Pointers Part 3

Functions and Heap Memory

garth@ggilmour.com

Using Pointers With Functions

- As with arrays the name of a function is its address
  - The location in read only memory of the compiled assembly
- The declaration of a function pointer is complex
  - For example ‘int(*f_ptr)(int,double)’ declares a pointer to a function that takes an int and a double and returns an int
- Brackets around ‘f_ptr’ are essential
  - To avoid declaring a function called ‘f_ptr’
    - Which takes two parameters and returns a pointer
- To call a function through a pointer just use ‘( )’
  - E.g. given the declaration above ‘f_ptr(27,3.6)’

```c
#include<stdio.h>
void funcOne(int i) {
		printf("funcOne called with %d\n", i);
}
int funcTwo(int i, double d) {
		printf("funcOne called with %d and %f\n", i, d);
		return 7;
}
double funcThree(char * str) {
		printf("funcOne called with %s\n", str);
		return 7.89;
}
int main() {
		void (* ptr1)(int) = funcOne;
		int (* ptr2)(int, double) = funcTwo;
		double (* ptr3)(char *) = funcThree;

		ptr1(10);
		ptr2(11, 12.34);
		ptr3("ABCD");
}
```

```
funcOne called with 10
funcOne called with 11 and 12.340000
funcOne called with ABCD
```

Function Pointers and ‘typedef’

- Function pointer declarations are hard to read
  - But can be simplified using the ‘typdef’ keyword
- Typedef declares an alternative name for a type
  - E.g. given ‘typedef int * INT_PTR’ the declaration ‘INT_PTR ptr’ makes ‘ptr’ a pointer to an integer
- Consider ‘typedef void (*FUNC_INT_PTR)(int);’
  - Anything declared as being of type FUNC_INT_PTR will be a pointer to a function that takes an int and returns void
  - Note you could also say ‘typedef void FUNC_INT(int);’
    - In which case the pointer declaration would be ‘FUNC_INT * ptr;’

```c
#include<stdio.h>

typedef void (*FUNC_INT_PTR)(int);

void funcOne(int i) {
	printf("funcOne called with %d\n", i);
}
void funcTwo(int i) {
	printf("funcTwo called with %d\n", i);
}
void funcThree(int i) {
	printf("funcThree called with %d\n", i);
}
void doCallback(FUNC_INT_PTR callback, int param) {
	callback(param);
}
int main() {
	FUNC_INT_PTR ptr1 = funcOne;
	FUNC_INT_PTR ptr2 = funcTwo;
	FUNC_INT_PTR ptr3 = funcThree;

	doCallback(ptr1,10);
	doCallback(ptr2,20);
	doCallback(ptr3,30);
}
```

```
funcOne called with 10
funcTwo called with 20
funcThree called with 30
```

Function Pointers in C++

- Function pointers are more complex in C++
  - Firstly because C++ supports overloading
    - Multiple functions can share the same name as long as they differ in the number, type or order of parameters
  - Secondly because C++ is an OO language
    - Objects contain fields and methods
      - Which may be declared public or private
    - Public methods are ‘slots’ on the outside of the object
      - A method pointer lets you call a slot on a particular object
- Consider the declaration ‘void (*f_ptr)(int) = func;’
  - ‘f_ptr’ points to the ‘func’ that takes an int and returns void
  - As opposed to any other function with the same name

Function Pointers in C++

```
void func() {
	cout << "<void>func()" << endl;
}
int func(int i) {
	cout << "<int>func(<int>)" << endl;
	return i;
}
int func(int i,double d) {
	cout << "<int>func(<int><double>)" << endl;
	return i;
}
class MyClass {
public:
	int func() {
		cout << "<int>MyClass::func()" << endl;
		return 0;
	}
};
```

```
void main() {

	//The compiler can work out which function
	// we mean via the type of the pointer
	void(*f_ptr1)() = func;
	int(*f_ptr2)(int) = func;
	int(*f_ptr3)(int,double) = func;

	f_ptr1();
	f_ptr2(27);
	f_ptr3(27,8.3);

	//Show both ways of calling a method
	// via a pointer
	MyClass mc;
	MyClass *mc_ptr = &mc;
	int(MyClass::*m_ptr)() = &MyClass::func;
	(mc.*m_ptr)();
	(mc_ptr->*m_ptr)();
}
```

Constant Pointers

- Declaring parameters as constant pointers is useful
  - For parameters whose size is greater than that of a pointer
    - The value is no longer copied which improves performance
  - The ‘const’ signals to the client that the value wont be changed
- What ‘const’ means depends on where it is placed
  - ‘int * const ptr’ defines a pointer which is const
    - It always points to the same address but that data can be changed
  - ‘const int * ptr’ and ‘int const * ptr’ define pointers to const data
    - The address can be changed but not the data
  - ‘const int * const ptr’ defines a const pointer to const data

Allocating Heap Memory in C

- Pointers are mostly used to refer to heap memory
  - This is necessary when storage requirements can only be worked out a runtime, e.g. the number of records in a file
- Heap memory is allocated via 4 functions
  - Malloc, calloc, realloc and free
- Malloc returns a pointer to memory of a specified size
  - Where the size is determined by the ‘sizeof’ operator
  - E.g. ‘struct P * ptr = (struct P *) malloc(sizeof(struct P));’
- Note that the returned memory is not initialized
  - You can set all bytes to the same value using ‘memset’

Allocating Heap Memory in C

- Calloc returns an array of a specified type
  - E.g. ‘int * ptr = (int *)calloc(10, sizeof(int));’
  - Unlike ‘malloc’ all the memory is initialized to zero
- Realloc resizes an array whilst preserving its contents
  - Either extra memory after the current block is allocated to you or the contents are copied into a larger block
  - E.g. ‘int * new_ptr = (int *) realloc(old_ptr, new_size)’
- Free returns allocated memory to the heap
  - Calling free on a null pointer has no effect

Memory Problems in C

- Functions should not return pointers to local variables
  - When the frame is removed from the call stack the memory for the local variable is released and the pointer becomes invalid
- Its OK to return a pointer to heap memory
  - Although the client then has the burden of deleting it
- A wild pointer is a pointer holding a random address
  - Because you mistakenly think it has been initialized
  - Because you corrupted it via errors in pointer arithmetic
  - Because the memory was returned to the heap via ‘free’
- Forgetting to delete memory is not technically an error
  - Instead it is a leak, which if repeated could exhaust the heap

Allocating Heap Memory in C++

- C++ allocates heap memory using the ‘new’ keyword
  - It provides the extra service of initializing the memory for you
- There are two versions of this operator
  - The ‘new’ operator allocates memory for a single value
    - For example ‘int * i_ptr = new int;’
  - The ‘new [ ]’ operator allocates memory for an array of values
    - For example ‘int * i_ptr = new int[10];’
- Heap memory must be managed carefully
  - To delete a single value use ‘delete i_ptr;’
  - To delete an array of values use ‘delete [ ] i_ptr;’

Memory Problems in C++

- If you use ‘new [ ]’ you <u>must</u> de-allocate with ‘delete [ ]’
  - If you use ‘new’ then literally anything can happen
  - Typically only the first element will be removed
- Calling delete twice will corrupt the heap
  - However it is safe to call delete on a pointer set to zero
  - So always reset a pointer to zero after deletion if it is going to remain in scope as the program continues
- The ‘new’ operator will fail when the heap is used up
  - Prior to standardisation ‘new’ returned zero
  - In standard C++ an exception of type ‘std::bad_alloc’ is thrown
    - This is a class which inherits from ‘std::exception’

Pass By Reference

Improved Parameter Passing

Introducing References

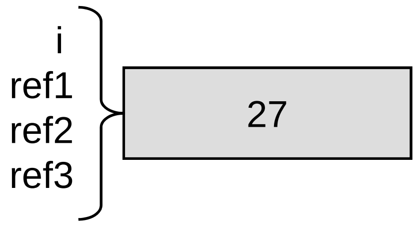
- The downside of pointers is the extra syntax
  - Continually having to remember to dereference the variable
- References were introduced as an alternative to pointers
  - Especially when it comes to avoiding pass by value
- A reference is an alternative name for a variable
  - It functions as an alias or synonym for the variable
- Unlike a pointer a reference has no separate existence
  - Once initialised it is indistinguishable from the variable is binds to
  - The address of the reference is the address of the variable

Declaring and Using References

```
//References don't work the same way as pointers...
int i = 27;
int & ref1 = i;
int & ref2 = ref1;
int & ref3 = ref2;

cout << "Address of i is " << &i << endl;
cout << "Addresses of references are " << &ref1 << " " << &ref2 << " " << &ref3 << endl;
cout << "Value of i is " << ref1 << " " << ref2 << " " << ref3 << endl;
```

```
Address of i is 0012FED4
Addresses of references are 0012FED4 0012FED4 0012FED4
Value of i is 27 27 27
```

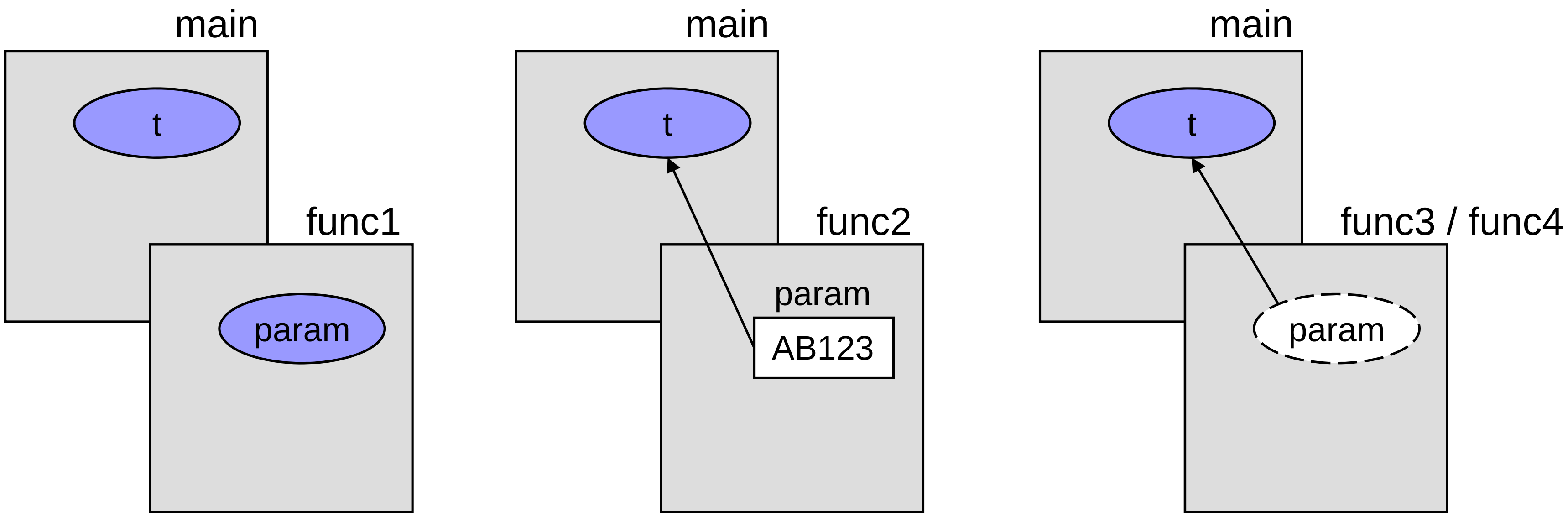


Benefits of References

- References provide the best of both worlds
  - The efficiency of passing by pointer combined with the convenience of pass by value
- The default way of passing non-basic parameters should be by constant reference
  - Use pass by value for basic types only
  - Use pass by pointer only to make it clear that the parameter lives in dynamically allocated memory
- The value returned from a function should be copied
  - Returning a reference or pointer usually leads to problems

```
//Pass by value - a copy is made
void func1(MyType param);
//Pass by pointer - the address is copied
void func2(MyType * param);
//Pass by reference - param is an alias
void func3(MyType & param);
//Pass by reference - param cannot be modified
void func4(const MyType & param);
```

```
int main() {
        MyType t;
        func1(t);
        func2(&t)
        func3(t);
        func4(t);
}
```



Using References

- Because a reference has no independent existence its declaration must also be its definition
  - As soon as it is declared it must be bound to a variable
- An initializer is not required when the reference:
  - Is a function parameter
  - Is the return type from a function
  - Is declared as a member of a class
  - Is declared with the ‘extern’ modifier
- A reference can be an alias for a pointer
  - Although mixing pointer and reference semantics is confusing

Combining References & Pointers

```
const char* str1 = "First test string";
const char* str2 = "Second test string";

//Swap the contents of two pointers using both
// pointer to pointers and references to pointers
void swapStrings(const char** str_ptr_ptr, const char* &str_ptr_ref) {

	const char* temp = *str_ptr_ptr;

	*str_ptr_ptr = str_ptr_ref;
	str_ptr_ref = temp;
}

void main() {
	swapStrings(&str1, str2);

	cout << "str1 now points to: " << str1 <<endl;
	cout << "str2 now points to: " << str2 <<endl;
}
```

Functions in C++

Resolving Overloaded Function Names

Function Overloading

- In C code functions are linked by their name
  - You cannot have two functions with the same name
- In C++ multiple functions can share the same name
  - They must differ in the type, number or order of their parameters
  - We refer to this as the signature of the method
- Functions that do the same thing in different ways can be grouped together for our convenience
  - So we can have ‘connect(string ip)’ and ‘connect(int ip)’
  - Rather than ‘connectByString(string ip)’ and ‘connectByInt(int ip)’

Overload Resolution

- Matching function calls to their correct definitions is known as Overload Resolution
  - Developers need to be familiar with how it works
- Overload Resolution occurs in many places:
  - In function calls to overloaded functions
  - When the function call operator is used
  - When a function pointer is used
  - When an expression uses overloaded operators
  - Anytime a constructor is called to initialize an object
  - When copy and conversion constructors are called
  - The rules are also important in exception handling

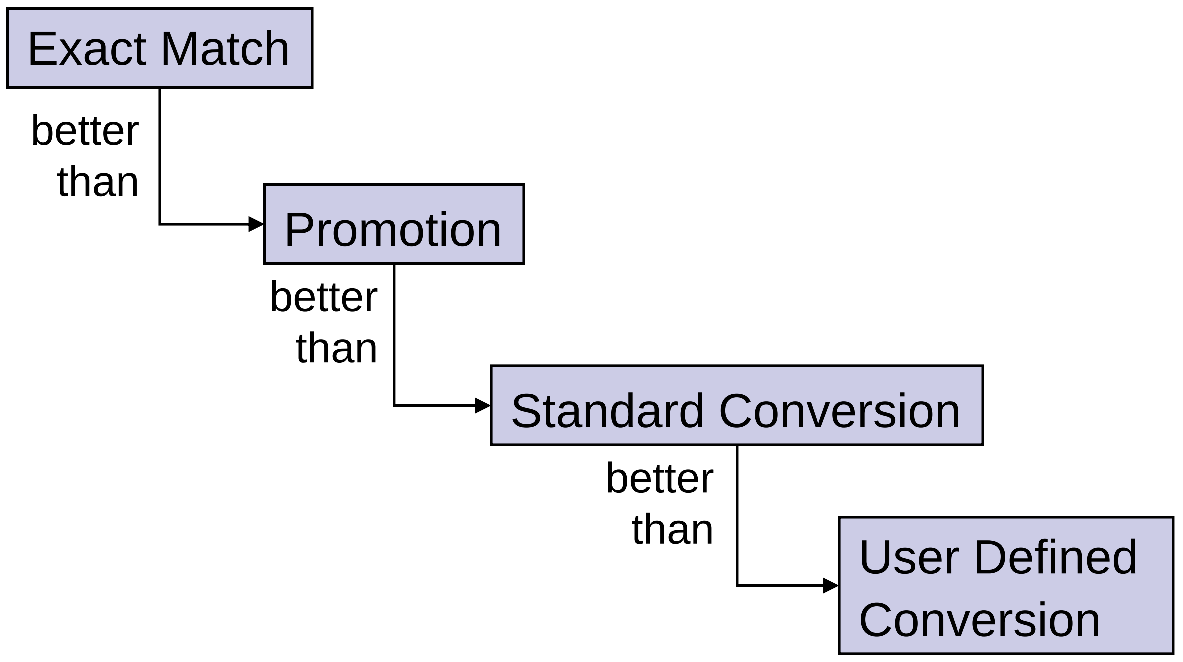
Resolving Overloaded Functions

- Name resolution in C++ is complex
  - It is often sidelined in introductory texts
- The procedure has three stages
  1. Identity the set of candidate functions
     - Those functions that match the name of the call
  2. Select the set of viable functions
     - Whose parameters match those used in the call
  3. Select the most viable function
     - The one whose parameters are the closest possible match
     - The complexity lies in finding the best possible match

Finding the Best Match

- Each parameter is ranked by asking:
  1. Is the parameter an exact match for the value supplied?
  2. Can the value be promoted to match the parameter?
  3. Can a standard conversion be used to convert the value?
  4. Is there a user defined conversion that can force a match?
- The result of this process determines the best match
  - Remember that the compiler views functions by signature
  - Each overloaded function just has the name part in common
  - Hence there is no problem with overloaded functions having the same return type - unlike virtual functions

Finding the Best Match



What is an Exact Match?

- Using a lvalue or rvalue of the specified type
  - E.g. calling ‘func(int)’ via ‘func(intVar)’ or ‘func(27)’
- Converting an array name to a pointer
  - E.g. calling ‘func(int *)’ via ‘func(intArray)’
- Converting a function name to a pointer
  - E.g. calling ‘func(void (*fptr)(int))’ via ‘func(fooBar)’
- A qualification conversion
  - E.g. calling ‘func(const float *)’ using a non const float pointer
- Where the argument is a valid initializer
  - E.g. calling ‘func(int &)’ via ‘func(myInt)’

Types of Conversion

- A standard conversion applies anywhere the range of the receiving type is greater
  - It is possible to convert from floating point to integer types
    - The fractional part of the number is lost
  - It is possible to convert from integer to floating point types
    - The converted value may lose some precision
  - Note that converting ‘0’ to a pointer is a conversion
    - If you call ‘func(0)’ where the candidates are ‘func(float *)’ and ‘func(int)’ the latter is always selected as an exact match
- User defined conversions are often constructors
  - The compiler will match ‘func(27)’ to ‘func(MyClass)’ if class ‘MyClass’ has a conversion constructor that takes an integer
  - This is something you want to be very careful about allowing

Namespaces

Partitioning C++ Code

garth@ggilmour.com

Namespaces

- Namespaces are fundamental to standard C++
  - They split your code into multiple declarative regions
  - A namespace may span any number of Translation Units
  - This is essential for any modern language
    - Where programs with hundreds of classes are common
    - UML packages map directly to C++ namespaces
- Unnamed namespaces replace static functions
  - An unnamed namespace is given an unknown name that is guaranteed to be different from all others in the program
  - Declarations inside the namespace cannot be discovered outside the current TU and hence all have internal linkage

Namespace Definitions

- Namespace members must be declared inside its scope
  - Definitions can be placed elsewhere
    - By qualifying the member with appropriate name
    - For example 'void MyNamespace::func(int i)'
  - The declaration must already be visible and the definition must occur inside an enclosing namespace
  - Unqualified names are said to be in the 'global namespace'
- A lesser known C++ feature is the namespace alias
  - For example 'namespace WRUS = Widgets_Are_Us'
- The standard libraries use the 'std' namespace
  - Header files drop the '.h' postfix to coexist with earlier versions
  - So prefer '#include <string>' to '#include<string.h>'

Using Declarations

- A using declaration introduces extra symbols into the current scope
  - Typically these symbols are from other namespaces or base classes
- The declaration typically occurs at the top of a '.cpp' or '.h' file
  - However it can also be used in class declarations

Declaration	Meaning
using namespace std	Bring in all visible symbols from namespace std
using ::foobar	Bring in symbol 'foobar' from the global namespace
using A::func	Bring in 'func' from the namespace or base class A
using A::B::C::func	Bring in the symbol from a nested namespace

Using Declarations

```
#include <iostream>
#include <string>

//bring in all symbols from std namespace
using namespace std;

namespace A {
   void print(string str) { cout << str << endl; }
}
namespace B {
   //bring in a single symbol from namespace A
   using A::print;

   class Base {
   public:
      void funcOne(int) {  print("Base::funcOne"); }
      void funcTwo(int) { print("Base::funcTwo"); }
   };
```

```
   class Derived : Base {
       //bring in funcOne from base class
      using Base::funcOne;
   public:
      void funcOne(char) {
         print("Derived::funcOne");
          //calls base class function
         funcOne(7);
      }
      void funcTwo(char) {
         print("Derived::funcTwo");
         //causes infinite recursion
         funcTwo(7);
      }
   };
}
```