

topics:

- functions: parameters and arguments
- call by value vs call by reference
- namespaces
- generic pointers
- dynamic memory allocation

resources:

- Pohl, chapter 3

functions: parameters and arguments

- function header declaration:

```
type name ( parameters );
```

- function definition:

```
type name ( parameters ) {  
    statements  
}
```

- function invocation:

```
name ( arguments );
```

or

```
variable_of_type = name ( arguments );
```

- functions have to be declared before they can be called
- the book uses the word "parameters" when a function is declared and "arguments" when a function is invoked (or "called")

- parameters can be either: *call by value* or *call by reference*

call by value

- when a function is called, the program control shifts from wherever the function call originates to the body of the function
- the function arguments get initialized as local variables within the function
- with *call by value*, the *value* of each argument is copied to a local variable within the function
- when the function ends, the program control returns to wherever the function was called from, and the memory allocated within the function returns to the program's memory stack
- even if the values of the local arguments within the function changed during the execution of the function, the values that were used to invoke the function do not change

- example:

```
// myfun1.cpp
#include <iostream>
using namespace std;

void myfun1( int a ) {
    a++;
    cout << "inside myfun1, a=" << a << endl;
} // end of myfun1()

int main() {
    int x = 7;
    cout << "before calling myfun1, x=" << x << endl;
    myfun1( x );
    cout << "after calling myfun1, x=" << x << endl;
}
```

and the output is:

before calling myfun1, x=7

inside myfun1, a=8
after calling myfun1, x=7

call by reference

- with *call by reference*, the *address* of each argument is copied to a local variable within the function
- when the function ends, the program control returns to wherever the function was called from, and the memory allocated within the function returns to the program's memory stack
- because the local arguments are addresses, any changes that were made to the values stored at these address locations during the execution of the function *are retained* when the function ends
- in C++, there are two ways to implement call by reference:
 - using references
 - using pointers

- example of call by reference using references:

```
// myfun2.cpp
#include <iostream>
using namespace std;

void myfun2( int *a ) {
    (*a)++;
    cout << "inside myfun2, a=" << *a << endl;
} // end of myfun2()

int main() {
    int x = 7;
    cout << "before calling myfun2, x=" << x << endl;
    myfun2( &x );
    cout << "after calling myfun2, x=" << x << endl;
}
```

and the output is:

before calling myfun2, x=7

```
inside myfun2, a=8
after calling myfun2, x=8
```

- example of call by reference using pointers:

```
// myfun3.cpp
#include <iostream>
using namespace std;

void myfun3( int &a ) {
    a++;
    cout << "inside myfun3, a=" << a << endl;
} // end of myfun3()

int main() {
    int x = 7;
    cout << "before calling myfun3, x=" << x << endl;
    myfun3( x );
    cout << "after calling myfun3, x=" << x << endl;
}
```

and the output is:

```
before calling myfun3, x=7
```

```
inside myfun3, a=8
after calling myfun3, x=8
```

another example

```
// myfun4.cpp
#include <iostream>
using namespace std;

void myfun4( int a, int *b, int &c ) {
    a++;
    (*b)--;
    c = a + (*b);
    cout << "inside myfun4, a=" << a << " b=" << *b
        << " c=" << c << endl;
} // end of myfun4()

int main() {
    int x = 7;
    cout << "before calling myfun4, x=" << x << endl;
    myfun4( x, &x, x );
    cout << "after calling myfun4, x=" << x << endl;
}
```

```
}
```

and the output is:

```
before calling myfun4, x=7  
inside myfun4, a=8 b=14 c=14  
after calling myfun4, x=14
```

passing arrays to functions

- given the following example:

```
int sum( int A[], int n ) {  
    int s=0;  
    for ( int i=0; i<n; i++ )  
        s += A[i];  
    return( s );  
} // end of sum()
```

- when the array A is passed to the function sum(), it is passed using call-by-value on it's base address (i.e., the address of A[0])
- note that within the context of a function header definition, the following two statements are equivalent:

```
int sum( int A[], ... ) { ... }
```

and

```
int sum( int *A, ... ) { ... }
```

but not in other contexts!

namespaces

- you have already been using namespaces as in:

```
#include <iostream>  
using namespace std;
```

- the std namespace is the standard C/C++ namespace that comes with the language
- a namespace is a way of grouping classes to avoid name conflict
- that is, you could have two things with the same name, but in different name spaces, and then there would be no conflict
- declaration of classes within a namespace looks like this:

```
namespace myspace {  
  
    class myclass1 { ... };  
  
    class myclass2 { ... };  
  
} \\ end of namespace
```

- note that when you define a namespace in a header file, you do not need to use the .h in the include statement:

```
#include <iostream>  
using namespace std;
```

versus

```
#include <time.h>
```

the first include statement is part of a namespace; the second is not

generic pointers

- last class, we talked about pointers to specific data types, e.g.,:

```
int *pi;  
char *pc;
```

- you can also have a pointer to a void:

```
void *pv;
```

- when you *dereference* the pointer, it is like converting it to that data type

- below are all legal statements, given the definitions above:

```
pv = pi;  
pv = pc;  
pi = reinterpret_cast<int*>(pv);  
*pi = 12;  
*pc = 'A';
```

dynamic memory allocation

- in C++, there are two functions that handle dynamic memory allocation: `new` and `delete`

- the syntax for `new` is:

```
new type-name  
new type-name initializer  
new type-name[expression]
```

- for example:

```
int *p, *q, *r;  
p = new int(5); // allocation and initialization  
q = new int[10]; // allocation, but uninitialized  
r = new int; // allocation, but uninitialized
```

- some compilers initialize values to 0 by default, but not all—that is not part of the language specification, so don't rely on it!

- the syntax for `delete` is:

```
delete expression
```

```
delete[ ] expression
```

- the first form is for non-arrays; the second form is for arrays

- example (from book, p137):

```
#include <iostream>  
using namespace std;  
  
int main() {  
    int *data;  
    int size;  
  
    cout << "\nenter array size: ";  
    cin >> size;  
    assert( size > 0 );  
  
    data = new int[size]; // allocate array of ints  
    assert( data != 0 );  
  
    for ( int j=0; j<size; j++ ) {
```

```
        cout << (data[j]=j) << '\t';  
    }  
    cout << "\n\n";  
    delete[] data;  
  
} // end of main()
```