C-net Programming Language Reference

Manual

Rediet Bekele - rsb2179	- Manager
Kidus Mulu - km3533	- Language Guru
Kingsley Neequaye - kn2427	- Tester
William Oseghare -who2103	- Systems Architect
Bruk Zewdie - bbz2103	- Systems Architect

1. Introduction

C-net is a language for network programming based on a subset of C. It was developed to create a seamless way for users to implement network programming through succinct code and easy interaction with network I/O. C-net provides an abstract wrapper for network sockets as objects for reading and writing along with built-in methods for performing common manipulations. In doing so, the language simplifies I/O for succinct and clear code. Furthermore, C-net discards the complex semantics of dynamic memory allocation in C by abstracting pointers for the user. As a result, C-net stores data on the heap, except for primitives and references to items on the heap, like Java. However, unlike Java, C-net does not have automatic garbage collection—users must explicitly deallocate memory that was previously allocated

2. Lexical Conventions

2.1 Tokens

There are five classes of tokens: identifiers, keywords, literals, operators and delimiters. White space (tabs, new lines, and blank spaces) and comments are ignored, except when used to separate tokens.

2.1.1 Identifiers

Identifiers are names that are associated with a certain value (in the case of a variable) or an operation (in the case of a function). They are any sequence of letters, digits, and '_' that **do not** begin with a digit. No more than the first eight characters are significant.

Examples of valid identifiers in C-net include `myvar, my1var, _myvar, name2`, … while invalid identifiers are any names that start with a digit such as `5_myvar, 4`

2.1.2 Keywords

These are identifiers that are reserved for use by the language and cannot be used otherwise.

int	TCP
char	break
float	continue
struct	if
string	else
socket	void
for	new
while	delete
UDP	nopen

2.1.3 Literals

Literals are values embedded into the program. There are two types of literals: numeric literals (integer, character, and float) and string literals.

Numeric literal

Integer literals

An integer literal is a sequence of digits in decimal that may optionally be preceded by '-' to signify a negative value.

Character literals

A character literal is a single printable ASCII character (values 32-126 inclusive) enclosed in single quotation marks (with a few exceptions). For computational purposes such as addition, subtraction or comparison with other characters, it can be treated as the number entry of the character in the ASCII table.

Non-printable characters can be represented with a backslash character '\' followed by three digits signifying the octal value of the ASCII table entry number for the desired character. Some examples are presented below.

Character literal representation in C-net	ASCII character represented
'\012'	New line (line feed) character
'\009'	Horizontal tab character

Backslash ('\') denotes an escape sequence for non-printable characters.

C-net provides shorthand notations to represent some special characters. Below are the acceptable shorthand char literal representations in source code translated to their corresponding octal representation in the right column:

Escape sequence	Translated character representation
'\n'	'\012'
'\t'	'\009'

‘\\’	‘\134’
‘\0’	‘\000’

Floating point literals

A floating point literal is a sequence of digits (with an optional preceding ‘-’) followed by a single decimal point, and possibly followed by a sequence of digits. For example, 3.1415 and 20. would both be considered valid floating point literals whereas .15 and 1.16.2021 would not.

String Literals

A string is a sequence of character literals. For the sake of tokenization, they are a sequence of ASCII printable characters (including digits) enclosed within double quotes (“ ”).

If the string literal contains the escape character ‘\’, the following three characters must be digits that specify an entry in the ASCII table in a similar manner to character literals. The exception to this is that the character following the escape character can be an entry from the table above of special escape characters (i.e ‘n’, ‘r’, ‘\’ or ‘0’) which will be interpreted according to the numeric association presented in the table.

The following are all valid examples of string literals:

```
“Hello, World!”

“Hello,\040World!”

“C:\\\\Users”

“Each\nword\not\na\nnew\nline\n”
```

Any string literal is typed as the built-in type string (discussed later), and escaped characters are sequences as their ASCII equivalents.

2.1.4 Operators

The following table outlines the operators available in the C-net programming language:

Arithmetic	+	-	*	/	%	
Assignment	=	+=	-=			
Relational	>	<	>=	<=	==	!=
Logical	&&	||	!	~		
Expression	[]	.				

Note: the index operator ([]) needs an integer constant in between the two square brackets.

2.1.5 Delimiters

The final types of tokens in the C-net language are broadly classified as delimiters. These can be either one of

1) **Comments** : A string of characters beginning with /* and ending with */, **OR** A string of character beginning with // and ending with a new line character
2) **Whitespace**: Any whitespace character such as a new line, a horizontal tab or a carriage return.
3) Opening and closing **curly braces** "{" and "}"
4) Opening and closing **parentheses** "(" and ")"
5) **Commas** ","
6) **Semicolon** character ";

- Comments and whitespace are used to separate tokens in program source code, and are otherwise insignificant to the program.
- Opening and closing curly braces, on the other hand, have significance in delimiting scope and declaration of user-defined structs which are both discussed in more detail later on.
- Opening and closing parentheses can delimit function arguments and can be used in arithmetic expressions.
- Commas can be used to delimit array elements and function arguments.
- Similarly, a semicolon is used to delimit statements.

3. Types

In the C-net programming language, all identifiers, literals, and expressions must have a type associated with them. All operations have a predefined set of types that they operate on and return, and the compiler must throw an error if the user attempts to use an operator on identifiers, expressions or literals that are typed differently than what the operator is defined to handle. For this reason, strong type checking is an essential part of the C-net programming language. In general, the types in C-net can be classified as primitive data types and complex data types.

Grammar:

```
typ :
    CHAR  { () }
    | INT  { () }
    | FLOAT  { () }
    | STRING  { () }
    | SOCKET  { () }
    | STRUCT ID  { () }
    | typ LBRACKET RBRACKET { () }
```

3.1 Primitive data types

Primitive data types are all numeric types that have pre-defined properties and representations in memory. In addition, they are somewhat special in that the data they represent is stored on the stack at program runtime. The following are the primitive data types defined in the C-net language:

1. int

The int type stores numeric values in 4 bytes of memory. It is the type given to any integer literal or an identifier that is typed as an int.

2. char

The char type represents a character by its numeric entry in the ASCII table stored in 1 byte of memory. It is the type of a character literal or an identifier marked as type char.

3. float

The float type stores a double-precision floating point number in 64 bytes of memory, following the standard of double precision representation. It is the type of a floating point literal or an identifier marked as float.

3.2 Complex data types

There are several built in complex data types in C-net. What makes complex data types different in C-net is that the data they represent resides in heap memory at runtime. An identifier typed as a complex data type is a 64 bit sized reference to a location in memory (a pointer). The specific complex data type determines how the memory that it references is to be manipulated by operators and what it is referencing.

The following are the complex data types that are allowed in C-net.

1. **struct *struct_name***

The struct type is a collection of primitive data types and complex data types that are represented together in memory. The built in types **string and socket** are modeled on a struct with additional language-provided features.

A struct must be given a name when it is declared by the user. Along with its name, the user must specify the elements of a struct along with their corresponding type and names. These elements will be called the members or fields of the struct. In the following example,

```
//name and age are the members of the struct person.

struct person{
  string name;
 int age;
};
```

A member of a struct may also be of type struct, including a struct of type itself. This means that the user is able to nest structs in C-net given that the nested struct is declared and defined before the nesting struct.

Example:

```
struct person{
  string name;
 int age;
};

struct couple{
 struct person p1;
 struct person p2;
};
```

After it is defined by the user, any identifier declared to have a type of struct will become a reference to a location in memory the size of all the combined sizes of its members.

2. **string**

The opaque string type represents a sequence of characters, and is the type assigned to a string literal or an identifier marked as string. In terms of representation, it can be thought of as a struct which contains a length field of type int and a pointer to a block of memory which has a size equal to the length field of the string. This block will contain the characters of the string.

Immutability of strings

It is important to note that despite the underlying pointer structure and struct-like representation, **C-net strings are immutable**. The user does not have direct access to the members of the string or the underlying implementation. All manipulations that involve the string are provided through member *functions* that have no effect on the string. Functionality provided by string member functions include getting the length and capitalizing it, among others (see below). The functions that have an effect on the string, such as capitalization, always return a new string object and the original string is never modified.

3. **socket**

A socket is a complex data type that represents a specific network socket associated with the running program. Its representation in memory is struct-like, containing members for the socket descriptor that is used for lower-level operating system operations on the socket, a flag that specifies the type of the socket, a string buffer that is used to buffer data when sending and receiving using the socket, and metadata such as port number, protocol, and ip address. Two transport layer network protocols are supported by C-net, the **UDP and TCP protocols.**

Similar to the string object, **the socket object is not mutable** and access and manipulation is through built-in member functions.

4. **array**

An array is a collection of a number of items of the same type. The type of an array is based on the objects that it contains. For example, an array containing ints would be an int array.

An array can contain either primitive or complex data types. The user must specify how many of the type's objects an array will contain and the memory allocation will be done by the compiler according to how much a single object of that type requires and how many of that type an array contains.

Once created, an array is fixed in size. However, **the contents of the array are mutable.**

5. **function**

A function is a series of statements (defined formally later) that can be executed repeatedly with possibly varying inputs. A function has a type which includes the return type of the function and the number and type of inputs that the function is expecting. This type is encapsulated by the signature of the function.

The signature of a function is denoted by its return type followed by the function name and a comma separated list types and names that the function accepts enclosed in brackets. The following function signature, for example, is one for a function named echo that accepts a single argument string s and returns a string:

```
string echo(string s)
```

The body of the function would have to follow this declaration immediately as C-net does not allow separate function declaration and definition.

4. Operators

4.1 Operator Types

4.1.1 Arithmetic operators

Grammar:

```
expr:
    expr PLUS expr       { () }
    | expr MINUS expr      { () }
    | expr TIMES expr      { () }
    | expr DIVIDE expr     { () }
    | expr MOD expr        { () }
```

Arithmetic operators are binary and are left-to-right-associative. The *, /, and % have the same precedence which is higher than the precedence of + and -. + and - have the same precedence.

The following table summarizes the types of operands that operators are defined on, what the operation returns, and what the return value is

Arithmetic operator	Types on which operator is defined		Return type of operator	The output of the operation on the specified operands
+	int	int	int	Integer addition
	float	float	float	Floating point addition
	char	char	char	Returns the character represented by the sum of the ASCII representation of the two characters

	string	string	string	String concatenation
-	int	int	int	Integer subtraction
	float	float	float	Floating point subtraction
	char	char	char	Returns the character represented by the difference of the ASCII representation of the two characters
*	int	int	int	Integer multiplication
	float	float	float	Floating point multiplication
	string	int	string	String concatenation the given integer number of times
%	int / char	int / char	int	The remainder after integer division whose sign is the same as the dividend

4.1.2 Relational operators

Grammar:

```
expr:
    expr EQ expr         { () }
    | expr NEQ expr        { () }
    | expr LT expr         { () }
    | expr LEQ expr        { () }
    | expr GT expr         { () }
    | expr GEQ expr        { () }
```

The relational operators are <, >, <=, >=, ==, and != and are left-to-right-associative. <, >, <=, and >= have the same precedence, ==, and != have relatively lower precedence. All relational operators have lower precedence than arithmetic operators.

In addition, == and != are defined for strings and compares if the characters contained by two strings are the same.

4.1.3 Logical operators

Grammar:

```
expr:
    NOT expr { () }
    | expr AND expr
    | expr OR expr
    | BITWISE expr
```

The logical operators take one or two operands and return a value based on some logical operation. They have lower precedence than the relational operators.

The following table summarizes the logical operators available in C-net. Logical operators are defined on int and char types and they return an int type.

Operator	The output of the operation on the specified operands
~	An integer resulting from the bitwise not of all the 32 bits in the original integer
!	A 1 if the operand is 0 and a 0 otherwise
&&	A 1 if both the operands are non-zero and a 0 otherwise
||	A 1 if both either one or both of the operands are non-zero and a 0 otherwise

4.1.4 Expression operators

Grammar:

```
expr:
    LBRACKET expr RBRACKET { () }
    | expr DOT ID
```

Expression operators are '[]' and '.', they are left-to-right-associative, and have the highest precedence.

The '[]' is defined differently based on the type that it is operating on. *type* is used to show that the array can be of any type, and the return will be of the same type. For example, the return value of an index into a char array will be a char.

Operator	Types on which operator is defined		Return type of operator	The output of the operation on the specified operands
[*x*]	*type* array	int	*type*	The element at the index *x*
	string	int	char	The *x*-th character in the input string

The '.' operator is defined in user defined structs as well as the built in string and socket types, although the behavior is different for the two.

Operator	Types on which operator is defined	Return type of operator	The output of the operation on the specified operands
.name	*struct*	Type of *name*	The value of the field named *name* in the struct the operator is applied on
.name()	string, socket	Return type of *name*	Returns the result of invoking the function *name* on either the string or socket object that the operator is operating on

4.1.5 Unary operators

Grammar:

```
expr:
    MINUS expr %prec NOT { () }
```

The unary operator (-) assigns negates an int and has higher precedence than the arithmetic operators and is lower in precedence to the expression operators.

4.1.6 Assignment Operators

Grammar:

```
expr:
    ID PLUSEQ expr       { () }
    | ID MINUSEQ expr       { () }
    | structmem ASSIGN expr { () }

vdecl_assign:
    typ ID ASSIGN INTLIT SEMI { () }
    | typ ID ASSIGN NEW typ LBRACKET INTLIT RBRACKET
LBRACE INTLIT RBRACE SEMI { () }
```

The assignment operators are = -= and +=

Operator	Behavior
=	assigns an expression on the right hand side of the = to a variable or struct member on the left hand side of the =
+=	adds the expression to the right of it to the variable to the left of it and assigns the result to the variable. *var* += *expr* is semantically equivalent to *var* = *val* + *expr*
-=	Subtracts the expression to the right from the variable on the left and assigns the result to the variable. *var* -= *expr* is semantically equivalent to *var* = *val* - *expr*.

Assignment operators are right-to-left-associative, have the same precedence level to one another, and have lower precedence than the logical operators.

4.2 Operator Precedence & Associativity

C-nets operator's precedence and associativity is listed in the table below in descending precedence order:

Operator	Symbol	Associativity
Dot	.	Left
Not	~	Right
Mod	%	Left
Times, Divide	*, /	Left
Plus, Minus	+, -	Left
Relational Operators	> , < , >= , <=	Left
Equality operators	==, =!	Left
And	&&	Left
Or	||	Left
Assignment	=	Left

5. Expressions

An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.

Example: `x + y, 10.75, a = 4`

There are two classes of expressions: **lvalue and rvalue**.

5.1 lvalue

An lvalue expression is an expression that represents a location in memory and can appear on the left side of an assignment operator. In general, an lvalue expression can also act as an rvalue. An lvalue can refer to one of the following:

- Id: the name of the variable of any type that can be used as an identifier.
- The declaration of an identifier. Example: `x` is an lvalue , so is its declaration `int x`

- Members of a struct type. Example

```
struct Person {
  string name;
  int age;
};
```

E.g., if we had a person struct as follows:

```
struct person p1 = new struct person;
```

p1.name is a valid lvalue

5.2 rvalue

An rvalue expression is a value held in a memory location and can only appear on the right hand side of an assignment operator. The different types of expressions can be classified in the following manner.

5.2.1 Literal expressions

Grammar:

```
expr:
    INTLIT                { () }
    | CHARLIT             { () }
    | FLOATLIT            { () }
    | STRLIT              { () }
```

```
35.5 // float literal expression
'a'  // character literal expression
"goat" // string literal expression
35 // integer literal expression
```

These evaluate to and return their representation as discussed earlier in the manual.

5.2.2 Operation expressions

These are expressions that are obtained by applying one of the available operators on an expression or pair of expressions. The return value is the result of the expression and is also an expression. Possible examples include

```
a = b // assignment operator; returns a after assignment

x & 4 // and operator

y = (a + (b = 9)) // multiple operators; returns y
```

5.2.3 Array expressions

Grammar:

```
expr:
    arraylit { () }

arraylit:
        LBRACE args RBRACE { () }
args:
    expr { () }
    | args COMMA expr { () }
```

An array expression is a left square bracket followed by zero or more elements separated by commas. An array expression can only be used in the context of an array declaration (discussed below under declarations). E.g.,

```
[3.0, 4.0, 5.0] // array of floating point numbers
['a', 'b', 'c'] // array of characters
[] // An empty array
```

5.2.4 Function calls

Grammar:

```
expr:
    ID LPAREN args RPAREN
```

A function call is a function name followed by the arguments to be passed to the function separated by commas and enclosed with parentheses. The arguments passed to the function must match the function's signature both by number and type. An example function call might look like:

```
myfunc(5,2) // function must accept two arguments typed int
```

5.2.5 New expressions

Grammar:

```
expr:
    NEW typ { () }
    | NEW typ LBRACKET expr RBRACKET { () }
```

A new expression is the new keyword followed by the type of object being created. The object may be of struct type or an array. For structs, new must be followed by the keyword struct and the name of the struct. For eg.

```
new struct person // struct person must be defined
```

For arrays, new must follow the type, open bracket, and integer for the size of the array and a close bracket.

```
new struct person[10]; // an array of 10 struct persons
```

6. Declarations

Declarations in C-net bind a certain identifier with a type and possibly a value. The syntax of declarations is that a type is specified followed by an identifier and a semicolon. In general,

C-net allows mixing pure declarations with declarations and assignments by preceding the semicolon with an expression which has the same type as the identifier being declared. The exception to this are structs (see below).

Grammar:

```
vdecls:
      vdecls vdecl { () }
      | vdecl { () }
vdecl:
     typ ID SEMI { () }
```

E.g.

```
int x; // valid -- declares a variable x of type int

int y = 5; // also valid -- declares y and assigns it to 5
```

6.1 Primitive Declarations

A variable of a primitive type can be declared and assigned an expression of its type (including literals) as the example above demonstrates.

6.2 Non-primitive Declarations

6.2.1 Arrays

An array can be declared and assigned to another array or a new expression as follows.

```
int[] x = new int[15]; // 15 is the size of the array
```

Arrays can be populated in either of the following two ways.

1. by a sequence of statements indexing the array by position and assigning it to a given value after the array has been initialized using a new expression

   ```
   int[] x = new int[10];
   ```

```
x[0]= 1;

x[1]= 2; ...
```

or

2. by following the array declaration with the elements enclosed in curly braces

```
int[] x = new int[]{1,2,3,4}; //size need not be specified
```

6.2.2 Structs

```
sdecl:
     STRUCT ID LBRACE vdecls RBRACE SEMI { () }
```

A struct declaration is the keyword struct followed by a name, an open curly brace, one or more member declarations, a closing curly brace, and a semicolon.

E.g., a struct definition:

```
struct person {
  string name;
  int age; };
```

A variable of type *struct person* can then be created as follows:

```
struct person p1 = new struct person; //declare a struct person

p1.name = "Joe"; // assign the member name to "Joe"

p1.age = 40; // assign the member age to 40
```

Mixing assignment and declaration is not allowed when declaring a struct variable.

6.2.3 Sockets

Objects of the socket type are declared in C-net using the built-in *nopen* function. The signature of the *nopen* function is

```
socket nopen(string server_address, int port, protocol)
```

```
/* protocol can either be TCP or UDP */
```

E.g.

```
socket sock = nopen(string server_address, int port, protocol);
```

6.2.4 Functions

Grammar:

```
fdecl :
    typ ID LPAREN opt_params RPAREN LBRACE opt_stmts
RBRACE { () }
```

Function declarations in C-net are done by first stating the return type of the function, followed by the name of the function and an optional parameter list of comma separated parameters enclosed in parentheses. This is the signature of the function and must immediately be followed by its definition, which is an open curly brace, a collection of 0 or more statements, and a close curly brace. A function that has a type other than void is required to have a return statement at the end of the function. Return statements are discussed below in the statements section.

E.g.

```
int multiply(int x, int y) // this is the function's signature
{
     return x * y; // a return statement required
}
```

7. Statements

Statements are a sequence of C-net code ending in a semicolon followed by any number of statements.Unlike expressions, statements inherently don't have a value .

7.1 Expression statement

An expression statement is an expression followed by a semicolon (i.e., "expr;").

E.g., `5;` `2 + 2;` `X++;` `y = x + 25;`

7.2 Declaration statement

Declarations (discussed above in detail) are also a subclass of statements.

7.3 Control flow

7.3.1 if/else

Grammar:

```
stmt:
    IF LPAREN expr RPAREN stmt_gen ELSE stmt_gen    { () }
    | IF LPAREN expr RPAREN stmt_gen %prec NOELSE { () }
```

The forms of conditional statements are the following

```
if ( expression ) statement
if ( expression ) statement else statement
```

The expression is evaluated and if it is a non-zero, the statement following the expression is executed. If the user utilizes the second case, when the expression evaluates to a zero, the second sub-statement is implemented.

7.3.2 Loops

while loop

Grammar:

```
stmt:
    WHILE LPAREN expr RPAREN stmt_gen { () }
```

The while loop has the following form:

```
while ( expression ) statement
```

This means that the statement is executed repeatedly in a loop as long as the expression doesn't evaluate to zero. The expression is evaluated before each iteration of the loop.

for loop

Grammar:

```
stmt:
    FOR LPAREN opt_expr SEMI opt_expr SEMI opt_expr RPAREN
```

The for loop statement has the following form:

```
for ( expression-1opt ; expression-2opt ; expression-3opt )
statement
```

Expression-1 represents the initialization for the loop. The expression-2 is evaluated before each iteration and if it evaluates to a non-zero, expression-3 is executed. The loop is exited when expression-2 returns a zero.

7.3.3 break and continue

- The break statement terminates the flow in a while or for statement. The control after termination is passed to the statement following the terminated statement.
- The continue statement skips the current iteration of the loop and continues with the next iteration.

```
break ;
continue ;
```

7.4 Block Statements

Grammar:

```
stmt_gen :
    | LBRACE stmts_gen RBRACE { () } /* Block */
```

Block statement is a series of one or more statements enclosed by an opening and closing curly braces ({}). Block identifies the scope of the primitive type variables that are declared in it. Primitives are stored on the stack by default. Therefore, any primitive type declared within a block can't be accessed outside of the block statement. This is because the stack is rolled right before the closing curly brace removing all access to the primitive type data stored on the stack. In contrast, non-primitives are stored on the heap by default. Therefore, the user has to explicitly delete the identifier associated with the non-primitive type data within the given block in order to avoid a memory leak.

7.5 Return Statements

Grammar:

```
stmt:
    | RETURN opt_expr SEMI { () }
```

A function returns to its caller by means of the return statement. The return statement my return with no value as in the case of `return ;` or the statement might return with the value of an expression that is supplied to it in the form of `return (expression) ;`

8. Built-in functions

The list of functions available for the built-in string and socket types are given below, along with their return types. They are accessed the same as struct members and called the same as functions. If we have a string variable named *`str`*, *`str.length()`* is the way to call the length function.

For string:

Function signature	**What the function does**
int length()	Returns the length of the string it is called on
string upper()	Returns the string it is called on in upper case (only the alphabetic characters change)
string lower()	Returns the string it is called on in lowercase

int atoi()	Returns the integer represented by the string. If it is not a valid integer, it will return a 0
float atof()	Returns the float represented by the string
int find(char c)	Searches for c in the given string a and returns the index of the first occurrence
string substring(int b, int e)	Returns a substring of the string from index b inclusive to index e exclusive
String reverse()	Returns the reverse of the string it is called on

For socket:

Function signature	**What the function does**
string read_line(int max_length)	Reads from the socket a maximum of max_length characters or until a new line is read and returns a string which holds the content that was read.
int print_line(string s)	Sends the string s followed by a new line character into the socket it is operating on and returns the number of bytes successfully written.
string read(int max_length)	Reads up to a max_length number of bytes from the socket and returns a string containing that data
int write(string s)	Writes the string s with no new line character and returns the number of bytes successfully written

9. Programs

A program in C-net is a collection of declarations: variable declarations, function declarations and struct declarations. The compiler reads the source file from top to bottom, so any variable, function or struct has to be declared before its first use.

Every program must have at the very least a main function with the following function signature:

```
int main(string[] args)
```

This is the entry point of the program and if it is absent, the compilation will fail. Following are two examples of source code in C-net. The first is a function that calculates the GCD of two numbers (it would have to be contained in a larger program with a main method), and the second is a full program demonstrating a simple chat server written in C-net.

Sample 1: GCD algorithm using C-net

```
int gcd(int a,int b){
     while(a!=b) {
          if(a>b)
               a -= b;
          else
               b -= a;
     }
     return a;
}
```

Sample 2: A simple chat server using C-net

```
int main(string[] args){
     if(args.length() < 2) {
               stderr.print_line("No port provided");
               exit(1);
     }

     portno = args[1].atoi()
     socket listener = fopen(LISTEN, portno, TCP);

     /* blocks here */
     socket connected_sock = listener.wait_until_connection();

     if (connected_sock.error() > 0)
               error("error accepting new connection");

     while(1) {
               string message = connected_sock.read_line(255);
               if (message.length() == 0)
                         stderr.print_line("Error reading from
client");

               stdout.print("Client: ");
               stdout.print_line(message);
               if (message == "Bye")
                         break;
```

```
            stdout.print_line("Server: ");
            message = stdin.read_line(255);

            connected_sock.write(message);
            if (connected_sock.ERR != 0)
                    error("Error writing to server\n")
    }

    connected_sock.close();
    listener.close();
    return 0;
}
```