Debugging using GDB: Find a segFault

- testgdb.cpp

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

int main()
{
    int x = 7;
    int *p = 0;
    cout << "x = " << x;
    cout << "The pointer points to the value " << *p;
}
```
GDB: The GNU Project Debugger

https://www.gnu.org/software/gdb/

GDB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.
How to use GDB?

- Rebuild it with debugging symbols and then run it under the debugger.

$ [ahnt@hopper]$ g++ -g testgdb.cpp -o testgdb
How to use GDB?

- `gdb ./a.out` - Start gdb attached to the program I'm interested in. This does not start the program running.
- `run` - Start program running
- `print p` - Show me the value of the object named p.
- `list` - Show the line that execution has paused at, and surrounding lines of code.
- `quit` - Exit gdb.
Doubly Linked List

- There is a type of linked list that allows us to go in both directions—forward and reverse—in a linked list. It is the **doubly linked list**.

- In addition to its element member, a node in a doubly linked list stores two pointers, a `next` link and a `prev` link, which point to the next node in the list and the previous node in the list, respectively.

- Such lists allow for a great variety of quick update operations, including efficient insertion and removal at any given position.
Header and Trailer Sentinels

- To simplify programming, it is convenient to add special nodes at both ends of a doubly linked list: a header node just before the head of the list, and a trailer node just after the tail of the list.

- These “dummy” or sentinel nodes do not store any elements. They provide quick access to the first and last nodes of the list.

- In particular, the header’s next pointer points to the first node of the list, and the prev pointer of the trailer node points to the last node of the list. An example is shown in Figure 3.12.

\[\text{Figure 3.12: A doubly linked list with sentinels, header and trailer, marking the ends of the list. An empty list would have these sentinels pointing to each other. We do not show the null prev pointer for the header nor do we show the null next pointer for the trailer.}\]
Insertion into a Doubly Linked List

- Given a node \( v \) of a doubly linked list (which could possibly be the header, but not the trailer), let \( z \) be a new node that we wish to insert immediately after \( v \).

- Let \( w \) be the node following \( v \), that is, \( w \) is the node pointed to by \( v \)'s next link. (This node exists, since we have sentinels.)

- To insert \( z \) after \( v \), we link it into the current list, by performing the following operations:
  - Make \( z \)'s prev link point to \( v \)
  - Make \( z \)'s next link point to \( w \)
  - Make \( w \)'s prev link point to \( z \)
  - Make \( v \)'s next link point to \( z \)
Figure 3.13: Adding a new node after the node storing JFK: (a) creating a new node with element BWI and linking it in; (b) after the insertion.
Removal from a Doubly Linked List

Figure 3.14: Removing the node storing PVD: (a) before the removal; (b) linking out the old node; (c) after node deletion.
```cpp
#ifndef DLINKEDLIST_H
#define DLINKEDLIST_H

#include <iostream>
#include <stdexcept>

using namespace std;

template <typename Object>
class DLinkedList {
public:
    DLinkedList();  // empty list constructor
    ~DLinkedList(); // destructor
    bool empty() const;  // is list empty?
    const Object& front() const;  // get front element
    const Object& back() const;  // get back element
    void addFront(const Object& e);  // add to front of list
    void addBack(const Object& e);  // add to back of list
    void removeFront();  // remove from front
    void removeBack();  // remove from back

private:
    struct DNode {  // node struct
        Object elem;
        DNode* prev;
        DNode* next;
    };

    DNode* header;  // list sentinels
    DNode* trailer;  // list sentinels

protected:
    void add(DNode* v, const Object& e);
    void remove(DNode* v);

};

#include "DLinkedList.cpp"

#endif
```
DLinkedList.cpp

DLinkedList::DLinkedList() { // constructor
    header = new DNode; // create sentinels
    trailer = new DNode;
    header->next = trailer; // have them point to each other
    trailer->prev = header;
}

DLinkedList::~DLinkedList() { // destructor
    while (!empty()) removeFront(); // remove all but sentinels
    delete header;
    delete trailer;
}

Code Fragment 3.24: Class constructor and destructor.

bool DLinkedList::empty() const // is list empty?
{ return (header->next == trailer); }

const Elem& DLinkedList::front() const // get front element
{ return header->next->elem; }

const Elem& DLinkedList::back() const // get back element
{ return trailer->prev->elem; }

Code Fragment 3.25: Accessor functions for the doubly linked list class.
DLinkedList.cpp

```cpp
void DLinkedList::add(DNode* v, const Elem& e) {
    DNode* u = new DNode; u->elem = e; // create a new node for e
    u->next = v; // link u in between v
    u->prev = v->prev;
    v->prev->next = v->prev = u;
}

void DLinkedList::addFront(const Elem& e) // add to front of list
{ add(header->next, e); }

void DLinkedList::addBack(const Elem& e) // add to back of list
{ add(trailer, e); }
```

**Code Fragment 3.26:** Inserting a new node into a doubly linked list. The protected utility function `add` inserts a node `z` before an arbitrary node `v`. The public member functions `addFront` and `addBack` both invoke this utility function.
void DLinkedList::remove(DNode* v) {
    DNode* u = v->prev;
    DNode* w = v->next;
    u->next = w;
    w->prev = u;
    delete v;
}

void DLinkedList::removeFront() {
    remove(header->next);
}

void DLinkedList::removeBack() {
    remove(trailer->prev);
}

Code Fragment 3.27: Removing a node from a doubly linked list. The local utility function remove removes the node v. The public member functions removeFront and removeBack invoke this utility function.
STL – Standard Template Library

- Collections of useful classes for common data structures
- Ability to store objects of any type (template)
- Study of containers
- Containers form the basis for treatment of data structures
- Container – class that stores a collection of data
- STL consists of 10 container classes:
  - Sequence containers
  - Adapter containers
  - Associative containers
STL Containers

● **Sequence Container**
  - Stores data by position in linear order:
  - First element, second element, etc:

● **Associate Container**
  - Stores elements by key, such as name, social security number or part number
  - Access an element by its key which may bear no relationship to the location of the element in the container

● **Adapter Container**
  - Contains another container as its underlying storage structure
STL Containers

- **Sequence Container**
  - Vector
  - Deque
  - List

- **Adapter Containers**
  - Stack
  - Queue
  - Priority queue

- **Associative Container**
  - Set, multiset
  - Map, multimap
STL Vectors

- Generalized array that stores a collection of elements of the same data type

- Vector – similar to an array
  - A vector holds a sequence of values, or elements.
  - A vector stores its elements in contiguous memory locations.
  - You can use the array subscript operator [ ] to read the individual elements in the vector
STL Vectors

- **Vector vs Array**
  - Unlike C++ arrays, STL vectors can be dynamically resized, and new elements may be efficiently appended or removed from the end of an array. If you add a value to a vector that is already full, the vector will automatically increase its size to accommodate the new value.
  - As with arrays, individual elements of a vector object can be indexed using the usual index operator (“[ ]”). Elements can also be accessed by a member function called **at**. The advantage of this member function over the index operator is that it performs range checking and generates an error exception if the index is out of bounds.
  - You do not have to declare the number of elements that the vector will have.
Declaring a vector

Example:

```cpp
#include <vector>

vector<int> scores (100); // 100 integer scores
vector<string> students; // list of students
```
## Other examples of vector Declarations

<table>
<thead>
<tr>
<th>Declaration Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vector&lt;float&gt; amounts;</code></td>
<td>Declares <em>amounts</em> as an empty vector of floats.</td>
</tr>
<tr>
<td><code>vector&lt;int&gt; scores(15);</code></td>
<td>Declares <em>scores</em> as a vector of 15 ints.</td>
</tr>
<tr>
<td><code>vector&lt;char&gt; letters(25, 'A');</code></td>
<td>Declares <em>letters</em> as a vector of 25 characters. Each element is initialized with 'A'.</td>
</tr>
<tr>
<td><code>vector&lt;double&gt; values2(values1);</code></td>
<td>Declares <em>values2</em> as a vector of doubles. All the elements of <em>values1</em>, which also a vector of doubles, are copied to <em>value2</em>.</td>
</tr>
</tbody>
</table>
Principal Member Functions of Vector

`vector(n):` Construct a vector with space for `n` elements; if no argument is given, create an empty vector.

`size():` Return the number of elements in `V`.

`empty():` Return true if `V` is empty and false otherwise.

`resize(n):` Resize `V`, so that it has space for `n` elements.

`reserve(n):` Request that the allocated storage space be large enough to hold `n` elements.

`operator[i]:` Return a reference to the `i`th element of `V`.

`at(i):` Same as `V[i]`, but throw an `out_of_range` exception if `i` is out of bounds, that is, if `i < 0` or `i ≥ V.size()`.

`front():` Return a reference to the first element of `V`.

`back():` Return a reference to the last element of `V`.

`push_back(e):` Append a copy of the element `e` to the end of `V`, thus increasing its size by one.

`pop_back():` Remove the last element of `V`, thus reducing its size by one.
Vector Container

- Allows direct access to the elements via an index operator [ ].
- Indices for the vector elements are in the range from 0 to size() - 1.

Example:

```cpp
#include <vector>

vector <int> v(20);

v[5]=15;
```
```cpp
#include <iostream>
#include <vector>
using namespace std;

int main(){
    vector<int> hours(5);
    vector<float> rates(5);

    cout << "Enter the hours worked by 5 employees and their hourly rates" << endl;
    for (int index = 0; index < hours.size(); index++) {
        cout << "Hours worked by employee #" << (index + 1) << ". ";
        cin >> hours[index];
        cout << "Hourly pay rate for employee #" << (index + 1) << ". ";
        cin >> rates[index];
    }

    int grossPay = 0;
    cout << "Here is the gross pay for each employee:" << endl;
    for (int index = 0; index < hours.size(); index++) {
        grossPay = hours[index] * rates[index];
        cout << "Gross pay for employee #" << (index + 1) << ": " << grossPay << endl;
    }

    return 0;
}
```
```cpp
#include <iostream>
#include <vector>
using namespace std;

int main(){
    int numEmployees;
    vector<int> hours;
    vector<float> rates;
    cout << "How many employees do you have? " << endl;
    cin >> numEmployees;

    cout << "Enter the hours worked by " << numEmployees << endl;
    cout << "employees and their hourly rates" << endl;

    int workHour, payRate;
    for (int index = 0; index < numEmployees; index++) {
        cout << "Hours worked by employee #" << (index + 1) << ": ";
        cin >> workHour;
        hours.push_back(workHour);
        cout << "Hourly pay rate for employee #" << (index + 1) << ": ";
        cin >> payRate;
        rates.push_back(payRate);
    }

    int grossPay = 0;
    cout << "Here is the gross pay for each employee:" << endl;
    for (int index = 0; index < numEmployees; index++) {
        grossPay = hours.at(index) * rates.at(index);
        cout << "Gross pay for employee #" << (index + 1) << ": " << grossPay << endl;
    }

    return 0;
}
```
STL Iterators

- You can think of an iterator as pointing to an item that is part of a larger container of items.
- For instance, all containers support a function called `begin`, which will return an iterator pointing to the beginning of the container (the first element) and function, `end`, that returns an iterator corresponding to having reached the end of the container.
- In fact, you can access the element by "dereferencing" the iterator with a `*`, just as you would dereference a pointer.
STL Iterators

- To request an iterator appropriate for a particular STL templated class, you use the syntax

\[
\text{std::class\_name<template\_parameters>::iterator name}
\]

- \text{name} is the name of the iterator variable you wish to create and the \text{class\_name} is the name of the STL container you are using, and the \text{template\_parameters} are the parameters to the template used to declare objects that will work with this iterator.

- Note that because the STL classes are part of the std namespace, you will need to either prefix every container class type with "std::", as in the example, or include "using namespace std;" at the top of your program.

- For instance, if you had an STL vector storing integers, you could create an iterator for it as follows:

\[
\text{std::vector<int> myIntVector;}
\text{std::vector<int>::iterator myIntVectorIterator;}
\]
```cpp
#include <iostream>
#include <vector>
using namespace std;

int main ()
{
    vector<int> myIntVector;
    vector<int>::iterator it;

    for (int i=1; i<=5; i++) myIntVector.push_back(i);

    std::cout << "myvector contains: ";
    for (it = myIntVector.begin(); it != myIntVector.end(); it++) {
        std::cout << *it << " ";
    }
    std::cout << endl;

    return 0;
}
```
STLVectorTest4.cpp

- Change the STLVectorTest2.cpp using iterator.
Implement Array Based Vector

- insert(index, element)
- erase(index)

**Algorithm insert(i, e):**

```plaintext
for j = n - 1, n - 2, ..., i do
A[i] ← e
n ← n + 1
```

**Algorithm erase(i):**

```plaintext
for j = i + 1, i + 2, ..., n - 1 do
n ← n - 1
```

**Code Fragment 6.1:** Methods insert(i, e) and erase(i) in the array implementation of the vector ADT. The member variable n stores the number of elements.

![Diagram of array-based vector implementation with indices shifting up for insertion and down for removal.](image)
An Extendable Array Implementation

Let us provide a means to grow the array $A$ that stores the elements of a vector $V$. Of course, in C++ we cannot actually grow the array $A$; its capacity is fixed at some number $N$. Instead, when an overflow occurs, that is, when $n=N$ and function insert is called, we perform the following steps:

1. Allocate a new array $B$ of capacity $N$
2. Copy $A[i]$ to $B[i]$, for $i = 0, \ldots, N - 1$
3. Deallocate $A$ and reassign $A$ to point to the new array $B$

---

Figure 6.2: The three steps for “growing” an extendable array: (a) create new array $B$; (b) copy elements from $A$ to $B$; (c) reassign $A$ to refer to the new array and delete the old array.
An Extendable Array Implementation

```cpp
void ArrayVector::reserve(int N) {
    if (capacity >= N) return; // reserve at least N spots
    Elem* B = new Elem[N];     // already big enough
    for (int j = 0; j < n; j++) // allocate bigger array
        B[j] = A[j];            // copy contents to new array
    if (A != NULL) delete[] A; // discard old array
    A = B;                      // make B the new array
    capacity = N;               // set new capacity
}

void ArrayVector::insert(int i, const Elem& e) {
    if (n >= capacity)         // overflow?
        reserve(max(1, 2 * capacity)); // double array size
    for (int j = n - 1; j >= i; j--) // shift elements up
    A[i] = e;                   // put in empty slot
    n++;                        // one more element
}

Code Fragment 6.5: The member functions reserve and insert for class ArrayVector.
```
Implement ArrayVector class

- Code is available at website.
- Code implementation of functions are correct, but will give you an error. What’s the error? How to fix it?
- Check the implementation of housekeeping functions and use them from next code.
Test ArrayVector class (testAV.cpp)

```cpp
#include "ArrayVector.h"
#include <iostream>
using namespace std;

int main() {
    ArrayVector<int> vec;
    vec.insert(0, 7); // (1)
    vec.insert(1, 4); // (1, 4)
    cout << vec.at(1) << endl; // (1, 4), output: 7
    vec.push_back(2); // (1, 4, 2)
    vec.push_back(10); // (1, 4, 2, 10)
    cout << vec[3] << endl; // (1, 4, 2, 10), output: 10
    vec.erase(3); // (1, 4, 2)
    cout << vec.back() << endl; // (1, 4, 2), output: 2
    vec.pop_back(); // (1, 4)
    cout << vec.back() << endl; // (1, 4), output: 4

    ArrayVector<int> copyVec = vec;
    cout << copyVec.back() << endl; // (1, 4), output: 4

    return 0;
}
```