
```

```cpp
int main() {
    vector<int> v1(5, 1);
    vector<int> v2 = {1, 1, 1, 1, 1};
    bool equals = check_pred(v1, v2,
        [](int a, int b) { return a == b; });
    cout << equals << '
';
    return 0;
}
```
Lambda functions

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;

int main() {
    vector<int> from(10);
    vector<int> to(10);
    // fills with 1, 2, ... , 10
    iota(from.begin(), from.end(), 1);
    for_each(from.begin(),
             from.end(),
             [](int i) {
                 cout << i << ' '; }
    );
    cout << '\n';

    int epsilon = 5;
    transform(from.begin(),
              from.end(),
              to.begin(),
              [epsilon](int d) {
                if (d <= epsilon) {
                    return 0;
                } else {
                    return d;
                }
            });
    for_each(to.begin(),
             to.end(),
             [](int i) {
                 cout << i << ' '; }
    );
    cout << '\n';
    return 0;
}
```
Recap

- C/C++ preprocessor
- Modularity in C++
- Software Product Lines (SPL)
- Exponential `#ifdef` variability
- Macros
- Macros gone wrong
- Predefined macros
- Templates
- Variadic function arguments
- Functor: function objects
- Lambda functions
Thank you for your attention

Questions?