COMP 2355 Introduction to Systems Programming

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/

Today: C++ Introduction

- Reference variables
- Function overloading
- Symbol mangling
- Default arguments
- Exceptions
- Streams

Reference Variables

C++:

```
void f() {
  int i = 5;
  int & iref = i;
  iref = 6;
  printf("%d\n", i);
}
```

C:

```
void f() {
  int i = 5;
  int * iref = &i;
  *iref = 6;
  printf("%d\n", i);
}
```

Call-By-Reference

C++:

```
void f(int & iref) {
  iref++;
}
void g() {
  int i = 5;
  f(i);
  printf("%d\n", i);
}
```

C:

```
void f(int * iref) {
  (*iref)++;
}
void g() {
  int i = 5;
  f(&i);
  printf("%d\n", i);
}
```

Overloading

C++:

```
int square(int a) {
  return a*a;
}
float square(float a) {
  return a*a;
}
double square(double a) {
  return a*a;
}
```

C:

```
int square_i(int a) {
  return a*a;
}
float square_f(float a) {
  return a*a;
}
double square_d(double a) {
  return a*a;
}
```

Overloading

- Can overload with different number of arguments
- Can overload `int` vs. `unsigned int` vs. `int*`
- Can not have functions differ only by return type
- Can not overload with `...` and share common prefix
- Can not overload `T` and `&T`

Examples

Allowed in C++:

```
int square(int a){
  return a*a;
}
float square(int * a){
  return (*a)*(*a);
}
int square(int a, int b){
  return a*b;
}
```

Not allowed in C++:

```
int square(int a) {
  return a*a;
}
float square(int a) {
  return a*a;
}
int square(int & a) {
  return a*a;
}
```

Examples

Allowed in C++:

```
int sq(int a, ...) {
  return a*a;
}
float sq(int * a, ...) {
  return (*a)*(*a);
}
```

Not allowed in C++:

```
int square(int a, float f) {
  return a*a;
}
int square(int a, ...) {
  return a*a;
}
```

Overloading

- Overloading is implemented using name mangling
- Similar to the corresponding C code we have seen
- Do not try to predict how C++ compilers mangle; different compiler (versions) differ!
- You can disable mangling like this:

```
extern "C" double sin(double d);
```

Why disable mangling?

- If you want to call C++ code from C, you can not expect the C compiler to use the mangled C++ names
- If you want to call C code, you need to tell the C++ compiler not to use mangled names

$\Rightarrow$ In either case, the header files need to specify in the declarations "do not mangle"

- Issue: "extern "C"" is not valid C syntax, but we need it in C headers for the C++ compiler!

How to Disable Mangling for C Headers

Solution: use the preprocessor:

```
#ifdef __cplusplus
extern "C" {
#endif
int my_normal_c_headers(float f);
#ifdef __cplusplus
}
#endif
```

Default arguments

```
void f(int a = 1, int b = 2, float c = 3.14);
void g(int a    , int b = 2, float c = 3.14);

void e(int a = 1, int b    , float c = 3.14); // illegal!

f();
f(1,5);
g(3);
```

Limitations

- You can provide default arguments for multiple arguments
- Default arguments must be provided right-to-left
- Default arguments can cause additional ambiguities when used with overloading
- Default arguments must be specified in the function declaration

Examples

Allowed in C++:

```
int square(int a=0);
int die(char*emsg="Bug");
int hconnect
  (char*host="localhost",
   unsigned short port=80);
int f() {
  return hconnect
    ("www.example.com");
}
```

Not allowed in C++:

```
int f(const char * a,
      int b = 5);
int f(const char * a);
int m() {
  return f(NULL);
}
```

Exceptions

- C++ Exceptions deceptively look like Java Exceptions
- `throw` throws an exception, but it does not have to be an object!
- `try` and `catch` are used to catch exceptions
- As in Java, throwing an exception unwinds the stack
- C++ has no `finally`
- Catching works with subtyping, just like in Java

Example

```
int f() { throw 52; }
int g() {
  f();
  printf("Unreachable!\n");
}
int main() {
  try { g(); }
  catch (int i) { printf("%d\n", i); }
}
```

Example: Nesting

```
int f() { throw "Oops!"; }
int g() {
  try { f(); }
  catch (int i) { printf("Wrong type!\n"); }
}
int main() {
  try { g(); }
  catch (const char * m) { printf("%s\n", m); }
}
```

Example: Catch All

```
int f() { throw 4.2; }
int main() {
  try {
    f();
  } catch (...) {
    printf("Error!\n");
  }
}
```

Example: Rethrow

```
int f() { throw "Oops!"; }
int g() {
  try { f(); }
  catch (const char * m) {
    printf("Yuck!\n");
    throw;
  }
}
int main() {
  try { g(); }
  catch (const char * m) { printf("%s\n", m); }
}
```

Programming with C++ Exceptions

- While you can throw an `int`, throw objects, usually subclasses of the `exception` class
- C++ is not garbage collected, so you need to think about how resources will be freed when the stack is unwound!
- The stack unwinding code will call destructors

$\Rightarrow$ More on cleaning up in the next lecture!

C++ IO Streams

- Typed
- Needs header `<iostream>`
- Extensible (see: lecture 16)
- `cin` ≡ `stdio`
- `cout` ≡ `stdout`
- `cerr` ≡ `stderr`

Example: Output

```
#include <iostream>
using namespace std;
int main(int argc, char ** argv) {
  int a = 5;
  cout << "Hello " << a << " World" << endl;
  return 0;
}
```

Example: Output Without Flushing

```
#include <iostream>
using namespace std;
int main(int argc, char ** argv) {
  int a = 5;
  cout << "Hello " << a << " World\n";
  return 0;
}
```

Example: Formatting Output

```
#include <iostream>
using namespace std;
int main(int argc, char ** argv) {
  int a = 5;
  cout << setw(12) << hex << 1024 << endl;
  // prints ‘‘........400’’ (dot == space)
  return 0;
}
```

Output Manipulators[1]

dec	switch to decimal
hex	switch to hexadecimal
oct	switch to octal
left	left-align
right	left-align
setbase(b)	set number base to b
setfill(c)	set fill character to c
setw(w)	set field width to w
setprecision(p)	set precision to p

[1] Require including `iomanip`

Output Manipulators

- Effects continue to apply for future output
- Exception: `setw` only applies to next field

Example: Input

```
#include <iostream>
using namespace std;
int main(int argc, char ** argv) {
  int k;
  float f;
  string st;
  cin >> st >> n >> k;
  // expects: somestring 1.2345 6
  return 0;
}
```

Input Processing

- Type conversion is automatic, but input must match
- Conversion splits elements at whitespace
- Input stops when conversion can no longer continue, just like with `sscanf`
- You can define your own input parsers for data types

$\Rightarrow$ Lecture 16

Opening Files: Output

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char ** argv) {
  ofstream myfile;
  myfile.open("/etc/passwd");
  myfile << "root::0:0:cracked:/:/bin/sh";
  myfile.close();
  return 0;
}
```

C++ IO Streams

- `ostream`, `ofstream`: output only
- `istream`, `ifstream`: input only
- `iostream`, `fstream`: input and output

C++ Binary IO

```
#include <iostream>
#include <fstream>
using namespace std;
void cppwrite(const void * data, unsigned int size) {
  ofstream fi ("data.dat", ios::out|ios::binary);
  if (fi.is_open()) {
     fi.write(data, size);
     fi.close();
  } else {
     cerr << "Unable to open file." << endl;
  }
}
```

IO Errors

- `int ios::fail(void)` – execution error
- `int ios::eof(void)` – end of file
- `int ios::bad(void)` – data loss
- `int ios::good(void)` – no error
- `int ios::hardfail(void)` – hardware error

IO Errors

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char ** argv) {
  int i;

  while (cin >> i, !infile.eof()) cout << i;
  return 0;
}
```

IO Errors and "!"

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char ** argv) {
  int i;

  while (cin >> i) cout << i;
  return 0;
}
```

Questions

?