MODERN C++ DESIGN PATTERNS

Function Pointers & Applications

by Prasanna Ghali

Trailing Return Type Syntax (1/2)

- We're familiar with following syntax for declaring functions

```
// since birth of C ...
int sum(int const *p, size_t N);
```

- Since C++11, we have alternative syntax for writing function declarations:

```
// since C++11 ...
auto sum(int const *p, size_t N) -> int;
```

Trailing Return Type Syntax (2/2)

- Use of `auto` in alternate function syntax has no meaning other than to be part of syntax

```
auto main() -> int {
  // statements as usual ...
  return 0;
}
```

- Use of `auto` without trailing return type syntax means something else!!!

```
auto sum(int x, int y) {
  // return type is automatically deduced by
  // compiler based on expressions in function body
}
```

Trailing Return Type Syntax (3/3)

- ~~Ordinary~~ Complicated declaration

```
char *(*foo(char*, int))[5];
```

is simplified using trailing return type syntax:

```
auto foo(char*, int) -> char* (*) [5];
```

Function Pointers

- Deciphering function pointer declarations
- Applications

Function Pointers: Intro (1/2)

- Functions are basically blocks of machine code in memory
- Compiler treats function pointers differently than pointers to data
  - Function's name evaluates to address of first byte of associated machine code block
  - No need to use address-of operator & to get pointer to function
  - No need to use dereference operator * on function pointer to call corresponding function

Function Pointers: Intro (2/2)


```
int f() {
  return 255;
}

int main() {
  std::cout << reinterpret_cast<void*>(f)  << ", "
            << reinterpret_cast<void*>(&f) << ", "
            << reinterpret_cast<void*>(*f) << ", "
            << std::hex << "0x" << f()     << "\n";
}
```

output

```
0x55ec958f6149, 0x55ec958f6149, 0x55ec958f6149, 0xff
```

Function Pointers: Declaration and Assignment (1/2)

- Functions can only be assigned to pointer variables declared with appropriate type

Function Pointers: Declaration and Assignment (2/2)


```
int main() {
  // auto (*pf)() -> int;
  int (*pf)();
  pf = f;         // ok
  pf = &f;        // ok
  pf = *f;        // ok
  pf = f();       // error!!!
  int i{pf()}; // ok
  f = pf;         // error: left-operand must be lvalue
  printf("%p, %p, %p, 0x%x\n", f, *f, &f, f());
  printf("%p, %p, %p, 0x%x\n", pf, *pf, &pf, pf());
}
```

```
int f() {
  return 255;
}
```

output

```
0x5642baf64169, 0x5642baf64169, 0x5642baf64169, 0xff
0x5642baf64169, 0x5642baf64169, 0x7ffd46ae0200, 0xff
```

Function Pointers: Calling the Function


```
int f() {
  return 255;
}

int main() {
  // initialize pointer variable with address of function f
  // auto (*pf)() -> int {f};
  int (*pf)() {f};

  // all statements are equivalent ...
  int value;
  value = f();     // call function "normally"
  value = pf();    // call function thro' pointer to function
  value = (*pf)(); // call function by dereferencing pointer
}
```

Function Pointers: Type Compatibity is Important (1/2)

- Since function pointer always points to function with specific signature, all functions you want to use with same function pointer must have same parameters and return type

Function Pointers: Type Compatibity is Important (2/2)


```
// function takes no parameters and returns an int value
int f() { return 255; } //auto f() -> int { return 255; }
// function takes no parameters and returns an int value
int g() { return 0; }   //auto g() -> int { return 0; }
// function takes no parameters and returns a double value
double h() { return 1.5; } //auto h() -> double { return 0.5; }

int main() {
  int    (*pf)(); // auto (*pf)() -> int;
  double (*ph)(); // auto (*ph)() -> double;
  pf = f; // ok: pf and f have same type
  pf = g; // ok: pf and g have same type
  pf = h; // error: incompatible types
  ph = h; // ok: ph and h have same type
  //pf = (auto (*) () -> int) h;
  pf = (int (*)()) h; // ok but crazy stuff ...
  printf("value: %d\n", pf()); // crazy value printed - not 1
}
```

Function Pointers: Passing Function Pointers (1/3)

- Easy enough to do – simply use a function pointer declaration as parameter to function

```
int Add(int x, int y) { return x + y; }
int Sub(int x, int y) { return x - y; }

int compute(int (*callback)(int,int), int x, int y) {
  return callback(x, y);
}

int main() {
  std::cout << "3 + 5 = " << compute(Add, 3, 5) << "\n";
  std::cout << "5 - 3 = " << compute(Sub, 5, 3) << "\n";
}
```

Function Pointers: Passing Function Pointers (2/3)

- Another [possibly easier] way to declare function `compute`

```
int Add(int x, int y) { return x + y; }
int Sub(int x, int y) { return x - y; }

int compute(int callback(int,int), int x, int y) {
  return callback(x, y);
}

int main() {
  std::cout << "3 + 5 = " << compute(Add, 3, 5) << "\n";
  std::cout << "5 - 3 = " << compute(Sub, 5, 3) << "\n";
}
```

Function Pointers: Passing Function Pointers (3/3)

- Third way to declare function `compute`

```
int Add(int x, int y) { return x + y; }
int Sub(int x, int y) { return x - y; }

int compute(auto callback(int,int) -> int, int x, int y) {
  return callback(x, y);
}

int main() {
  std::cout << "3 + 5 = " << compute(Add, 3, 5) << "\n";
  std::cout << "5 - 3 = " << compute(Sub, 5, 3) << "\n";
}
```

Function Pointers: Returning Function Pointer (1/2)

- Easy enough to do – simply use a function pointer declaration as function return type

```
// function that evaluates an operator and
// returns a pointer to appropriate function
int (*Select(char opcode))(int,int) {
  switch (opcode) {
    case '+': return Add;
    case '-': return Sub;
    case '*': return Mul;
    case '/': return Div;
    default: return nullptr;
  }
}
```

Function Pointers: Returning Function Pointer (2/2)


```cpp
// basic arithmetic functions ...
int Add(int x, int y) { return x+y; }
int Sub(int x, int y) { return x-y; }
int Mul(int x, int y) { return x*y; }
int Div(int x, int y) { return x/y; }

// using alternative trailing return type syntax
auto Select(char ch) -> int (*)(int, int) {
  switch (opcode) {
    case '+': return Add;
    case '-': return Sub;
    case '*': return Mul;
    case '/': return Div;
  }
  return nullptr;
}
```

Function Pointers and Arrays

- Can store function pointer types in arrays and iterate thro' such arrays …
- Useful for executing functions in an order that may not be known at compile time and without using conditional statements

Function Pointers and Arrays: Example 1


```
#include <cmath>

const int SIZE {3};
// a_ptr_fns is array of SIZE elements where each element is
// "pointer to function that takes double and returns double"
double (*a_ptr_fns[SIZE])(double) {std::sin, std::cos, std::tan};

// call functions indirectly by iterating thro' static array
void test_trig_fns() {
  for (int i{}; i < SIZE; ++i) {
    double x {a_ptr_fns[i](2.0)};         // ok: simplest ...
    //double x {(*a_ptr_fns[i])(2.0)};    // ok: harder ...
    //double x {(*(a_ptr_fns+i))(2.0)};   // ok: more harder ...
    //double x {(**(a_ptr_fns+i))(2.0)};  // ok: harder still ...
    //double x {*a_ptr_fns [i](2.0)};     // error ...
    std::cout << x << ' ';
  }
  std::cout << '\n';
}
```

Function Pointers and Arrays: Example 2


```cpp
// assign addresses of trig functions to static array
const int SIZE{3};
double (*a_ptr_fns[SIZE])(double) {std::sin, std::cos, std::tan};

void test_trig_fns2() {
// appf is array of SIZE elements where each element is "pointer to
// to pointer to function that takes double and returns double"
  double (**appf[SIZE])(double) {a_ptr_fns,a_ptr_fns+1,a_ptr_fns+2};
  for (int i{}; i < SIZE; ++i) {
    double x {(*appf[i])(2.0)};      // easiest ...
    //double x {(**(appf+i))(2.0)}; // harder ...
    std::cout << x << ' ';
  }
  std::cout << '\n';
}

int main() { test_trig_fns2(); }
```

Function Pointers and Arrays: Example 3


```
const int SIZE{3};
double (*a_ptr_fns[SIZE])(double) {std::sin, std::cos, std::tan};

void test_trig_fns3() {
// papf has type: "pointer to array of SIZE elements where
// each element is pointer to function that takes double
// and returns double"
  double (*(*papf)[SIZE])(double) {&a_ptr_fns};
  for (int i{}; i < SIZE; ++i) {
    double x {(*papf)[i](2.0)};         // simplest ...
    //double x {(*(*papf)[i])(2.0)}; // harder ...
    //double x {*papf[i](2.0)};         // error ...
    //double x {(*papf[i])(2.0)};     // compiles but crashes ...
    std::cout << x << ' ';
  }
  std::cout << '\n';
}
int main() { test_trig_fns3(); }
```

What Are Callbacks? (1/2)

- Suppose a function `compute` calls another function to perform a task …
- When the actual function called changes at runtime [and is conveniently passed as parameter to function `compute`], the called function is called *callback*
- Callbacks are common tools in every aspect of software development …

What Are Callbacks? (2/2)


```
int Add(int a, int b) { return a + b; }
int Sub(int a, int b) { return a - b; }

int compute1(int (*callback)(int,int), int a, int b) {
  return callback(a, b);
}
int compute2(int callback(int,int), int a, int b) {
  return callback(a, b);
}
int compute3(auto callback(int,int) -> int, int x, int y) {
  return callback(x, y);
}

int main() {
  std::cout << "3 + 5 = " << compute1(Add, 3, 5) << "\n";
  std::cout << "5 - 3 = " << compute2(Sub, 5, 3) << "\n";
  std::cout << "5 + 3 = " << compute3(Add, 5, 3) << "\n";
}
```

Callback Example: qsort

- qsort declared in C standard library …

```
void qsort(void* base, size_t num, size_t size,
           int (*compar)(const void*,const void*));
```

- Could have been also declared as:

```
void qsort(void* base, size_t num, size_t size,
           int compar(const void*,const void*));
```

- See *qsort.cpp*

Simplifying Complex Declarations

- Use `typedef` storage specifier in both C and C++ code to write type aliases
- Use trailing return type syntax since C++11 to simplify function declarations
- Use keyword `using` since C++11 to write type aliases

`using`: Alias Declaration Syntax

keyword *any standard or derived type*

```
using     IDENTIFIER = existing-type;
```

mnemonic traditionally in uppercase

using: Examples


```
// works in both C and C++ code
typedef long int BigInt;
typedef int (*PtrToFunc)(double);

// works since C++11
using BIGINT     = long int;
using PTRTOFUNC = int (*) (double);
using PTR_FUNC2 = auto (*)(double) -> int;
```

using: Example 1 with Function Pointers and Arrays


```
const int SIZE {3};
//double (*a_ptr_fns[SIZE])(double) { std::sin, std::cos, std::tan };
using PTR_FUNC = auto (*)(double) -> double;
PTR_FUNC a_ptr_fns[SIZE] { std::sin, std::cos, std::tan };

// call functions indirectly by iterating thro' static array
void test_trig_fns() {
  for (int i{}; i < SIZE; ++i) {
    std::cout << a_ptr_fns[i](2.0) << ' ';
    //std::cout << (*a_ptr_fns[i])(2.0) << ' '; // ok
  }
  std::cout << '\n';
}

int main() { test_trig_fns(); }
```

using: Example 2 with Function Pointers and Arrays


```
const int SIZE{3};
//double (*a_ptr_fns[SIZE])(double) {std::sin, std::cos, std::tan};
using PTR_FUNC = auto (*)(double) -> double;
PTR_FUNC a_ptr_fns[SIZE] { std::sin, std::cos, std::tan };

void test_trig_fns2() {
  //double (**appf[SIZE])(double){a_ptr_fns,a_ptr_fns+1,a_ptr_fns+2};
  PTR_FUNC* appf[SIZE] { a_ptr_fns, a_ptr_fns+1, a_ptr_fns+2 };

  for (int i{}; i < SIZE; ++i) {
    std::cout << (*appf[i])(2.0) << ' ';
    //std::cout << (**appf[i])(2.0) << ' '; // ok
  }
  std::cout << '\n';
}

int main() { test_trig_fns2(); }
```

using: Example 3 with Function Pointers and Arrays


```
const int SIZE{3};
//double (*a_ptr_fns[SIZE])(double) {std::sin, std::cos, std::tan};
using PTR_FUNC = auto (*)(double) -> double;
PTR_FUNC a_ptr_fns[SIZE] { std::sin, std::cos, std::tan };

void test_trig_fns3() {
  //double (*(*papf)[SIZE])(double) {&a_ptr_fns};
  using PTR_A3F = PTR_FUNC (*)[SIZE];
  PTR_A3F papf {&a_ptr_fns};
  for (int i{}; i < SIZE; ++i) {
    std::cout << (*papf)[i](2.0) << ' ';
    //std::cout << (*(*papf)[i])(2.0) << ' '; // ok too ...
  }
  std::cout << '\n';
}

int main() { test_trig_fns3(); }
```