**No except-**

C++11 includes another concept based keyword ***noexcept*.** This keyword can be used for specifying that any function cannot throw — or is not ready to throw.

Here is a code snippet:

Ex-1 void test() noexcept;

|  |  |  |
| --- | --- | --- |
| **noexcept(** *expression* **)** |  |  |
|  | | | |

Returns a [prvalue](https://en.cppreference.com/w/cpp/language/value_category) of type bool.

This declares that test() won't be able to throw. If the exception is not deal with locally inside test() — and when test() method throws — the program will get terminated, calling std:: terminate() method that by default calls the std:: abort().

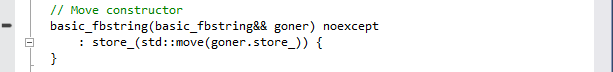
No except focuses on a lot of issues (empty) exception a specification is having

Runtime checking: Exceptions in terms of C++ get checked at runtime rather than at compile time, and hence they offer no programmer guarantees whether all exceptions got to be handled or not. The runtime failure mode (which internally calls std:: unexpected()) does not lend itself to recovery.

Runtime overhead: Checking at runtime needs the compiler producing additional code which may also hamper optimizations.

Unfeasible in generic code: Contained by any generic code, it is not usually possible to know what types of exceptions will get thrown on template arguments and hence an exact exception specification can't be written.

Ex-2

[](http://www.javadepend.com/Blog/wp-content/uploads/c9.png)

**Null ptr-**

Understanding nullptr in C++

Consider the following C++ program that shows problem with NULL (need of nullptr)

|  |
| --- |
| // C++ program to demonstrate problem with NULL  #include <bits/stdc++.h>  using namespace std;  function with integer argument  int fun(int N)   { cout << "fun(int)"; }  // Overloaded function with char pointer argument  int fun(char\* s)  { cout << "fun(char \*)"; }  int main()  {  // Ideally, it should have called fun(char \*),  // but it causes compiler error.  fun(NULL);  } |

***chevron\_right***

***filter\_none***

**Output:**

16:13: error: call of overloaded ‘fun (NULL)' is ambiguous

fun(NULL);

What is the problem with above program?

NULL is typically defined as (void \*)0 and conversion of NULL to integral types is allowed. So the function call fun(NULL) becomes ambiguous.

How does nullptr solve the problem?  
In the above program, if we replace NULL with nullptr, we get the output as “fun(char \*)”.

nullptr is a keyword that can be used at all places where NULL is expected. Like NULL, nullptr is implicitly convertible and comparable to any pointer type. Unlike NULL, it is not implicitly convertible or comparable to integral types.

|  |
| --- |
| // This program does NOT compile  #include<stdio.h>  int main()  {  int x = nullptr;  } |

Output:

Compiler Error

As a side note, nullptr is convertible to bool.

|  |
| --- |
| // This program compiles  #include<iostream>  using namespace std;  int main()  {  int \*ptr = nullptr;  // Below line compiles  if (ptr) { cout << "true"; }  else { cout << "false"; }  } |

Output:

false

There are some unspecified things when we compare two simple pointers but comparison between two values of type nullptr\_t is specified as, comparison by <= and >= return true and comparison by < and > returns false and comparing any pointer type with nullptr by == and != returns true or false if it is null or non-null respectively.

**MOVE-**

* **Move constructors.**

A move constructor looks like this:

C::C(C&& other); //C++11 move constructor

It doesn't allocate new resources. Instead, it prefers other’s resources and then sets other to its default-constructed state.

Ex-1 MemoryPage(MemoryPage&& other): size(0), buf(nullptr)

{

// pilfer other's resource

size=other.size;

buf=other.buf;

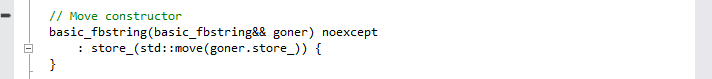
// reset other

other.size=0;

other.buf=nullptr;

}

Ex-2

[](http://www.javadepend.com/Blog/wp-content/uploads/c12.png)

Ex-3

// Move constructor.

MemoryBlock(MemoryBlock&& other)

: \_data(nullptr)

, \_length(0)

{

std::cout << "In MemoryBlock(MemoryBlock&&). length = "

<< other.\_length << ". Moving resource." << std::endl;

// Copy the data pointer and its length from the

// source object.

\_data = other.\_data;

\_length = other.\_length;

// Release the data pointer from the source object so that

// the destructor does not free the memory multiple times.

other.\_data = nullptr;

other.\_length = 0;

}

// Move assignment operator.

MemoryBlock& operator=(MemoryBlock&& other)

{

std::cout << "In operator=(MemoryBlock&&). length = "

<< other.\_length << "." << std::endl;

if (this != &other)

{

// Free the existing resource.

delete[] \_data;

// Copy the data pointer and its length from the

// source object.

\_data = other.\_data;

\_length = other.\_length;

// Release the data pointer from the source object so that

// the destructor does not free the memory multiple times.

other.\_data = nullptr;

other.\_length = 0;

}

return \*this;

}

The move constructor is much faster than a copy constructor because it doesn't allocate memory nor does it copy memory buffers.

* **Move assignment operators.**

A move assignment operator has the following signature: C& C::operator=(C&& other); .A move assignment operator is similar to a copy constructor except that before pilfering the source object, it releases any resources that its object may own. The move assignment operator performs four logical steps:

Release any resources that \*this currently owns. Pilfer other's resource.Set other to a default state.

Return \*this.

Here's a definition of MemoryPage's move assignment operator:

//C++11

Ex-1 MemoryPage& MemoryPage::operator=(MemoryPage&& other)

{

if (this!=&other)

{

// release the current object's resources

delete[] buf;

size=0;

// pilfer other's resource

size=other.size;

buf=other.buf;

// reset other

other.size=0;

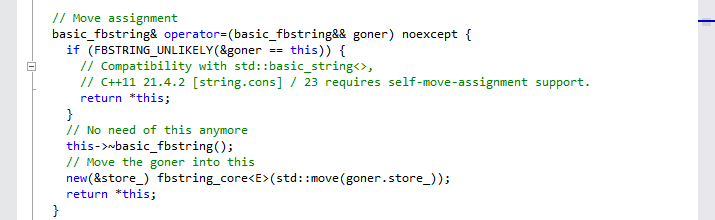
other.buf=nullptr;

}

return \*this;

}

Ex-2

[](http://www.javadepend.com/Blog/wp-content/uploads/c10.png)

**Socket Programming in C/C++**

**What is socket programming?**

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket (node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server

**Situation Socket is required**

Suppose that you are a socket programmer who works for an application development company that specializes in socket applications for the i5/OS™. Operating system. To keep ahead of its competitors, your company has decided to develop a suite of applications that use the AF\_INET6 address family, which accept connections from IPv4 and IPv6. You want to create an application that processes requests from both IPv4 and IPv6 nodes. You know that the i5/OS operating system supports the AF\_INET6 address family sockets, which provides interoperability with AF\_INET address family sockets. You also know that you can accomplish this by using an IPv4-mapped IPv6 address format.

**What are the advantages of socket programming?**

ADVANTAGE: Socket programming usually pertains to the basic communication protocols like TCP/UDP and raw sockets like ICMP. These protocols have a small communication overhead when compared to underlying protocols such as HTTP/DHCP/SMTP etc.

**Scenario objectives**

This scenario has the following objectives and goals:

1-Create a server application that accepts and processes requests from IPv6 and IPv4 clients

2-Create a client application that requests data from an IPv4 or IPv6 server application

**Prerequisite steps**

Before developing your application that meets these objectives, complete the following tasks:

1. Install QSYSINC library. This library provides necessary header files that are needed when compiling socket applications.
2. Install the ILE C licensed program (5761-WDS option 51).
3. Install and configure an Ethernet card. For information about Ethernet options, see the Ethernet topic in the information center.
4. Set up TCP/IP and IPv6 network. Refer to the information about configuring TCP/IP and configuring IPv6.

## **Scenario details**

For the IPv6 network, for which you create applications to handle, requests from IPv6 and IPv4 clients. The i5/OS operating system contains the program that listens and processes requests from these clients. The network consists of two separate domains, one that contains IPv4 clients exclusively and the other remote network that contains only IPv6 clients. The domain name of the system is myserver.myco.com. The server application uses the AF\_INET6 address family to process these incoming requests with the in6addr\_any specified on the bind() API call.

**Stages for server**.

* **Socket creation:**

int sockfd = socket(domain, type, protocol)

**sockfd:** socket descriptor, an integer (like a file-handle)  
**domain:** integer, communication domain e.g., AF\_INET (IPv4 protocol) , AF\_INET6 (IPv6 protocol)  
**type:** communication type  
SOCK\_STREAM: TCP(reliable, connection oriented)  
SOCK\_DGRAM: UDP(unreliable, connectionless)  
**protocol:** Protocol value for Internet Protocol(IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet.(man protocols for more details)

* **Setsockopt:**
* int setsockopt(int sockfd, int level, int optname,

const void \*optval, socklen\_t optlen);

This helps in manipulating options for the socket referred by the file descriptor sockfd. This is completely optional, but it helps in reuse of address and port. Prevents error such as: “address already in use”.

* **Bind:**
* int bind(int sockfd, const struct sockaddr \*addr,

socklen\_t addrlen);

After creation of the socket, bind function binds the socket to the address and port number specified in addr(custom data structure). In the example code, we bind the server to the localhost, hence we use INADDR\_ANY to specify the IP address.

* **Listen:**

int listen(int sockfd, int backlog);

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog, defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

* **Accept:**

int new\_socket= accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, connection is established between client and server, and they are ready to transfer data.

**Stages for Client**

* **Socket connection:** Exactly same as that of server’s socket creation
* **Connect:**
* int connect(int sockfd, const struct sockaddr \*addr,

socklen\_t addrlen);

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server’s address and port is specified in addr.

**Server.c**

// Server side C/C++ program to demonstrate Socket programming

#include <unistd.h>

#include <stdio.h>

#include <sys/socket.h>

#include <stdlib.h>

#include <netinet/in.h>

#include <string.h>

#define PORT 8080

int main(int argc, char const \*argv[])

{

int server\_fd, new\_socket, valread;

struct sockaddr\_in address;

int opt = 1;

int addrlen = sizeof(address);

char buffer[1024] = {0};

char \*hello = "Hello from server";

// Creating socket file descriptor

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0)

{

perror("socket failed");

exit(EXIT\_FAILURE);

}

// Forcefully attaching socket to the port 8080

if (setsockopt(server\_fd, SOL\_SOCKET, SO\_REUSEADDR | SO\_REUSEPORT,

&opt, sizeof(opt)))

{

perror("setsockopt");

exit(EXIT\_FAILURE);

}

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons( PORT );

// Forcefully attaching socket to the port 8080

if (bind(server\_fd, (struct sockaddr \*)&address,

sizeof(address))<0)

{

perror("bind failed");

exit(EXIT\_FAILURE);

}

if (listen(server\_fd, 3) < 0)

{

perror("listen");

exit(EXIT\_FAILURE);

}

if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address,

(socklen\_t\*)&addrlen))<0)

{

perror("accept");

exit(EXIT\_FAILURE);

}

valread = read( new\_socket , buffer, 1024);

printf("%s\n",buffer );

send(new\_socket , hello , strlen(hello) , 0 );

printf("Hello message sent\n");

return 0;

}

**Client.C**

// Client side C/C++ program to demonstrate Socket programming

#include <stdio.h>

#include <sys/socket.h>

#include <stdlib.h>

#include <netinet/in.h>

#include <string.h>

#define PORT 8080

int main(int argc, char const \*argv[])

{

struct sockaddr\_in address;

int sock = 0, valread;

struct sockaddr\_in serv\_addr;

char \*hello = "Hello from client";

char buffer[1024] = {0};

if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

{

printf("\n Socket creation error \n");

return -1;

}

memset(&serv\_addr, '0', sizeof(serv\_addr));

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(PORT);

// Convert IPv4 and IPv6 addresses from text to binary form

if(inet\_pton(AF\_INET, "127.0.0.1", &serv\_addr.sin\_addr)<=0)

{

printf("\nInvalid address/ Address not supported \n");

return -1;

}

if (connect(sock, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) < 0)

{

printf("\nConnection Failed \n");

return -1;

}

send(sock , hello , strlen(hello) , 0 );

printf("Hello message sent\n");

valread = read( sock , buffer, 1024);

printf("%s\n",buffer );

return 0;

}

**Output:**

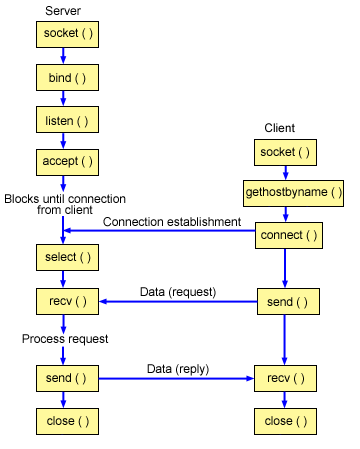
Client:Hello message sent

Hello from server

Server:Hello from client

Hello message sent

**State diagram for server and client model**



### Elementary socket system calls

### socket

To do network I/O, the first thing a process must do is to call the **socket** system call, specifying the type of communication protocol desired.

#include <sys/types.h>

#include <sys/socket.h>

int socket(int *family*, int *type*, int *protocol*);

The *family* is one of

AF\_UNIX -- Unix internal protocols

AF\_INET -- Internet protocols

AF\_NS -- Xerox NS Protocols

AF\_IMPLINK -- IMP link layer

The AF\_ prefix stands for "address family." In the first project, we are going to use AF\_INET.  
  
The socket *type* is one of the following:

SOCK\_STREAM stream socket

SOCK\_DGRAM datagram socket

SOCK\_RAW raw socket

SOCK\_SEQPACKET sequenced packet socket

SOCK\_RDM reliably delivered message socket (not implemented yet)

The *protocol* argument to the socket system call is typically set to 0 for most user applications. The valid combinations are shown as follows.

|  |  |  |  |
| --- | --- | --- | --- |
| *family* | *type* | *protocol* | *Actual protocol* |
| AF\_INET | SOCK\_DGRAM | IPPROTO\_UDP | UDP |
| AF\_INET | SOCK\_STREAM | IPPROTO\_TCP | TCP |
| AF\_INET | SOCK\_RAW | IPPROTO\_ICMP | ICMP |
| AF\_INET | SOCK\_RAW | IPPROTO\_RAW | (raw) |

### bind

The **bind** system call assigns a name to an unnamed socket.

#include <sys/types.h>

#include <sys/socket.h>

int bind(int *sockfd*, struct sockaddr *\*myaddr*, int *addrlen*);

The first argument is the socket descriptor returned from **socket** system call. The second argument is a pointer to a protocol-specific address and the third argument is the size of this address. There are three uses of **bind**.

1. Servers register their well-known address with the system. It tells the system "this is my address and any messages received for this address are to be given to me." Both connection-oriented and connectionless servers need to do this before accepting client requests.
2. A client can register a specific address for itself.
3. A connectionless client needs to assure that the system assigns it some unique address, so that the other end (the server) has a valid return address to send its responses to. This corresponds to making certain an envelope has a valid return address, if we expect to get a reply from the person we sent the letter to.

### connect

A client process **connect**s a socket descriptor following the **socket** system call to establish a connection with a server.

#include <sys/types.h>

#include <sys/socket.h>

int connect(int *sockfd*, struct sockaddr *\*servaddr*, int *addrlen*);

The *sockfd* is a socket descriptor that was returned by the **socket** system call. The second and third arguments are a pointer to a socket address, and its size, as described earlier.   
For most connection-oriented protocols (TCP, for example), the **connect** system call results in the actual establishment of a connection between the local system and the foreign system.   
The **connect** system call does not return until the connection is established, or an error is returned to the process.   
The client does not have to **bind** a local address before calling **connect**. The connection typically causes these four elements of the association 5-tuple to be assigned: *local-addr*, *local-process*,*foreign-addr*, and *foreign-process*. In all the connection-oriented clients, we will let **connect** assign the local address.

### listen

This system call is used by a connection-oriented server to indicate that it is willing to receive connections.

#include <sys/types.h>

#include <sys/socket.h>

int listen(int *sockfd*, int *backlog*);

It is usually executed after both the **socket** and **bind** system calls, and immediately before the **accept** system call. The *backlog* argument specifies how many connection requests can be queued by the system while it waits for the server to execute the **accept** system call. This argument is usually specified as 5, the maximum value currently allowed.

### accept

After a connection-oriented server executes the **listen** system call described above, an actual connection from some client process is waited for by having the server execute the **accept** system call.

#include <sys/types.h>

#include <sys/socket.h>

int accept(int *sockfd*, struct sockaddr *\*peer*, int *\*addrlen*);

**accept** takes the first connection request on the queue and creates another socket with the same properties as *sockfd*. If there are no connection requests pending, this call blocks the caller until one arrives.   
The *peer* and *addrlen* arguments are used to return the address of the connected peer process (the client). *addrlen* is called a value-result argument: the caller sets its value before the system call, and the system call stores a result in the variable. For this system call the caller sets *addrlen* to the size of the **sockaddr** structure whose address is passed as the *peer* argument.

### send, sendto, recv and recvfrom

These system calls are similar to the standard **read** and **write** system calls, but additional arguments are required.

#include <sys/types.h>

#include <sys/socket.h>

int send(int *sockfd*, char *\*buff*, int *nbytes*, int *flags*);

int sendto(int *sockfd*, char *\*buff*, int *nbytes*, int *flags*, struct sockaddr *\*to*, int *addrlen*);

int recv(int *sockfd*, char *\*buff*, int *nbytes*, int *flags*);

int recvfrom(int *sockfd*, char *\*buff*, int *nbytes*, int *flags*, struct sockaddr *\*from*, int *\*addrlen*);

The first three arguments, *sockfd*, *buff*, and *nbytes*, to the four system calls are similar to the first three arguments for **read** and **write**. The *flags* argument can be safely set to zero ignoring the details for it. The *to* argument for **sendto** specifies the protocol-specific address of where the data is to be sent. Since this address is protocol-specific, its length must be specified by *addrlen*. The**recvfrom** system call fills in the protocol-specific address of who sent the data into *from*. The length of this address is also returned to the caller in *addrlen*. Note that the final argument to **sendto**is an integer value, while the final argument to **recvfrom** is a pointer to an integer value.

### close

The normal Unix **close** system call is also used to close a socket.

int close(int *fd*);

If the socket being closed is associated with a protocol that promises reliable delivery (e.g., TCP or SPP), the system must assure that any data within the kernel that still has to be transmitted or acknowledged, is sent. Normally, the system returns from the **close** immediately, but the kernel still tries to send any data already queued.