CH08-320143

Object-oriented Programming II

OOP II

Lecture 5 & 6

Dr. Kinga Lipskoch

Spring 2018

Agenda Week 3

- An Application: Blackjack
- Debugging Techniques: `gdb`, preprocessor, trace macros, trace files
- Memory Leaks
- Unit Testing
- Recap from "Programming in C II": Makefiles

Putting it All Together (1)

A game - Blackjack:

- Each player tries to get 21 points
- Numbered cards: face value
- Ace counts 1 or 11 points
- Jack, queen, king count 10 points
- Casino (or house) plays against players
- Each player (incl. casino) receives two cards
- One of the casino cards is hidden
- Each player may take additional cards

Putting it All Together (2)

A game - Blackjack:

- If player exceeds 21 points, player is busted
- After all players have taken cards, casino reveals the hidden card and will take additional cards as long as the total is 16 or less
- If casino busts, all players still in the game (not busted) win
- player $>$ casino: player wins
- player $<$ casino: player loses
- player $==$ casino: tie (a.k.a. pushing)

Classes of the Application

Class	Base class	Description
`Card`		Playing card
`Hand`		A hand, collection of cards
`Deck`	`Hand`	A deck, similar to `Hand`, but can be shuffled or dealt
`Generic Player`	`Hand`	Generic Blackjack Player
`Player`	`Generic Player`	Human Player
`House`	`Generic Player`	Computer Player, Casino
`Game`		The game

Card class

- `Card` objects like real-life cards
- `Hand` has a vector of pointers to `Cards`
- `Cards` move from one `Hand` to another

`Player` class

- `Player`
- Blackjack hands with names
- `Player` and `House` are derived from `Hand` via `Generic Player`
- `Generic Player` contains the shared functionalities of `Player` and `House`

Deck class

- Derived from `Hand`
- Deals card to players (either `Player` or `House`)
- Can be implemented by polymorphic function

The Game

- Deal players and the casino (house) two cards
- Hide the house's first card
- Display players' cards and house's other card
- Deal additional cards to players
- Reveal house's first card
- Deal additional cards to house
- Determine the winner
- `blackjack.cpp`

Debugging Programs with gdb

- `gdb` is a powerful text-based interactive debugger
  - Runs a program
  - Stops at predetermined location
  - Displays / sets variables
  - Or executes one line at a time
- GUI (Graphical User Interface) wrapper for `gdb`: Affinic, Code::Blocks, Codelite, DDD, etc.

Debugging with gdb (1)

```
1  #include <iostream>
2  using namespace std;
3
4  int main (void) {
5    const int data[5] = {1, 2, 3, 4, 5};
6    int i, sum;
7
8    for (i = 0; i >= 0; i++) {
9      sum += data[i];
10   }
11
12   cout << "sum = " << sum << endl;
13   return 0;
14 }
```

Debugging with gdb (2)

- `$> g++ -g -o sum sum.cpp`
- `$> gdb sum`
- `(gdb) run`
  - program received `EXC_BAD_ACCESS` in line `xy`
  - now you can see what went wrong
- `(gdb) list 9 // to see listing`
- `(gdb) print sum // to see content of var sum`
- `(gdb) print i // to see content of var i`

Debugging with gdb (3)

- `(gdb) print data`
  - `$3 = 1, 2, 3, 4, 5`
- `(gdb) print data[0]`
  - `$4 = 1`
- `(gdb) quit`
  - `program is running, exit anyway (y/n) y`

Debugging with gdb (4)

- `(gdb) set var i=5`
- `(gdb) print i`
- `(gdb) set var i = i * 2`
- `(gdb) print i`
- `(gdb) set var i = $10+20`
- `(gdb) print i`
- `(gdb) print main::i`
- `(gdb) set var i = 35`
- `(gdb) p /x i // short command`
- `(gdb) list foo`

Debugging with gdb Using Breakpoints

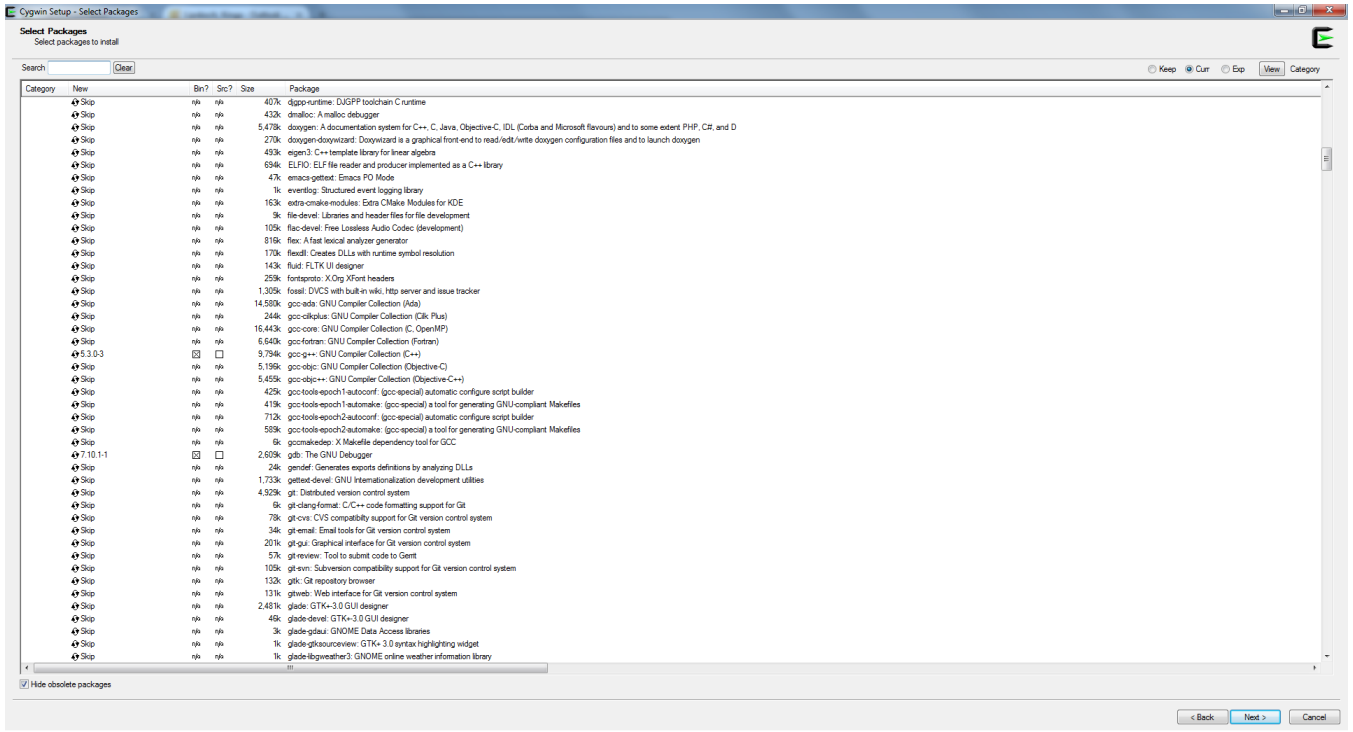
- (gdb) break 12 // add breakpoint at line 12
- (gdb) break main
- (gdb) break mod2.cpp:foo
- (gdb) continue
- (gdb) c // short command for continue
- (gdb) info break // show information about breakpoints
- (gdb) clear 12 // delete breakpoint
- (gdb) clear main
- (gdb) next // excutes the functions as an entire unit
- (gdb) step // enters + executes functions line by line

Debugging with gdb Use `help`

- `(gdb) help all`
- `(gdb) help <command-class>`
- `(gdb) help <command>`

Install Cygwin on Windows (1)

- Go to `https://cygwin.com/install.html`, download `setup-x86_64.exe` and install it
- During installation add `gcc-g++`, `gdb` and `make` listed under `Devel`



Install Cygwin on Windows (2)

- Once installed under `C:/cygwin64` you will have a Unix-like environment
- You can use it to compile and debug your code using `g++` and `gdb`

```
klipskoch@WGJSYMW1 ~
$ g++ -Wall -o hello hello.cpp

klipskoch@WGJSYMW1 ~
$ ./hello
Hello world in C++

klipskoch@WGJSYMW1 ~
$ gdb
GNU gdb (GDB) (Cygwin 7.10.1-1) 7.10.1
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-cygwin".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

Conditional Compilation: Debugging with the Preprocessor

```
1  #include <iostream>
2  #define DEBUG
3  using namespace std;
4  int triple_add(int i, int j, int k) {
5    return i + j + k;
6  }
7  int main() {
8    int i, j, k;
9    cin >> i >> j >> k;
10 #ifdef DEBUG
11   cerr << "i = " << i << endl << "j = " << j <<
       endl << "k = " << endl;
12 #endif
13   cout << triple_add(i, j, k) << endl;
14   return 0;
15 }
```

Conditional Compilation: Using the Command Line

- You can control from command line whether your program should output debugging messages or not
- `g++ -DDEBUG debug_pre.cpp`
- Compiles the file and also defines the preprocessor variable `DEBUG` for you

Trace Macros (1)

- Sometimes it's useful to print the code of each statement as it is executed, either to cout or to trace a file
- `#define TRACE(ARG) cout << #ARG << endl; ARG`
- Now you can go through and surround the statements you want to trace with this macro
- Apply `TRACE()` on the following:

```
for(int i=0; i<100; i++)
  cout << i << endl;
```

  expands into something which is not exactly what was originally meant
- tracemacro1.cpp
- Thus, use this technique carefully

Trace Macros (2)

- The following variation of the `TRACE()` macro can be used:
  ```
  #define D(a) cout << #a "=[" << a << "]" << endl;
  ```
- `D(expr)` displays the expression followed by its value (assuming there is an overloaded << operator for the result type)
- `tracemacro2.cpp`
- These trace macros represent the two most fundamental things you do with a debugger: trace and display values
- Good debuggers are excellent for productivity, but sometimes they are not available or not convenient to use
- These techniques always work regardless of the situation

Trace File

- The following code creates a trace file and sends all the output that would normally go to `cout` into that file
- Use `#define TRACEON` and include the header file
- `trace.h`
- `tracetest.cpp`
- All `cout` statements in the program are now sent to the trace file

Memory Leaks

- A memory leak is a type of resource leak that occurs when a computer program incorrectly manages memory allocations in such a way that memory which is no longer needed is not released
- In OOP, a memory leak may happen when an object is stored in memory but cannot be accessed by the running code
- Example:

```
1 void memLeak() {
2   int *data = new int;
3   *data = 15;
4 }
```

Finding Memory Leaks

- Common problems with memory allocation include mistakenly calling `delete`, calling `delete` more than once, forgetting to `delete` a pointer
- To use the memory checking system, you need to include the header file `MemCheck.h`, link the object file `MemCheck.o` into your application to intercept all the calls to `new` and `delete`, and call the macro `MEM_ON()` to initiate memory tracing
- `MemCheck.h`
- `MemCheck.cpp`
- `MemTest.cpp`

Tools for Finding Memory Leaks

- cppcheck – open source, static analyzer
  http://cppcheck.sourceforge.net/
- Valgrind – open source, Linux, Mac OS, Android, runtime analyzer
  http://valgrind.org/
- Visual Leak Detector – open source, Windows
  https://vld.codeplex.com/
- MTuner – commercial
  http://mtuner.net/
- Splint – open source, any plattform
  http://splint.org/
- ...

Unit Testing: Definition

- Unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use [1]
- Intuitively, one can view a unit as the smallest testable part of an application
- In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure
- In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method

[1] Kolawa, Adam; Huizinga, Dorota (2007). Automated Defect Prevention: Best Practices in Software Management. Wiley-IEEE Computer Society Press.

Simple Unit Testing

- Writing software is all about meeting requirements
- It is difficult to articulate software requirements without sampling an evolving, working system
- Solution: specify a little, design a little, code a little, and test a little; Then after evaluation the outcome, do it all over again; This is called iterative development
- Testing + programming is faster than just programming
- If your code passes all your tests, you know that if the system is not working, your code is probably not the problem

Automated Testing

- Often developers use come well-behaved input to produce some expect output, which they inspect visually
- Two problems:
  - Programs do not always receive only well-behaved input
  - Inspecting the output visually is tedious and error prone
- Better approach: first think like a tester and write tests, and then think like a developer and write your code
- Concretely, formulate tests as collections of boolean expressions and have a test program report any failures

Simple Example with a `Date` class

- A date can be initialized with a string (`YYYYMMDD`) or nothing (meaning today's date)
- A `Date` object can yield its year, month, and day or a string of the form "`YYYYMMDD`"
- All relational comparisons should be available, as well as computing the duration between two days (in years, months, and days)
- `Date1.h`
- `Date1.cpp`
- `SimpleDateTest.cpp`

More Tests for Date

- One can proceed with the implementation of the Date class iteratively until all the requirements are met
- By writing the tests first, one is more likely to think of corner cases that may break your implementation
- `Date2.h`
- `Date2.cpp`
- `SimpleDateTest2.cpp`

Tests with Exceptions for Date

- You can throw specific exceptions in case of invalid input, usage or invalid operations
- The global variables can be incremented depending if an exception is thrown or not
- Date3.h
- Date3.cpp
- SimpleDateTest3.cpp

Classify Tests Using a Makefile

- Date3.h
- Date3.cpp
- TestOperators.cpp
- TestFunctions.cpp
- TestDuration.cpp
- TestExceptions.cpp
- Doxyfile
- Makefile.txt
- `make -f Makefile.txt test-all`
- `make -f Makefile.txt run-testoperators`
- ...
- `make -f Makefile.txt run-test-all`
- `make -f Makefile.txt clear`

Unit Test Tools

- API Sanity Checker
  http://lvc.github.io/api-sanity-checker/
- Cantata++ - commercial
  http://www.qa-systems.com/cantata.html
- CppUnit
  https://freedesktop.org/wiki/Software/cppunit/
- Google Test
  https://github.com/google/googletest/
- Parasoft C/C++test - commercial
  https://www.parasoft.com/product/cpptest/
- ...

Makefile (1)

- A makefile has the name "Makefile"
- Makefile contains following sections:
  - Comments
  - Macros
  - Explicit rules
  - Default rules

Makefile (2)

- Comments
  - Any line that starts with a # is a comment
- Macro format
  - `name = data`
  - Ex: `OBJ=linked_list.o use_linked_list.o`
  - Can be referred to as `$(OBJ)` from now on

Makefile (3)

Explicit rules

- `target:source1 [source2] [source3]`
  `command1`
  `[command2]`
  `[command3]`
- `target` is the name of file to create
- File is created from `source1` (and `source2`, ...)
- `use_linked_list: use_linked_list.o linked_list.o`
  `g++ -o use_linked_list`
  `use_linked_list.o linked_list.o`

Makefile (4)

Explicit rules

- `target:`

  `command`

  Commands are unconditionally executed each time make is run
- Commands may be omitted, built-in rules are used then to determine what to do

  `use_linked_list.o: linked_list.h use_linked_list.cpp`
- Create `use_linked_list.o` from `linked_list.h` and `use_linked_list.cpp` using standard suffix rule for getting to `use_linked_list.o` from `linked_list.cpp`
- `$(CC) $(CFLAGS) -c file.cpp`

Example Makefile

```
CC = g++
CFLAGS = -Wall

OBJ = linked_list.o use_linked_list.o

all: use_linked_list

use_linked_list: $(OBJ)
                 $(CC) $(CFLAGS) -o use_linked_list $(OBJ)

use_linked_list.o: linked_list.h use_linked_list.cpp

linked_list.o: linked_list.h linked_list.cpp

clean:
       rm -f use_linked_list *.o
```

Final Exam: Details

- Saturday, the 28th of April, 2018, 10:00 - 12:00 in East Wing, IRC
- Programming exercises to be solved on paper
  - You have two hours to solve exercises
  - Similar to the programming assignments
- You may not use books or other documentation while taking the exam
- You may not use mobile phones, calculators or any other electronic devices
- TAs will give a tutorial before the exam