Implementation

Minsoo Ryu

Hanyang University

Contents

1. Programming Techniques
2. Coding Style

Generic Programming

- Generic programming is about generalizing software components so that they can be easily reused in a wide variety of situations
  - Generic programs are written in an extended grammar and are made adaptable by specifying variable parts that are then somehow instantiated later by the compiler with respect to the base grammar

- Two major techniques
  - Type-indexed programming in procedural languages like C
  - C++ templates

Type-Indexed Programming

- Implement procedures such that it can handle different types depending in a variety of situations

```
void generic_func (…, …, int type)
{
    switch(type){
    case A:
            func_A( );
             // ...
            break;
    case B:
            func_B();
            // …
            break;
    ….
}
```

C++ Templates

- C++ Templates allow to create generic functions that admit any data type as parameters and return value without having to overload the function with all the possible data types

- Templates provide direct support for generic programming
  - The C++ template mechanism allows a type to be a parameter in the definition of a class or a function

C++ Templates

❒ Prototype of a class template

- **template <class T> class X { };**

```
template <class T> class two_store      // template class declaration
{
      T value[2];
public:
      two_store (T first, T second) { value[0] = first; value[1] = second; }
      T getmax ( ) {return value[0] > value[1] ? value[0] : value[1]; }
};
```

```
int main ( )
{
      two_store <int> myobject (100, 75);          // instantiation
      cout << myobject.getmax();
      return 0;
}
```

Automata-Based Programming

- Automata-based programming is a programming paradigm in which the program or its part is thought of as a model of a finite state machine or any other (often more complicated) formal automata
  - FSM-based programming is generally the same, but, formally speaking, doesn't cover all possible variants as FSM stands for finite state machine and automata-based programming doesn't necessarily employ FSMs in the strict sense

Example

- Consider we need a program in C that reads a text from standard input stream, line by line, and prints the first word of each line

```
int main()
{
    int c;
    do {
        c = getchar();
        while(c == ' ')
            c = getchar();
        while(c != EOF && c != ' ' && c != '\n') {
            putchar(c);
            c = getchar();
        }
        putchar('\n');
        while(c != EOF && c != '\n')
            c = getchar();
    } while(c != EOF);
    return 0;
}
```

Automata-Based Program

- Three stages: skipping the leading spaces, printing the word and skipping the trailing characters
  - Let's call them states before, inside and after

```
int main()
{
    enum states {
        before, inside, after
    } state;
    int c;
    state = before;
    while((c = getchar()) != EOF) {
        switch(state) {
            case before:
                if(c == '\n') {
                    putchar('\n');
                } else
                if(c != ' ') {
                    putchar(c);
                    state = inside;
                }
                break;
            case inside:
                switch(c) {
                    case ' ':  state = after; break;
                    case '\n':
                        putchar('\n');
                        state = before;
                        break;
                    default:   putchar(c);
                }
                break;
            case after:
                if(c == '\n') {
                    putchar('\n');
                    state = before;
                }
        }
    }
    return 0;
}
```

Using a State Transition Table

```
enum states { before = 0, inside = 1, after = 2 };
struct branch {
    enum states new_state:2;
    int should_putchar:1;
};
struct branch the_table[3][3] = {
                /* ' '          '\n'        others */
    /* before */ { {before,0}, {before,1}, {inside,1} },
    /* inside */ { {after, 0}, {before,1}, {inside,1} },
    /* after  */ { {after, 0}, {before,1}, {after, 0} }
};
void step(enum states *state, int c)
{
    int idx2 = (c == ' ') ? 0 : (c == '\n') ? 1 : 2;
    struct branch *b = & the_table[*state][idx2];
    *state = b->new_state;
    if(b->should_putchar) putchar(c);
}
int main()
{
    int c;
    enum states state = before;
    while((c = getchar()) != EOF)
        step(&state, c);
    return 0;
}
```

Concurrent Programming

- Concurrent computing is the concurrent (simultaneous) execution of multiple interacting computational tasks
  - These tasks may be implemented as separate programs, or as a set of processes or threads created by a single program
  - The tasks may also be executing on a single processor, several processors in close proximity, or distributed across a network
  - Concurrent computing is related to parallel computing, but focuses more on the interactions between tasks
  - Correct sequencing of the interactions or communications between different tasks, and the coordination of access to resources that are shared between tasks, are key concerns during the design of concurrent computing systems

Concurrent Interaction and Communication

- **Shared memory communication (read and write)**
  - Concurrent components communicate by altering the contents of shared memory locations
  - This style of concurrent programming usually requires the application of some form of locking (e.g., mutexes (means mutual exclusion), semaphores, or monitors) to coordinate between threads

- **Message passing communication (send and receive)**
  - Concurrent components communicate by exchanging messages
  - The exchange of messages may be carried out asynchronously or may use a rendezvous style in which the sender blocks until the message is received

Publish/Subscribe

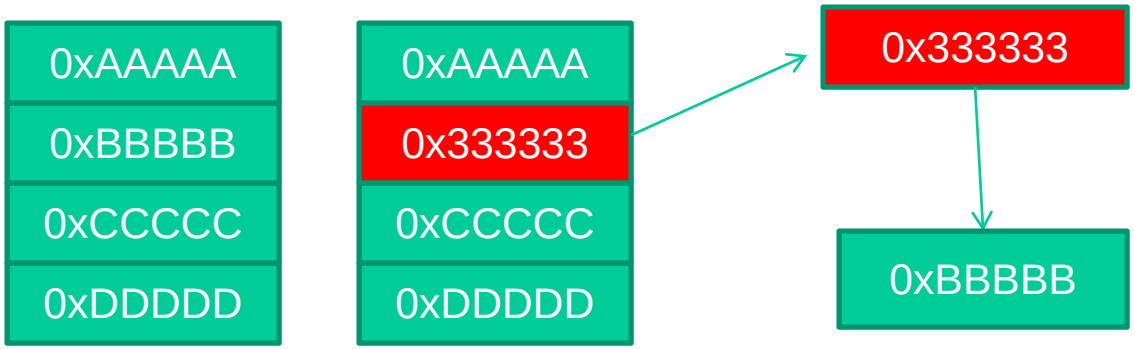
- Publish/subscribe is an asynchronous messaging paradigm where senders (publishers) of messages are not programmed to send their messages to specific receivers (subscribers)
  - Rather, published messages are characterized into classes, without knowledge of what (if any) subscribers there may be
  - Subscribers express interest in one or more classes, and only receive messages that are of interest, without knowledge of what (if any) publishers there are
  - This decoupling of publishers and subscribers can allow for greater scalability and a more dynamic network topology
  - Publish/subscribe is a sibling of the Message Queue paradigm

Hooking

- Hooking in programming is a technique employing so called hooks to make a chain of procedures as an event handler
- Thus, after the handled event occurs, control flow follows the chain in specific order
- The new hook registers its own address as handler for the event and is expected to call the original handler at some point, usually at the end
  - Each hook is required to pass execution to the previous handler, eventually arriving to the default one, otherwise the chain is broken

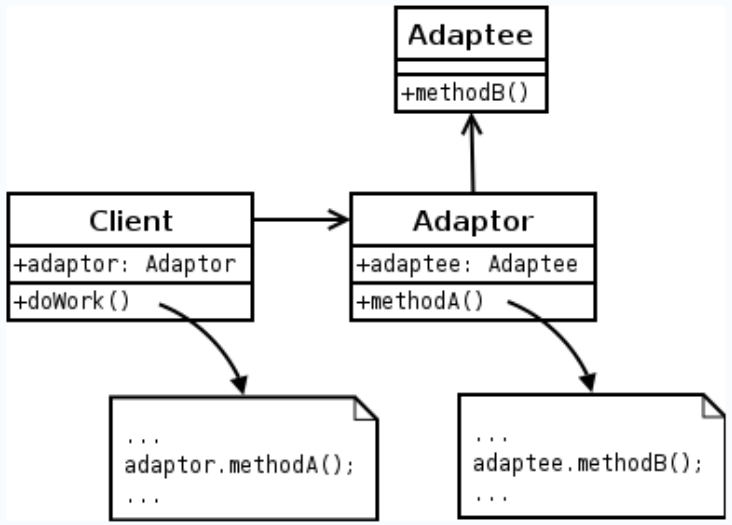
Hooking

- Hooking can be used for many purposes, including debugging and extending original functionality
- It can also be misused to inject (potentially malicious) code to the event handler



Wrappers (Adapters)

- A wrapper, also known as adapter, adapts one interface into one that a client expects
  - An adapter allows to work together that normally could not because of incompatible interfaces by wrapping its own interface around that of an already existing interface
  - The adapter is also responsible for handling any logic necessary to transform data into a form that is useful for the consumer

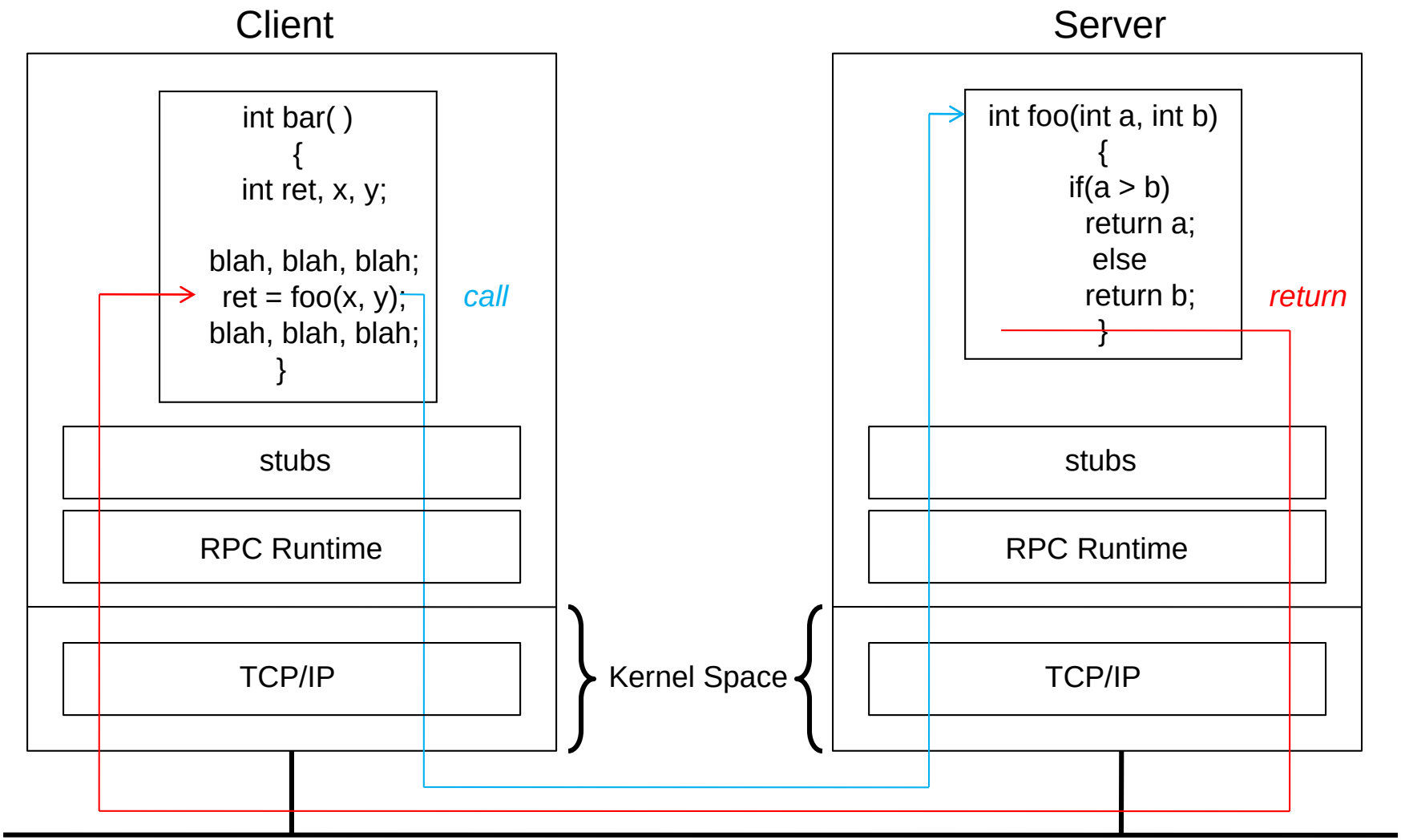


Stubs

- A method stub or simply stub in software development is a piece of code used to stand in for some other programming functionality
  - A stub may simulate the behavior of existing code (such as a procedure on a remote machine) or be a temporary substitute for yet-to-be-developed code
  - Stubs are therefore most useful in distributed computing in addition to general software development and testing

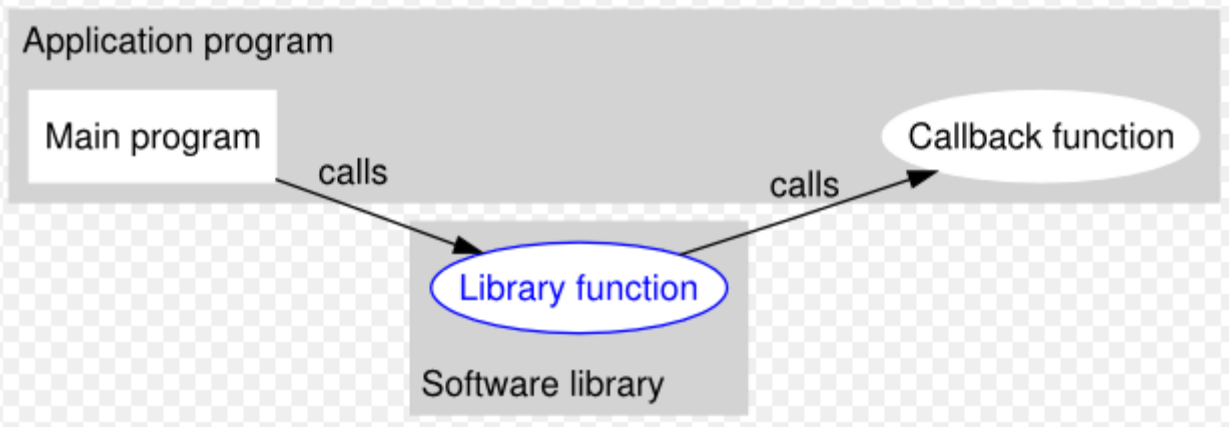
- RPC (Remote Procedure Call) in distributed computing
  - The client-side stub (proxy) is a procedure that looks to the client as if it were a callable server procedure
  - A server-side stub (skeleton) looks to the server as if it's a calling client

Stubs in Remote Procedure Call



Callbacks

- A callback is executable code that is passed as an argument to other code
  - It allows a lower-level software layer to call a subroutine (or function) defined in a higher-level layer



Contents

1. Programming Techniques
2. Coding Style

Coding Style

- Coding style (also called coding standards or code convention or programming style) is a term that describes conventions for writing source code in a certain programming language

- Coding style is often dependent on the actual choice of programming language one is writing in C style will vary from BASIC style, and so on

Good Coding Style

- Good style, being a subjective matter, is difficult to concretely categorize; however, there are a number of general characteristics
- With the advent of software that formats source code automatically, the focus on how source code looks should yield to a greater focus on naming, logic, and higher techniques
  - As a practical point, using a computer to format source code saves time, and it is possible to then enforce company-wide standards without religious debates

Appropriate variable names

- **Appropriate choices for variable names is seen as the keystone for good style**
- **Poorly-named variables make code harder to read and understand**
  - **Because of the choice of variable names, the function of the code is difficult to work out**
  - **However, if the variable names are made more descriptive**

```
get a b c
if a < 12 and b < 60 and c < 60
      return true
else
      return false
```

```
get hours minutes seconds
if hours < 12 and minutes < 60 and seconds < 60
      return true
else
      return false
```

Indent style

- **Indent style, in programming languages that use braces or indenting to delimit logical blocks of code, such as C, is also a key to good style**
  - **Using a logical and consistent indent style makes one's code more readable**

```
if(hours<12&&minutes<60&&seconds<60){return true;}
else{return false;}
```

```
if (hours < 12 && minutes < 60 && seconds < 60)
{
    return true;
}
else
{
    return false;
}
```

Looping and control structures

- **The use of logical control structures for looping adds to good programming style as well**
  - **It helps someone reading code to understand the program's sequence of execution (in imperative programming languages)**
  - **The use of the "for" construct makes the code much easier to read**

```
count = 0
while count < 5
      print count * 2
      count = count + 1
endwhile
```

```
for count = 0, count < 5, count=count+1
      print count * 2
```

Spacing

- **Free-format languages often completely ignore whitespace**
  - **Making good use of spacing in one's layout is therefore considered good programming style**

```
int count;for(count=0;count<10;count++){printf("%d",count*count+count);}
```

```
int count;
for (count = 0; count < 10; count++)
{
      printf("%d", count * count + count);
}
```

- **In the C-family languages, it is also recommended to avoid using tab characters in the middle of a line as different text editors render their width differently**

GNU Coding Style

- The GNU coding standards are a set of rules and guidelines for writing programs that work consistently within the GNU system
  - The standards document is part of the GNU Project and is available from the GNU website
  - Though it focuses on writing free software for GNU in C, much of it can be applied more generally
  - In particular, contributors to the GNU Project should always try to follow the standards — whether or not their programs are implemented in C —, and it can be a good idea for most authors of free Unix programs to follow the standards — whether or not their programs are officially part of the GNU Project
- Visit http://www.gnu.org/prep/standards/

```
int
main (int argc, char *argv[])
{
  struct gizmo foo;

  fetch_gizmo (&foo, argv[1]);

 check:
  if (foo.type == MOOMIN)
    puts ("It's a moomin.");
  else if (foo.bar < GIZMO_SNUFKIN_THRESHOLD
           || (strcmp (foo.class_name, "snufkin") == 0)
               && foo.bar < GIZMO_SNUFKIN_THRESHOLD / 2))
    puts ("It's a snufkin.");
  else
    {
      char *barney   /* Pointer to the first character after
                        the last slash in the file name.  */
      int wilma;     /* Approximate size of the universe.  */
      int fred;      /* Max value of the `bar' field.  */

      do
        {
          frobnicate (&foo, GIZMO_SNUFKIN_THRESHOLD,
                      &barney, &wilma, &fred);
          twiddle (&foo, barney, wilma + fred);
        }
      while (foo.bar >= GIZMO_SNUFKIN_THRESHOLD);

      store_size (wilma);

      goto check;
    }

  return 0;
}
```

Don’ts (http://thc.org/root/phun/unmaintain.html)

- **The following don’ts do not guarantee you a lifetime of employment**
  - **Naming**
    - New Uses For *Names For Baby* like “Fred”
    - Single Letter Variable Names like “a, b, c”
    - Creative Miss-spelling like “SetFileOpenning, SetFileClozing”
    - Be abstract like “PerformDataFunction, DoIt”
    - Thesaurus Surrogatisation like “display, show, present”
  - **Camouflage**
    - Code That Masquerades As Comments and Vice Versa
    - Hide Macro Definitions in amongst rubbish comments
      - ✓ #define a=b a=0-b
    - Look busy
      - ✓ #define fastcopy(x,y,z) /*xyz*/

        ...

        fastcopy(array1, array2, size); /* does nothing */
  - **…**