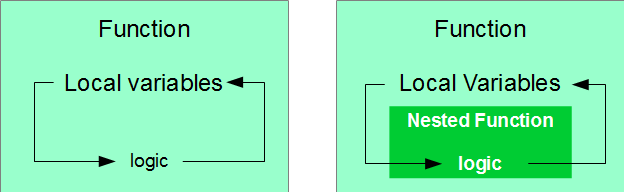
**Functions**

Object-oriented languages use function types to represent behavior.  Function types implement cohesive blocks of logic.  They reside in memory alongside other types and may have global, class or function scope.  Well-designed functions exhibit high cohesion and low coupling.  High cohesion refers to focus on a single task.  Low coupling refers to a minimal interface with other functions.

Function types may be overloaded, inherited, derived, templated, and nested.  Nested types can access variables within the scope of their host function.  Such types couple their logic to their nesting environment and are known as *closures*.  C++ supports closures in the form of lambda expressions.  C++ also supports function types in the form of function objects or *functors* by overloading the function call operator (()) on any class.



This chapter describes the support that C++ offers for function types.  It covers function linkage across modules, recursion, pointers to functions, function objects and lambda expressions.  We use lambda expressions for simple logic that is only required locally, function pointers for more complex logic that does not depend on anything but function parameters, and function objects for more complex logic that needs to maintain specific states.

Function Syntax

C++11 introduced syntax to specify program instructions in as simple and least-repetitive way as possible.  These syntactic improvements to C++98 included:

* type-inference declaration
* trailing return type declaration

Trailing Return-Type Declaration

A C++ compiler needs sufficient information about a function to infer its return type.  Sometimes this information is in the identifier and sometimes in the parameter list itself.  A return-type inference takes the form

auto identifier(parameter-type-list) -> return-type;

The auto keyword identifies the declaration as one with a return type inference.  The return-type follows the closing parenthesis and depends on the identifier or the parameter list.  This is particularly useful with

* enumerations that have been defined within a class
* templated functions

Enumeration defined within a class

Consider the function get() in the following program.  When the compiler starts processing its definition the return type (TicketType) is unknown.  TicketType is defined in class scope.  Initially, the compiler is unaware that the return type is defined inside the Ticket class.  Only after the compiler becomes aware from the identifier Ticket::get() that the function is a member of the Ticket class can it accept the return type:

|  |  |
| --- | --- |
| // Trailing Return - Enumeration within a class  // auto\_return.cpp  #include <iostream>  class Ticket {  public:  enum class TicketType { Adult, Child };  void set(TicketType tt);  TicketType get() const;  private:  TicketType ticketType;  };  std::ostream& operator<<(std::ostream& os,  const Ticket::TicketType& tt) {  const char\* label = nullptr;  switch(tt) {  case Ticket::TicketType::Adult:  label = "Adult Ticket";  break;  case Ticket::TicketType::Child:  label = "Child Ticket";  break;  default:  label = "No Ticket";  }  os << label;  return os;  }  void Ticket::set(TicketType tt) { ticketType = tt; }  auto Ticket::get() const -> TicketType { return ticketType; }  int main () {  Ticket a, b;  a.set(Ticket::TicketType::Adult);  b.set(Ticket::TicketType::Child);  std::cout << a.get() << std::endl;  std::cout << b.get() << std::endl;  } | Adult Ticket  Child Ticket |

Without auto and the trailing return-type in the definition of get(), we need to scope the return type (TicketType) by adding the class name; that is, to write:

|  |
| --- |
| Ticket::TicketType Ticket::get() const { return ticketType; } |

instead of:

|  |
| --- |
| TicketType Ticket::get() const { return ticketType; } |

In other words, the trailing return type simplifies the syntax for such cases.

Templated Functions

The trailing return type syntax is particularly useful with templated functions.  In the following program, the return type of the function add() is unknown to the compiler when it starts processing the definition and is only known after the compiler has finished processing the parameter list (const T& t, const U& u).  The return type is the type of the result of evaluating expression t + u.  To obtain the type of this result, we use the decltype() specifier:

|  |  |
| --- | --- |
| // decltype with templates  // decltype.cpp  #include <iostream>  template<typename T, typename U>  auto add(const T& t, const U& u) -> decltype(t + u) { return t + u; }  int main() {  int i = 3, j = 6;  double x = 4.5;  std::cout << add(i, j) << std::endl;  std::cout << add(i, x) << std::endl;  } | 9  7.5 |

In the first case the expression evaluates to an int type; in the second case, the expression evaluates to a double.  In both cases, the return type is the type returned by operator+(const T&, const U&).

Linkage

A function type can have either external or internal linkage.  A function with external linkage is visible outside its translation unit, while a function with internal linkage is invisible outside its translation unit.

The default linkage for a function type in C++ is external.  That is, the keyword extern is redundant.  The main() function of every application must have external linkage.

To specify internal linkage, we preface the function declaration with the keyword static.  For example,

|  |  |
| --- | --- |
| // Function Linkage - Module B  // linkage\_b.cpp  #include <iostream>  static void display() { // internal linkage  std::cout << "in module b\n";  }  void module\_b() { // external linkage  display();  } | in module a  in module b |
| // Function Linkage - Main Module  // linkage\_a.cpp  #include <iostream>  static void display() { // internal linkage  std::cout << "in module a\n";  }  void module\_b(); // external linkage  int main() { // external linkage  display();  module\_b();  } |

The different definitions of display() in the two modules do not conflict, since each display() has internal linkage within its own translation unit.

Recursion

A recursive function is a function that calls itself from within its own body.  Recursive functions require an exit condition that defines when the recursion terminates.  Once it terminates control begins stepping back through the function call stack to the initial caller.  The exit condition prevents stack overflow caused by an ever increasing set of recursive calls.

The following program uses a recursive function to calculate factorials:

|  |  |
| --- | --- |
| // Recursive Functions  // fibonacci.cpp  #include <iostream>  unsigned factorial(unsigned x) {  return (x > 2u) ? x \* factorial(x - 1) : x;  }  int main () {  std::cout << "2! = " << factorial(2) << std::endl;  std::cout << "3! = " << factorial(3) << std::endl;  std::cout << "4! = " << factorial(4) << std::endl;  } | 2! = 2  3! = 6  4! = 24 |

The recursion stops once x drops to a value less than or equal to 2u.  To keep track of the return path, a recursive function can consume a significant amount of stack space.

Stack Space Alternative

Stack space is a precious resource.  The alternative to highly recursive logic is an iteration construct.  The function listed above can be rewritten as follows:

|  |  |
| --- | --- |
| // Recursive Functions  // iteration.cpp  #include <iostream>  unsigned factorial(unsigned x) {  unsigned result = 1u;  while (x > 1u)  result = result \* x--;  return result;  }  int main () {  std::cout << "2! = " << factorial(2) << std::endl;  std::cout << "3! = " << factorial(3) << std::endl;  std::cout << "4! = " << factorial(4) << std::endl;  } | 2! = 2  3! = 6  4! = 24 |

Function Pointer

Because functions reside in memory, they are addressible and their addresses can be stored in pointers.  A function pointer holds the address of a function type.  The address identifies the location in memory where control is transferred to start execution of the function's code.

The definition of a pointer to a function resembles that of a pointer to an object.  It takes the form

return-type (\*identifier)(parameter-type-list) [= fn];

return-type is the return type of the function, identifier is the name of the pointer to the function and (= fn) is the initial address, which is optional. .  The parentheses around the name identify the definition as a pointer definition rather than the declaration of a function prototype.  The brackets stand for optional.

To appreciate the parenthesized syntax, compare the following statements:

|  |
| --- |
| T\* ptrToObject; // 1  T (\*ptrToFunction)(T&); // 2  T\* function(T&); // 3 |

1. ptrToObject is the name of a pointer that holds the address of a region of type T
2. ptrToFunction is the name of a pointer that holds the address of a region of type T (\*)(T&)
3. function is the name of a function that receives a reference to an object of type T& and returns the address of a region that holds an object of type T

A function pointer can hold the address of any region of memory that contains instructions for a function that receives a reference to an object of type T and returns the value of an object of type T.

To define the pointer ptrToFunction and initialize it to the address of function fn, we simply write

|  |
| --- |
| T (\*ptrToFunction)(T&) = fn;  T (\*ptrToFunction)(T&) = &fn; |

The & is optional since fn implicitly converts to &fn.

To change the function pointed to by ptrToFunction to gn, we simply write

|  |
| --- |
| T (\*ptrToFunction)(T&) = fn;  // ...  ptrToFunction = gn; |

The following program sorts an array of elements of type T using two different comparison criteria: descending order and ascending order.  Separate functions define the different criterion.  A function pointer receives the address of the function for the selected criterion in the last parameter of the sort function:

|  |  |
| --- | --- |
| // Function Pointers  // function\_pointers.cpp  #include <iostream>  // ascending order comparison  template <typename T>  bool ascending(T a, T b) { return a > b; }  // descending order comparison  template <typename T>  bool descending(T a, T b) { return a < b; }  // bubble sort  template <typename T>  void sort(T\* a, int n, bool (\*comp)(T, T)) {  for (int i = n - 1; i > 0; i--) {  for (int j = 0; j < i; j++) {  if (comp(a[j], a[j+1])) {  T temp = a[j];  a[j] = a[j+1];  a[j+1] = temp;  }  }  }  }  template <typename T>  void display(T\* a, int n) {  for (int i = 0; i < n; i++)  std::cout << a[i] << ' ';  std::cout << std::endl;  }  int main() {  int a[] = {1, 5, 2, 3, 6, 7, 2};  int n = sizeof a / sizeof (int);  sort(a, n, ascending<int>);  display(a, n);  sort(a, n, descending<int>);  display(a, n);  } | 1 2 2 3 5 6 7  7 6 5 3 2 2 1 |

comp is the name of the comparision function used within the sort function.  The type specialization <int> in the call to the sort function addresses the ambiguity regarding the type to be compared.  Without this specialization, the compiler cannot determine for which type to generate comparison code.

Arrays of Pointers to Functions

If several functions share the same return types and the same ordered set of parameter types, we may store their addresses in an array of pointers to functions.  Each element of such an array points to one of the functions in the set or to nullptr address.

The definition of an array of pointers to functions takes the form

return-type (\*identifier[n])(parameter-type-list) = { initialization-list };

where n is the number of elements in the array and the optional = { initialization-list } is a comma separated list of function addresses enclosed within braces.

The sort example above may be rewritten in terms of an array of pointers to functions:

|  |  |
| --- | --- |
| // Function Pointers  // array\_function\_pointers.cpp  #include <iostream>  // ascending order comparison  template <typename T>  bool ascending(T a, T b) { return a > b; }  // descending order comparison  template <typename T>  bool descending(T a, T b) { return a < b; }  // bubble sort  template <typename T>  void sort(T\* a, int n, bool (\*comp)(T, T)) {  for (int i = n - 1; i > 0; i--) {  for (int j = 0; j < i; j++) {  if (comp(a[j], a[j+1])) {  T temp = a[j];  a[j] = a[j+1];  a[j+1] = temp;  }  }  }  }  template <typename T>  void display(T\* a, int n) {  for (int i = 0; i < n; i++)  std::cout << a[i] << ' ';  std::cout << std::endl;  }  int main() {  int a[] = {1, 5, 2, 3, 6, 7, 2};  bool (\*criterion[2])(int, int) = {ascending,  descending};  int n = sizeof a / sizeof (int);  for (int i = 0; i < 2; i++) {  sort(a, n, criterion[i]);  display(a, n);  }  } | 1 2 2 3 5 6 7  7 6 5 3 2 2 1 |

Since the array of pointers declaration specifies the int type, there is no need to specialize the initial values.

Function Objects

A function object is an object-oriented representation of a function.  In C++, a function object is also called a functor.  The term originates in mathematics, specifically category theory, and may be interpreted as the pattern for a type that allows a function without changing the structure of the type.  In C++ terms, we can define a functor as the class of which function objects are instantiated.  That is, a function object is an instance of a functor.  Some writers equate the term functor with function object in C++.

The class that defines the structure of a function object in C++ overloads the function call operator.  The constructor for that class accepts the state for the function object.  Like an instance of a class and unlike a function pointer, a function object can store state, which may affect the result of a call to the represented function.  We use function objects to perform the same operation in several different parts of an application with possibly different states.

Let us solve the above sort example using a function object that holds the comparison direction (ascending or descending) as a state variable:

|  |  |
| --- | --- |
| // Function Objects  // function\_object.cpp  #include <iostream>  // order options  enum class Order { ascending, descending };  // order comparison  template <typename T>  class Compare {  Order order;  public:  Compare(Order o) : order(o) {}  bool operator()(T& a, T& b) const {  return order == Order::ascending ? a > b : a < b;  }  };  // bubble sort  template <typename T>  void sort(T\* a, int n, const Compare<T>& comp) {  for (int i = n - 1; i > 0; i--) {  for (int j = 0; j < i; j++) {  if (comp(a[j], a[j+1])) {  T temp = a[j];  a[j] = a[j+1];  a[j+1] = temp;  }  }  }  }  template <typename T>  void display(T\* a, int n) {  for (int i = 0; i < n; i++)  std::cout << a[i] << ' ';  std::cout << std::endl;  }  int main() {  int a[] = {1, 5, 2, 3, 6, 7, 2};  int n = sizeof a / sizeof (int);  sort(a, n, Compare<int>(Order::ascending));  display(a, n);  sort(a, n, Compare<int>(Order::descending));  display(a, n);  } | 1 2 2 3 5 6 7  7 6 5 3 2 2 1 |

Each function object holds the sorting order selected by the caller.

Lambda Expressions

A function object that is only used in a local area of an application (that is, within a function) can be represented by a lambda expression.  A *lambda expression* does not require an identifier and is shorthand for a function object.  If the expression is used more than once it can be given a name and referred to by that name within the body of its host function.  A lambda expression can capture variables within the scope of its host function.  It consists of its own body and a capture-list that references the non-local variables accessed by statements in that body.  We call a lambda expression with its referencing environment a *closure*.  A closure, unlike a function pointer, has direct access to non-local variables.

A lambda expression takes the form

[capture-list](parameter-declaration-clause)-> optional-return-type {

// function body

}

capture-list is an optional comma separated list of the capture specifications for the non-local variables accessed by the function body.  parameter-declaration-clause is a comma separated list of the parameters that receive values of local variables within the scope of the function body.

Capture List

The capture list of a lambda expression is the mechanism for passing all non-local variables to the body of the lambda expression.

Empty List

The simplest lambda expressions do not rely on variables within their environment.  They have empty capture lists.

[] denotes no capture.

In the following examples, the compiler deduces the return value from each of the lambda expressions:

|  |  |
| --- | --- |
| // Lambda Expressions  // lambda.cpp  #include <iostream>  int main() {  // lambda expressions  auto hello = []() { return "Hello World"; };  auto add4 = [](int i) { return i + 4; };  auto sub4 = [](int i) { return i - 4; };  // calls on the lambda expressions  std::cout << hello() << std::endl;  std::cout << add4(10) << std::endl;  std::cout << sub4(10) << std::endl;  } | Hello World  14  6 |

Capture by Value

[=] denotes capture by value.

In the following examples, each lambda expression captures the non-local variable k by value.  The template syntax provides the mechanism for receiving a lambda expression in a function definition.  Here, the function parameter func receives the expression:

|  |  |
| --- | --- |
| // Lambda Expressions  // capture\_by\_value.cpp  #include <iostream>  template<typename Func>  int add(int i, Func func) { return func(i); }  template<typename Func>  int sub(int i, Func func) { return func(i); }  int main() {  int k = 4;  std::cout << add(10, [=](int i){ return i + k; }) << std::endl;  std::cout << sub(10, [=](int i){ return i - k; }) << std::endl;  } | 14  6 |

Capture by Reference

[&] denotes capture by reference.

In the following examples, the lambda expressions capture the non-local variable k by reference, which enables a change to its value from within each expression:

|  |  |
| --- | --- |
| // Lambda Expressions  // capture\_by\_reference.cpp  #include <iostream>  template<typename Func>  int add(int i, Func func) { return func(i); }  template<typename Func>  int sub(int i, Func func) { return func(i); }  int main() {  int k = 4;  std::cout << add(10, [&](int i){ return i + k++; }) << std::endl;  std::cout << "k = " << k << std::endl;  std::cout << sub(10, [&](int i){ return i - k++; }) << std::endl;  std::cout << "k = " << k << std::endl;  } | 14  k = 5  5  k = 6 |

Exceptions to Capture Defaults

Exceptions to the capture list defaults can be added to a capture-list.  For example,

* [=](...) - captures all non-local variables by value
* [&](...) - captures all non-local variables by reference
* [=,&x,&y](...) - captures x and y by reference, all else by value
* [&,x,y](...) - captures x and y by value, all else by reference
* [x,&y](...) - captures x by value and y by reference
* [this](...) - captures this by value

The keyword mutable allows modification of non-local variables captured by value.  A lambda expression that allows such modification takes the form

[capture-list](parameter-declaration-clause) mutable -> optional-return-type{

// function body

}