Functions

(aka methods, subroutines, subprograms, procedures)

- When used or called, a function can be treated as a black box
  - User cares only about what the function does, not about how (performance and errors aside)
  - The function’s interface or prototype is a description of how to connect with or call the function

- Pass-by-value input (parameters) may be
  - Constants
  - Variables
  - Expressions

```c
int a = 100;
int b = 25;

50 10
a b
a+50 2*b
```

```plaintext
x ÷ y
```

x ÷ y
Functions
(methods, subroutines, subprograms, procedures)

Function call looks like a sequential statement (caller treats it like a black box).

Function definition must be concerned with call sequence. Execution resumes with the statement following the call.
Function Definition

Elements of a function

- **Return value type**
- **Function name**
  - Follows rules for all identifiers
  - 31 characters significant minimum
  - First character is a letter or _
  - Subsequent characters are letters, digits, or _
- **Argument list**
  - Empty or void
  - List of types and names
- **Body**
  - Local variables
  - Statements
  - May be intermixed in C++

```c
type name(arguments) {
  local variables;
  statements;
}
```

- **Functions without arguments**
  - C: type name(void)
  - C++: type name(void)
  - C++: type name( )
Function Return Values

Getting a value from a function

- The default return type is `int`
  - Functions returning an integer should be explicitly typed
  - The `void` return type specifies a function that does not return a value (i.e., a “procedure”)

- Functions fall into one of several categories
  - Function computes and returns a value from its arguments
  - Changes its arguments and returns a status as the functional value
  - Does not return a value
    - Changes it arguments
    - Changes global data or performs some action

- Functions terminate by “falling off the end” or calling `return`
  - `return` may be called anywhere in the function
  - Function return values are specified with the `return` statement
Function Examples

User defined functions

double calcPayment(double P, double R, int N)
{
    return P * R / (1 - pow(1 + R, -N));
}

double random()
{
    static double x = 0;
    x = x * (x + 1) % 2147483648L;
    return x;
}
Function Prototypes

Function declarations

- **Function prototypes** have three components
  - Name
  - Return value type
  - Argument list

- Information is placed in the compiler’s symbol table
  - No body implies that no code is generated
  - Permits the compiler to
    - verify usage: number of arguments and void or return value
    - type-check arguments
    - perform appropriate type conversions on arguments and return value

- Introduced in C++ and back-ported to ANSI C
  - C: if a function is called without a prototype, the compiler guesses
  - C++: if a function is called without a prototype, it will not compile
Function Calls

Invoking a function

- Functions are called by name followed by a parameter list
  - Explicit type information is not given in the function call
  - Empty parameter lists still require the parentheses
  - The return value may be ignored

- Note that function calls do not contain typing information

- Examples
  - double  payment = calcPayment(P, R, N);
  - double  payment = calcPayment(100000, 0.08/12, 360);
  - double  pseudo_random = random( );
  - calcPayment(100000, 0.08/12, 360);
Function Example

```c
double sqr(double x);  // prototype: declares function sqr

int main( )
{
  double y;           // without a declaration, the compiler
  y = sqr(2);  // assumes that sqr returns an int
    // and passes the 2 without change
}

double sqr(double x)  // defines function sqr
{
  return x * x;
}
```
Function Example

Functions first

double sqr(double x) // defines & declares function sqr
{
    return x * x;
}

int main()
{
    double y;

    y = sqr(2);    // 2 promoted to double before call
}
Functions

A review

- Functions are composed of four parts
  - Return value type
  - Name
  - Argument list
  - Body

- Function names are used in three places
  - Definition-- generates code
  - Prototype (declaration)-- entered into the symbol tabel, no code
  - Call (typing information is not present)

- Functions return when
  - The last statement in the body finishes
  - When a \texttt{return} is executed

- Functions that return values \textbf{must} do so by calling \texttt{return}
Storage Classes

A review

- **auto (local) variables**
  - Default or keyword `auto`
  - Storage allocated on stack upon block entry and deallocated upon block exit (brackets denote blocks: `{ variables; statements }`)
  - Initialized each time a block is entered
  - `auto type name;` or usually just `type name;`

- **Local static variables**
  - Keyword `static`
  - Storage allocated at program startup and remains throughout program execution
  - Values persist between function calls
  - Initialized only once
  - `static type name;"
Global Variables

Variables defined outside of functions

- File scope (program scope with external declarations)
- Values maintained throughout program execution
- Programmer may initialize; compiler initializes to 0
- Local variables supersede (i.e., hide) global variables
- Should be used sparingly

Used to:
- Reduce the number of parameters
- “Return” multiple values
- Allow widely “separated” functions to communicate
- Return status or error values

Problems with global data
- Shared data \textit{couples} functions-- must test & debug as a unit
- Contributes to name space pollution
Global / Local Variable Example

Local variables hide global variables with the same name

- Not a multiple definition error
  - Each counter variable is in a different scope
  - Local scope supercedes global scope (i.e., local variable hides global variable)

```cpp
int nlines = 10;
int counter = 100; // global variable

void function() {
    int counter = 200; // local variable
    cout << "nlines " << nlines << "counter " << counter;
}
```
extern Variables

Variable declarations: type information for the compiler

- External variable statements *declare* variables *defined* at global scope in another file (i.e., they expand the variables scope to the entire program)
- May not be initialized
- A variable may have multiple declarations as long as they are all the same
- Often placed in header files

```cpp
int counter = 100;
void increment()
{
    counter++;
}

file1.cpp

extern int counter;

int report()
{
    return counter;
}

file2.cpp
```
Variable Scope Example

Function / block scoping rules

<table>
<thead>
<tr>
<th>application.cpp:</th>
<th>functions.cpp:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void func1(int); // prototypes</td>
<td>extern int number; // defined in driver</td>
</tr>
<tr>
<td>void func2(int);</td>
<td>// visible in driver</td>
</tr>
<tr>
<td>int increment(void);</td>
<td></td>
</tr>
<tr>
<td>int number = 10; // visible in driver</td>
<td>void func1(int i) // visible in func1</td>
</tr>
<tr>
<td>int count = 0; // visible in driver</td>
<td>{ int j; // visible in func</td>
</tr>
<tr>
<td>int main( )</td>
<td></td>
</tr>
<tr>
<td>{ int count; // visible in main</td>
<td>void func2(int i) // not the same</td>
</tr>
<tr>
<td>int i; // visible in main</td>
<td>{ int j; // i &amp; j as above</td>
</tr>
<tr>
<td>} while (count != 0)</td>
<td></td>
</tr>
<tr>
<td>{ int i; // hides i above</td>
<td>int increment( )</td>
</tr>
<tr>
<td>} // visible in loop</td>
<td>{ static int count = 0; // persistent</td>
</tr>
<tr>
<td>} // visible in loop</td>
<td>return count++; // hides count</td>
</tr>
<tr>
<td>} // from driver</td>
<td></td>
</tr>
</tbody>
</table>

CS 1410 Chapter 5 Slide 16 of 42
Additional Storage Classes

Special-case storage classes

- **register**
  - A *suggestion* that a variable is heavily used and should be maintained in a fast CPU register
  - Modern register allocation algorithms generally do better than programmers and this storage class is not frequently used today

- **volatile**
  - Denotes a variable that may be changed by something other than the program
  - The variable is reloaded from memory into a register each time it is used
  - Turns off compiler register optimization
Address Operators

Memory content vs memory address

```c
int i;
int* p = &i;
i = 10;

*i is undefined
*p is undefined

0x0a000010

*i is 10
*p is 10
```

```c
int i;
int* p = &i;
i = 10;

*i is undefined
*p is undefined

0x0a000010

*i is 10
*p is 10
```

```c
int i;
int* p = &i;
i = 10;

*i is undefined
*p is undefined

0x0a000010

*i is 10
*p is 10
```
Pass By Value
Bitwise copy from the call to the formal parameter

```c
void func(int i);

int main()
{
    int a = 10;
    func(a);
}

void func(int i)
{
    i = i + 1;
}
```
“Something (as a sign or indication) that refers a reader or consulter to another source of information (as a book or passage)” (Merriam-Webster)

“A value that enables a program to indirectly access a particular data item, such as a variable or a record, in the computer's memory. . . .The reference is said to refer to the data item, and accessing that data is called dereferencing the reference.” (Wikipedia)
Pointers v References

Part 2

- Pointers and references are both addresses
- Pointers are simple, primitive data types
  - Addresses are exposed
  - Addresses may be operated on (i.e., they allow address arithmetic)
  - Must be explicitly dereferenced
- References wrap addresses
  - Addresses are NOT exposed
  - Addresses man NOT be operated on (address arithmetic disallowed)
  - Dereferencing is automatic
  - Compiler generates additional language-specific code to protect the address and to implicitly perform the dereferencing
Pass By Pointer

Pass by value where the value is an reference (three required changes)

- Pass by address is used
  - When a function must change its argument
  - To increase efficiency when passing large data types

```c
void func(int* i);

void main( )
{
    int b = 10;
    func(&b);
}

void func(int* i)
{
    *i = *i + 1;
}
```
Pass By Reference
Reference arguments are aliases for formal arguments

```c
void func(int& i);

int main()
{
    int c = 5;
    func(c);
}

void func(int& i)
{
    i = i + 1;
}
```

**Diagram:**
- Initial state: `c = 5`
- After `func(c)`: `c = 6`, `i` is an alias for `c`

**Notes:**
- When using `int& i` in `func()`, `i` becomes an alias for the `int` variable `c` in the `main()` function.
Pass By Reference Example

Swapping variables

- Pass the *address* of an object
  - Provides efficiency (especially for large objects)
  - Propagate parameter changes to the calling function
  - Creates an *alias* (i.e., two names for the same memory location)
  - Do not have the “safety” of a pass by value

- One place to modify syntax
  - Function definition (from value to reference variable)

```cpp
void swap(int& v1, int& v2)
{
    int temp = v1;
    v1 = v2;
    v2 = temp;
}

int main()
{
    int a = 10, b = 20;
    swap(a, b);
}
```
Swapping Two Variables

The classic pass by address example

```c
void swap(int* v1, int* v2)
{
    int temp = *v1;
    *v1 = *v2;
    *v2 = temp;
}

int main( )
{
    int a = 10, b = 20;
    swap(&a, &b);
}
```
Pass By Value: Structures

Bitwise copy

```c
struct part
{
    char type;
    int id;
};

void func(part p);

int main()
{
    part d = {'d', 10};
    func(d);
}

void func(part p)
{
    p.id = 1000;
}
```
Pass By Pointer: Structures

Pass by value where the value is an reference (i.e., an address)

```c
struct part
{
    char type;
    int id;
};

void func(part* p);

int main()
{
    part e = {'e', 10};
    func(&e);
}

void func(part* p)
{
    p->id = 1000;
}
```
Pass By Reference: Structures

Same picture, same results, different syntax

```c
struct part
{
    char    type;
    int     id;
};

void func(part& p);

int main()
{
    part f = { 'f', 10 };    
    func(f);    
}

void func(part&  p)
{
    p.id = 1000;    
}
```

---

**Time**

`define f`  `call func`  `run func`  `return`

- f 0x12
- f 0x12
- f 0x12
- f 0x12
Pass By Value and Structure Replacement

```c
struct part
{
    char type;
    int id;
};

void func(part p);

int main()
{
    part g = { 'g', 100 }; func(g);
}

void func(part p)
{
    part t;
    t.type = 'k';
    t.id = 10000;
    p = t;
}
```
struct part
{
    char type;
    int id;
};

void func(part* p);

int main()
{
    part h = {'h', 100};

    func(&h);
}

void func(part* p)
{
    part* t = new part;
    t->type = 'k';
    t->id = 10000;
    p = t;
}
Pass By Reference and Structure Replacement

```c
struct part
{
    char type;
    int id;
};

void func(part& p);

int main()
{
    part i = {'i', 100};
    func(i);
    func(i);
}

void func(part& p)
{
    part t;
    t.type = 'k';
    t.id = 10000;
    p = t;
}
```
Pass By Double Pointer and Structure Replacement

```c
struct part {
    char type;
    int id;
};

void func(part** i);

int main() {
    part j = {'j', 100};
    part* ptr = &j;
    func(&ptr);
}

void func(part** p) {
    part* t = new part;
    t->type = 'k';
    t->id = 10000;
    *p = t;
}
```
Pointers To Pointers

Multiple indirection

- Useful for allowing a function to change an argument that is already a pointer

```cpp
int i1 = 0, i2 = 20;
void func(int **pp)
{
    cout << **pp << endl;
    *pp = &i2;
    cout << **pp << endl;
}
int main()
{
    int *p;
    p = &i1;
    cout << *p << endl;
    func(&p);
    cout << *p << endl;
}
```
Overloaded Functions

C++ only feature

- Each function has the same (user-supplied) name
- **Must** have a unique *signature* (i.e., argument list)
  - Different number of arguments [ f(void), f(int i), f(int i, int j) ]
  - Different type of arguments [ f(int i), f(char* s), f(double d) ]
  - Different type of argument in corresponding positions [ f(int i), f(char* s); g(int i, char* s), g(char* s, int i) ]
- **Should** have similar usage, etc.
- Can **not** overload on return type
  - Return value may be ignored
  - Intrinsic or user-defined automatic type conversion
  - Older compilers required all overloaded functions to have the same return type
Default Arguments

C++ only feature

- Default values used when the user does not supply a value
- Must follow non-default arguments (in a left-to-right order)
- Example

  ▶ void size(int s = 0);
  - size(5) // no default, s = 5
  - size(); // s defaults to 0

  ▶ void f(int a, int b = 0, int c = 10);
  - f(1, 2, 3);
  - f(1, 2); // c defaults to 10
  - f(1); // defaults taken: b = 0, c = 10

- Cannot violate signature rules for overloaded functions

  ▶ void f(int i, int j = 0)
  ▶ void f(int i) // conflicts with default above
Recursive Functions

Essential elements of recursion

- **Direct recursion**: a function calls itself
- **Indirect recursion**: A calls B, B calls C, ..., Y calls Z, Z calls A
- **Must** have 3 features:
  - Branch (usually in an if) that makes the recursive call
  - Branch (usually in an if) that does not recurse (i.e., terminates the recursion) -- condition may be implicit rather than explicit
  - Input must change with each call
- **Theoretically**, recursion may be written as a loop
  - There is an existence proof of this – but it’s not a constructive proof
Recursion Example 1
(Print the digits of an integer one at a time)

```cpp
void forward(int number)
{
    if (number != 0)
    {
        forward(number / 10);
        cout << number % 10;
    }
}
```
Recursion Example 2
(Print the digits of an integer in reverse order)

```c++
void reverse(int number)
{
    if (number != 0)
    {
        cout << number % 10;
        reverse(number / 10);
    }
}
```
Graphical Representation

Activation records and statement sequencing

4th call

number == 0

3rd call

number == 1
3 forward(0)
4 cout(1)

2nd call

number == 12
2 forward(1)
5 cout(2)

1st call

number == 123
1 forward(12)
6 cout(3)

prints digits forward

number == 0

1st call

number == 1
5 cout(1)
6 reverse(0)

2nd call

number == 12
3 cout(2)
4 reverse(1)

3rd call

number == 123
1 cout(3)
2 reverse(12)

prints digits reversed
Functions with an unknown number of arguments at compile time

Functions must have at least one predetermined argument

- Variable arguments follow the fixed argument(s)
- Variable arguments are indicated with an ellipsis (…) in definitions and in prototypes
- Type checking is suspended
- `int printf(char* control, ...)`
- `float avg(int n, ...)`

Based on macros defined in `<stdarg.h>`

- `va_list` variable length argument list type
- `void va_start(va_list ap, lastfix);`
- `type va_arg(va_list ap, type);`
- `void va_end(va_list ap);`
Pointers To Functions

Dynamic function manipulation

- The address of a function is its entry point
- The name of the function, without any adornment, is its address
- “Configurable” algorithms

```c
int func1(char* str) { ... }
int func2(char* str) { ... }
int (*fp) (char* s) = func1;
int (*fa[10]) (char* s) = { func1, func2 };
fa[0] = func1;
fa[1] = func2;
fp ("Hello World");
fa[1]("Hello World");
```
The ANSI qsort Library Function

An implementation of the quick sort algorithm

```c
void qsort(void* base, size_t num, size_t size, int (*fcmp)(const void*, const void*))
```

- `base` address of first element in an array
- `num` number of elements in the array
- `size` the size in bytes of each array element
- `fcmp` a pointer to a function, which compares two array elements. Takes two void pointers for parameters and returns an integer: <0, ==0, or >0, if the first element is ordered before, the same, or after the second element