Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 6

- Functions
- Variable Scope
- Memory Allocation in Functions
- Passing Parameters
- Default Arguments

Function Declarations

```
int add4(int a1, int a2, int a3, int a4);
```

- Functions must be declared before they are used
- Syntax:

  <type> <function-name> (<param-list>);

  <function-name> ::= <ident>
  <param-list> ::= ε | <list>   (ε = empty word, | means or)
  <list> ::= <type> <ident> | <type><ident> , <list>
- return type void indicates that nothing is returned
- empty parameter list: no parameters are used

Derivation of function declaration on whiteboard

Function Definitions

```
int add4(int a1, int a2, int a3, int a4);
{
  return a1+a2+a3+a4;
}
```

- Functions must be defined (possibly in a separate source file) if they are used
- Syntax:

  <type> <name> (<param-list>) { <statements> }
- Exit void functions with `return;`
- Values are returned by `return` <expr> `;`

  (type of expression must match function return type)
- Parameters are treated as local variables
- In C++, function definitions cannot be nested

Function Examples

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

// compute the square of x (x*x <= INT_MAX)
int square(int x) { return x*x; }

// recursively compute n! (n >= 0)
int fac(int n) {
  if (n <= 1) return 1;
  return n * fac(n-1);
}

int main() {
  cout  << square(3) << ' ' << fac(3) << endl;
  return 0;
}
```

More Function Examples

```
// compute the n-th Fibonacci number (n >= 0)
int fib(int n) {
  if (n <= 1) return n;
  return fib(n-1) + fib(n-2);
}

// compute the greatest common divisor of a,b > 0
int gcd(int a, int b) {
  int r = a % b;
  if  (r != 0) return gcd(b, r);
  return b;
}
```

Some Library Functions

```
int main(int argc, char *argv[]);  // execution starts here
void exit(int err);     // exit execution with an error code
                        // (0 usually indicates success)
int abs(int x);         // compute the absolute value of x
double sin(double x);   // compute the sine of x
double floor(double x); // round x down to nearest integer
```

To learn more about standard C library functions consult `http://www.cplusplus.com/ref/`

or `man stdio.h / stdlib.h / string.h / math.h`

More later. There will also be a lab devoted to the C library

Variable Scope

- Variables (and constants) have a lifespan from the time they are created until they are no longer used
- Local variables are declared within statement blocks enclosed by {}
- They are unknown outside the block
- Memory for them is allocated on the process stack and not automatically initialized
- When functions are exited, memory for local variables is released

Local Variable Scope

```
int main()
{
  int   uninitialized;
  float initialized = 22.0/7.0;
  float x = 2.0; // (*)

  { // nested block
    float initialized = 3.1415; // (**)
    float x; // masks x (*)

    x = 2*initialized;  // refers to variable (**)
  }

  x = 3.1415926; // changes x (*)

  for (int i=10; i >= 0; --i) { cout << '?'; }
  i = 5; // i unknown here! local to for block

  int i; // variables can be defined anywhere!
  for (i=10; i >= 0; --i) { }
  // i lives here! value is -1
}
```

Function Call Mechanism

- Uses process stack (Last-In-First-Out data structure)
- Stackpointer register (SP) in CPU points to next available byte in memory
- When a function is called the return address is first pushed onto the stack (e.g. store address (4 bytes on 32-bit machines) starting at the location SP points to, then add 4 to SP)
- Make room for local variables by increasing SP by a constant
- Evaluate parameters and store values in local variables (on stack)
- When returning, store result in register or on stack for the caller to be used
- Decrease SP and jump to stored return address

Example on whiteboard

Passing Parameters: Call-By-Value

```
void increment(int x) { ++x; }

int y=5;
increment(y);

// oops, that didn't work: y is still 5!
```

When a function `A f(B x);` is called via ... `f(e)` ... expression `e` is evaluated and its value is copied into the local variable `x`

Statements in the body of `f` act on this local copy and do not change values in the evaluated expression `e`

Passing Parameters: Call-By-Reference

```
void increment(int &x) { ++x; }

int y=5;
increment(y);
// that worked: y now 6
```

- A reference to a variable is passed to a function (in form of a memory address)
- Statements in the function body that act on the parameter change the variable that has been passed to the function
- Syntax: <call-by-ref-par> ::= <type> & <identifier>
- Side effects (e.g. function can return more than one value)
- Can only pass variables (because address is required)

Swap Function

```
void naive_swap(int &x, int &y)
{
  x = y;
  y = x;
}

void swap(int &x, int &y)
{
  int temp = x; x = y; y = temp;
}

int a=1, b=2;
naive_swap(a, b); // oops: a=b=2?!

a=1; b=2;
swap(a, b); // ok! a=2, b=1
```

Passing Large Objects

```
void do_something(T big) { ... }
...
T x;
do_something(x); // slow!
```

- Passing large objects by value wasteful: they are copied into local variable
- Better: const reference

```
void do_something(const T &big) { ... }
...
T x;
do_something(x); // equivalent but much faster!
```

Pros & Cons

Call-by-Value

+ Callee detached from caller, no direct side-effects

– Data is copied to a local variable. Can be time consuming

Call-by-Reference

– Side Effects; need to look at function declaration to see whether call-by-reference is used

– Only variables as parameters

+ Only reference is copied. Fast.
(const qualifier protects read-only parameters)

Default Arguments

```
void print(int value, int base = 10);

print(31); print(31,10);
print(31,16); print(31,2);

->  31 31 1f 11111
```

- Arguments can have default values
- Syntax in parameter list of function declaration:
  <type> <identifier> = <constant-expression>
- All default arguments must be in the rightmost positions
- Omitting arguments begins with the rightmost one

Default Argument Example

```
void foo(int a, int b=2, int c=3, int d=4);

foo();          is illegal
foo(x);         calls   foo(x,2,3,4);
foo(x,y);       calls   foo(x,y,3,4);
foo(x,y,z);     calls   foo(x,y,z,4);
foo(4,3,2,1);   calls   foo(4,3,2,1);

// illegal:
void bar(int a=1, int b, int c=3, int d);

// why? bar(x,y,z) would be ambiguous
```