Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 7

- Function Overloading
- C/C++ Preprocessor
- Testing

Function Overloading

```
void print(int);
void print(double);
void print(char);

char  c;
int   i;
short s;
float f;

print(c); // exact match: calls print(char)
print(i); // exact match: calls print(int)
print(s); // integral promotion: calls print(int)
print(f); // float promotion: calls print(double)

print('a'); // exact match: calls print(char)
```


Function Overloading Details

- We would like to use a single function name for similar functionality applied to different types. E.g. `+ - * / print`
- Compiler distinguishes functions by their signature: function name + list of parameter types without & and `const`. Return type is not considered!
- To find the matching function the compiler
  - looks for an exact type match first,
  - then for matches after promotion within integer and floating point types, and then
  - for other conversions of built-in or user types

C/C++ Preprocessor

- Compilation: transforming a textual program description into an executable form
- Preprocessor: separate first step in compilation:
  - Remove comments
  - Macro substitution (`#define`)
  - Conditional compilation (`#if`)
  - File inclusion (`#include`)
- Preprocessor directive: first non-white-space character in line is #
- Only one per line

Macro Substitution

```
#define FOREVER for(;;)

FOREVER { foo(); }
is translated into:    for (;;) { foo(); }
```

- Syntax of a macro definition:
  `#define` <identifier> <replacement text>
- Subsequent occurrences of the identifier in identifier context get replaced by the replacement text. E.g.
  `xxFOREVERxx` and `"FOREVER"` are not replaced!
- Replacement text normally is the remainder of line
- Long definitions may be continued by placing `\` at the end of each line to be continued
- Scope is from point of definition to the end of current file

Macros With Parameters

```
#define extract_index(x) (((x) >> 8) & 0xff)

index = extract_index(packed_data);

becomes

index = (((packed_data) >> 8) & 0xff);
```

- Syntax: `#define` <ident>`(`<ident>`,...,`<ident>`)` <text>
- Macro parameters get replaced by actual arguments when macro is expanded
- Macro expansion is done recursively until no more matches are found

More Macro Examples

```
#define FOR(i,n) for (i=0; i<(n); ++i)

FOR (i, 10) { foo(i); }
becomes
for (i=0; i<(10); ++i) { foo(i); }

#define MAX(a,b)  ((a)>(b)?(a):(b))
not recommended! multiple evaluation!
also, use lots of () to ensure evaluation order!

MAX(a++,b++)
becomes
((a++)>(b++)?(a++):(b++)) OOPS! 2x a++,b++!
```

In C++ there is hardly any reason for using parameterized macros anymore! Use template/inline functions (later).

#if Statement

- Syntax & Semantics

```
#if <const-expr>   : true iff const-expr != 0
#ifdef <ident>     : true iff <ident> is defined
#ifndef <ident>    : true iff <ident> is undef.
#else              : alternative path
#elif <const-expr>: else-if condition
#endif             : end of #if statement
```

- <const-expr> can consist of macro names, integer constants, operators, parenthesis and `defined(<macro-name>)`
- `#error "text"`: generates error msg. `"text"`

Conditional Compilation

```
#ifdef UNIX
  ... Unix code
#elifdef WINDOWS
  ... Windows code
#else
  #error "Unsupported OS"
#endif
```

```
#define TEST 1

#if TEST
  ... test code
#endif
```

- Compiling parts of programs depending on constant expressions. If false, program text is skipped
- Useful for dealing with different environments and debugging
- Can pass macro definitions to gcc/g++ via -D option. E.g.

```
g++ -DUNIX -DNDEBUG foo.c // UNIX,NDEBUG defined
g++ -DFOO=3 foo.c         // FOO has value 3
```


File Inclusion

- Two forms:
  `#include "filename"`
  `#include <filename>`
- Line is replaced by the content of the file filename, which itself may contain `#include` lines
- `"filename"` : search for file begins in directory where the source program is located. If not found, search in system header directories
- `<filename>` : search file in system header directories
- Main purpose: including interface information such as function and class declarations

#include Examples (1)

```
#include <iostream>

std::cin, std::cout, std::cerr,
overloaded operators << >> etc. now declared


#include "mydecl.h"

Your functions and classes declared in local
file mytypes.h now visible
```


#include Examples (2)

How to avoid including the same file twice which would cause compiler error messages or warnings?

mydecl.h:

```
#ifndef MYDECL_H   // distinct macro for
#define MYDECL_H   // each header file

#define FOR(i,n) for (i=0; i<(n); ++i)

int square(int x);
int swap(int &x, int &y);
int bitcount(unsigned int x);

#endif
```

Testing and Debugging

```
// returns approximation of square-root of x
// precondition: x >= 0

double sqrt(double x) {
  if (x < 0) { cerr << "sqrt:x<0" << endl; exit(5); }
  ... compute square root r
  ... check whether r*r is close to x
  return r;
}
```

- Testing each function is CRUCIAL
- Pre- and post-conditions should be checked during program execution in function body
- Also check border cases with separate code

assert Macro

- Syntax: `#include <cassert>`
  `assert(<expression>);`
- Execution stops iff the expression evaluates to 0. An error message informs about the program file and line number where the assertion failed
- Check can be turned off by defining `NDEBUG` before `#include <cassert>`
  (usually done with compiler option `-DNDEBUG`)
- Turn assert on when debugging program
- Turn off to speed up execution when convinced that code is correct

assert Example

```
//#define NDEBUG // uncomment to turn assert checks
                 // off or pass -DNDEBUG to g++
#include <cassert>

// computes the square root of x
// precondition: x >= 0

double sqrt(double x) {
  assert(x >= 0); // pre-condition

  ... compute square root r

  assert(...r approximates sqrt(x)...); // post-cond.
  return r;
}
```


Checking Preprocessor Output

`g++ -E ...`

stops compilation after the preprocessing phase and prints result to stdout

Easy way to check what the preprocessor does

(online demonstration)