Functions

PHYS2G03

McMaster University

Procedures, Subprograms, Subroutines, Methods and Functions

… All the same thing

- These names refer to a set of instructions that can be called independently from anywhere in a program
- This idea of jumping to a labeled place, doing instructions there and returning to where you started dates to the earliest machine code
- Machine code calls them subroutines
- **C/C++ calls them functions**

math.cpp

Pre-existing functions

Programming languages have available functions, particularly for math

e.g. y=sqrt(x) Function to return square root of a number

The code for these already exists and to use it you just need a header with sqrt declared, e.g.

#include <cmath> (C++ header style)

#include <math.h> (Note: .h indicates C style header, works for C/C++)

What about making your own functions?

How does C/C++ know it's a function? It is all about the brackets () {}

Variable Declaration:

```
int Z;
```

Function Prototype/ Declaration:

```
int Z();
```

No code for the function here – just telling the compiler there is a function called Z <u>and</u> <u>what it looks like on the outside</u>

Function definition:

```
int Z()   {  }
```

This contains the actual code for the function called Z inside the { } i.e. <u>what it does</u>

Your own functions: Compiling

```
int square( int x ) {
    return x*x;
}

int main() {
    int y;
    y = square(2);
}
```

Your own functions: Compiling

```
int square( int x ) {
    return x*x;
}

int main() {
    int y;
    y = square(2);
}
```

Definition of function square

square is defined *before* main. Defining a function also declares it. This would compile OK

Your own functions: Compiling

```
int main() {
    int y;
    y = square(2);
}

int square( int x ) {
    return x*x;
}
```

square is defined *after* main. Defining a function also declares it but it is too late for main.

This would **not** compile

Your own functions: Compiling

mysource.cpp

```
int square( int );


int main() {
    int y;
    y = square(2);
}
```

square is declared *before* main. **This is best practice**. Now it does not matter where square is defined (could be in a different source file)

othersource.cpp

```
int square( int x ) {
    return x*x;
}
```

Your own functions: Compiling

mysource.cpp

```
#include “square.h”

int main() {
     int y;
     y = square(2);
}
```

header files contain declarations (also called prototypes). The #include statement literally copies text from square.h into your program

othersource.cpp

```
int square( int x ) {
      return x*x;
}
```

square.h

```
int square( int );
```

C/C++ Function Terms

A function is a self-contained block of code:

It has an **argument list** of values passed in with specified types (can be left empty - meaning none)

- In good programming style this is the only data that the function uses

It has a **return value** of a specified type (can be *void* meaning none)

- This is the primary way to return information. Another ways is to modify data indicated by the arguments.

C/C++ Function Terms

A function is a self-contained block of code.

So the argument values are usually ALL it knows. The rest of the program and any outside variable names do not exist as far as the function is concerned.

Function layout

```
typeR name( type1 arg1, type2 arg2,..., typen argn)
{
    typeA localA;
    typeB localB;
    typeR returnval;
    // code

    return returnval;
}
```

Function layout

```
typeR name( type1 arg1, type2 arg2,..., typen argn)
{
    typeA localA;
    typeB localB;
    typeR returnval;
    // code

    return returnval;
}
```

arguments:
variables that exist only in the function.
Values sent in from outside when function is called

Function layout

```
typeR name( type1 arg1, type2 arg2,..., typen argn)
{
    typeA localA;
    typeB localB;
    typeR returnval;
    // code

    return returnval;
}
```

return value
Expression must be same type as function itself

Function layout

```
type name( type1 arg1, type2 arg2, ..., typen argn)
{
    typeA localA;
    typeB localB;
    // code

    return returnval;
}
```

local variables
Only exist in the function.
No values sent from outside

Functions: Usage

```
int square( int x ) {
    return x*x;
}

int main() {
    int y;
    y = square(2);
    y = square(3)+2;
}
```

What happens here:
Function square starts
Variable x created for
while function active
Set x = 2
x*x is returned
x destroyed and only
value of x*x is returned
Now: y = 4

Example Functions

```
int square( int x ) {
    return x*x;
}

int main() {
    int y;
    y = square(2);
    y = square(3)+2;
}
```

How function works:

Function square starts again

Variable x created (again) for function

Set x to 3

x*x is returned   (9)

y = 9+2 = 11   now

Example Functions

```
int square( int x ) {
    return y+x;
}

int main() {
    int y=1;
    y = square(2);
}
```

This is an error, the function does not know about y
This is the concept of **scope**. The scope of a variable name is ONLY the block of code it is declared in.

Example Functions

```
int square( int x ) {
    int y;
    return y+x;
}

int main() {
    int y=1;
    y = square(2);
}
```

This is also a mistake, it will compile but this y has no connection at all to the other y
y above might be zero or maybe even garbage depending on compiler

Example Functions

```
[wadsley@phys-ugrad ~]$ more test.cpp
int square( int x ) {
  int y;
  return y+x;
}

int main() {
  int y;
  y = square(2);
}
[wadsley@phys-ugrad ~]$ c++ -Wall test.cpp -o test
test.cpp: In function ‘int square(int)’:
test.cpp:3:12: warning: ‘y’ is used uninitialized in this function [-
Wuninitialized]
   return y+x;
            ^
```

This is a mistake, it will compile but this y has no connection at all to the other y
Compiling with **–Wall (warn all)** tells you this

Void (Empty) Arguments

```
int theanswer(  ) {
    return 42;
}



printf(“The answer is %i\n”, theanswer() );
```

Void Functions: no return value

```
void printit( int x ) {
    printf(“%i\n”,x);
}



printit ( 5 );
```

Example Functions

(details later: struct, class, new types)

```
// Dot product of two vectors
float dot( vector a, vector b) {
    float d = (a.x*b.x + a.y*b.y + a.z*b.z);
    return d;
}

vector x,y;
float z;
z = 2*dot(x,y);
```

vector is a user defined type (in this case a struct)

Function Arguments: Pass by value

- In C/C++, function arguments result in temporary new variables – the values of the arguments are copied into these variables
- Changing the variables listed as arguments has no direct affect on the original values
- This is called passing by **value**

Example Functions

```
int square( int x ) {
    x = x*x;    // square x
    return x;
}

int y,z;
z=2;
y = square(z);
// y = 4,  z = 2 not changed
```

x is a new variable set to the value of the argument (z in this case)
Changing x has no effect on z

Function Arguments:
reference type: Pass by reference

- C++ can reference the original values using the memory location of the originals instead

  myfunc(int &x,   float &r)

C++ uses & *in the function argument declaration* to show explicitly that it wants a reference to the memory that holds the variable, not just the value

- An argument declared with & is called a **reference**

C++ by reference

```
void byvalue( int x ) {
  x = 2;
}
void byreference( int & x ) {
  x = 3;
}
int main()
{
  int A;
  A = 1;
  byvalue( A );
  byreference( A );
  }
```

by value vs. by reference

```
void byvalue( int x ) {
  x = 2;
}
void byreference( int &x ) {
  x = 3;
}
int main()
{
  int A;
  A = 1;
  byvalue( A );
  byreference( A );
  }
```

x is unrelated to A
– just a copy of its value
x has its own memory

A is not changed A=1 still

by value vs. by reference

```
void byvalue( int x ) {
  x = 2;
}
void byreference( int &x ) {
  x = 3;
}
int main()
{
  int A;
  A = 1;
  byvalue( A );
  byreference( A );
  }
```

x is the same as A
x uses A's memory
Any change to x
changes A

A changed, now A=3

reference.cpp

```cpp
void byvalue( int x ) {
  x = 2;
  std::cout << "byvalue:  x = " << x << "\n";
}

void byreference( int &x ) {
  x = 3;
  std::cout << "byreference:  x = " << x << "\n";
}

int main()
{
  int a;

  a = 1;
  std::cout << "(1)  a = " << a << "\n";
  byvalue( a );
  std::cout << "(1)  a = " << a << "\n";
  byreference( a );
  std::cout << "(1)  a = " << a << "\n";
}
```

```
cp –r /home/2G03/func ~/
cd func
make reference
reference
```

Function Arguments: Pass by reference

- C doesn't have reference types but can do the same thing with pointers. C++ references are a lot easier to use and safer.
- **C/C++:** myfunc(int *x, float *r)

Arguments declared with * are pointers, memory locations – pointers to where x and r are stored in memory

We will explore pointers more later.

I have provided referencec.c for comparison

We use & to convert a variable to a pointer to the variable memory, not just the value

We uses * to change the value at that memory location

C or C++ pointer version

```
void byvalue( int x ) {
  x = 2;
}
void bypointer( int * px ) {
  *px = 3;
}
int main()
{
  int a;
  a = 1;
  byvalue( a );
  bypointer( & a );
  }
```

C or C++ pointer version

```
void byvalue( int x ) {
  x = 2;
}
void bypointer( int *px ) {
  *px = 3;
}
int main()
{
  int a;
  a = 1;
  byvalue( a );
  bypointer( & a );
  }
```

px is a pointer to an int
*px is that int
* means “dereference” not multiply here

Send a pointer to a (a pointer is where in memory a is – so a can be changed)

C++ by reference

```
void byvalue( int x ) {
  x = 2;
}
void byreference( int & x ) {
  x = 3;
}
int main()
{
  int a;
  a = 1;
  byvalue( a );
  byreference( a );
  }
```

C++ by reference

```
void byvalue( int x ) {
  x = 2;
}

void byreference( int & x ) {
  x = 3;
}
int main()
{
  int a;
  a = 1;
  byvalue( a );
  byreference( a );
  }
```

x is a reference to an int
x references that int

References (and pointers): A new way to return information

If you can refer directly to a function argument, you can change information outside your function that way

It is the easiest way to change a lot of variables at once (e.g. sorting a list)

However, this is less obvious than using return values so you need to be careful not to mess with the outside program’s variables