Chapter 1

PROGRAMMING PRINCIPLES

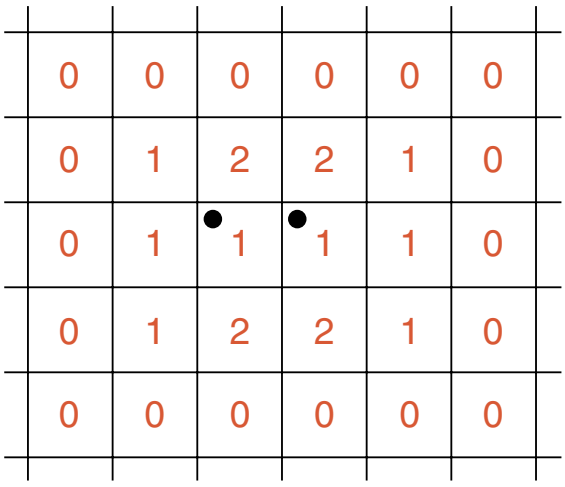
1. Introduction: Problems with large programs
2. The Game of Life (a continuing example)
3. Programming style
4. Coding, testing, and further refinement
5. Program Maintenance
6. Preview
   (a) Software Engineering
   (b) Problem Analysis
   (c) Requirements Specification
   (d) Coding
7. Pointers and Pitfalls
8. References

Problems of Large Programs

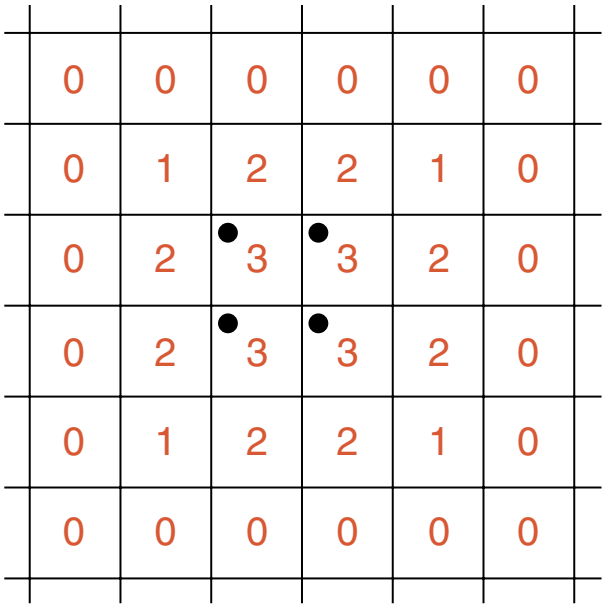
1. The patchwork approach
2. Problem specification
3. Program organization
4. Data organization and data structures
5. Algorithm selection and analysis
6. Debugging
7. Testing and verification
8. Maintenance
9. Highlights of C++
   (a) Data abstraction
   (b) Object-oriented design
   (c) Reusable code
   (d) Refinement, improvement, extension of C

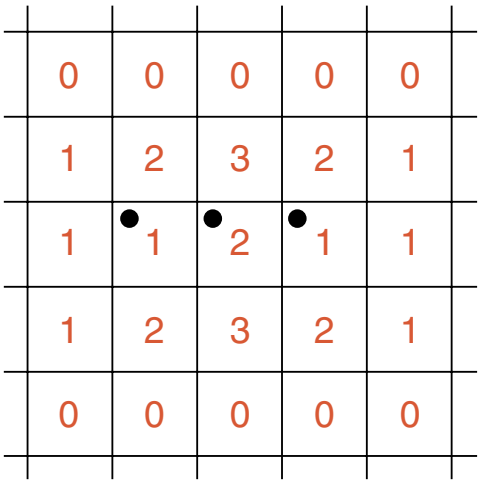
Rules for the Game of Life

1. The neighbors of a given cell are the eight cells that touch it vertically, horizontally, or diagonally. Every cell is either living or dead.
2. A living cell stays alive in the next generation if it has either 2 or 3 living neighbors; it dies if it has 0, 1, 4, or more living neighbors.
3. A dead cell becomes alive in the next generation if it has exactly three neighboring cells, no more or fewer, that are already alive. All other dead cells remain dead in the next generation.
4. All births and deaths take place at exactly the same time, so that a dying cell can help to give birth to another, but cannot prevent the death of others by reducing overcrowding, nor can cells being born either preserve or kill cells living in the previous generation.

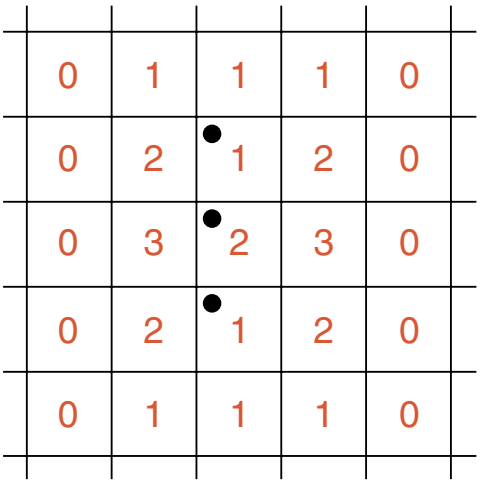


Life: Examples and Outline





and



Set up a Life configuration as an initial arrangement of living and dead cells.

Print the Life configuration.

While the user wants to see further generations:

Update the configuration by applying the rules of the Life game.
Print the current configuration.

C++ Classes, Objects, and Methods

- A ***class*** collects data and the methods used to access or change the data.
- Such a collection of data and methods is called an ***object*** belonging to the given class.
- Every C++ class consists of ***members*** that represent either variables (called ***data members***) or functions (called ***methods*** or ***member functions***). The member functions of a class are normally used to access or alter the data members.

Example: For the Life game, the class is called Life, and configuration becomes a Life *object*. We use three methods: initialize( ) will set up the initial configuration of living and dead cells; print( ) will print out the current configuration; and update( ) will make all the changes that occur in moving from one generation to the next.

- ***Clients***, that is, user programs with access to a particular class, can declare and manipulate objects of that class.
- The ***specifications*** of a method, function, or program are statements of precisely what is done. ***Preconditions*** state what is required before; ***Postconditions*** state what has happened when the method, function, or program has finished.

Programming Precept

Include precise specifications
with every program, function, and method that you write.

Information Hiding

- By relying on its specifications, a client can use a method without needing to know how the data are actually stored or how the methods are actually programmed. This important programming strategy known as ***information hiding***.

- Data members and methods available to a client are called ***public***; further ***private*** variables and functions may be used in the implementation of the class, but are not available to a client.

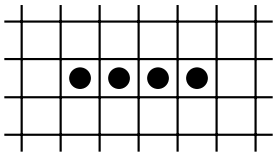
Convention

Methods of a class are public.
Functions in a class are private.

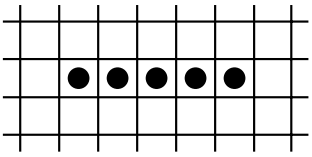
Life: The Main Program

```
#include "utility.h"
#include "life.h"


int main( )                          //    Program to play Conway's game of Life.
/* Pre:   The user supplies an initial configuration of living cells.
   Post:  The program prints a sequence of pictures showing the changes in the config-
          uration of living cells according to the rules for the game of Life.
   Uses:  The class Life and its methods initialize( ), print( ), and update( ).
          The functions  instructions( ), user_says_yes( ).  */

{
   Life configuration;
   instructions( );
   configuration.initialize( );
   configuration.print( );
   cout << "Continue viewing new generations? " << endl;
   while (user_says_yes( )) {
      configuration.update( );
      configuration.print( );
      cout << "Continue viewing new generations? " << endl;
   }
}
```

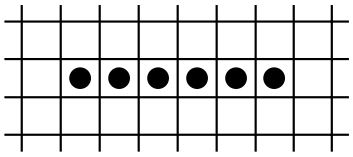
Examples of Life Configurations



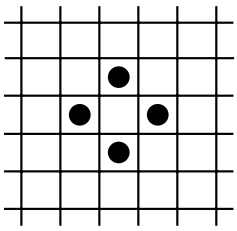
(a)



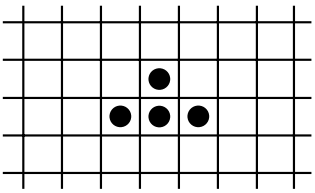
(b)



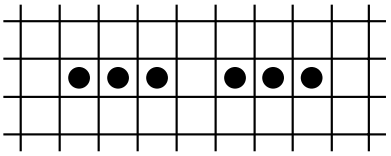
(c)



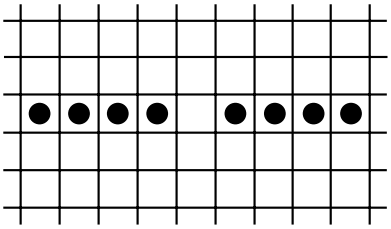
(d)



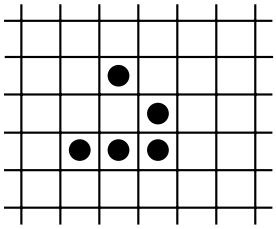
(e)



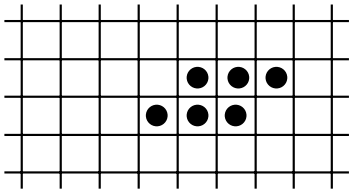
(f)



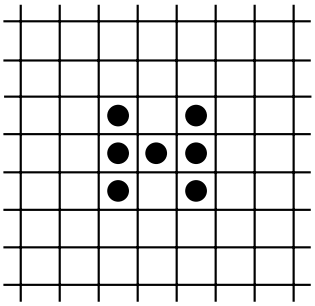
(g)



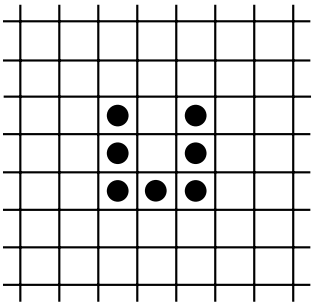
(h)



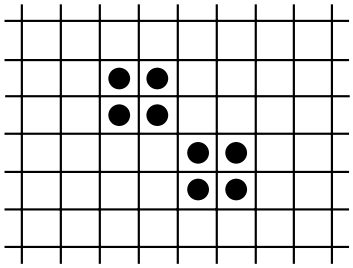
(i)



(j)



(k)



(l)

Guidelines for Choosing Names

Programming Precept

Always name your classes, variables, and functions with the greatest care, and explain them thoroughly.

1. Give special care to the choice of names for classes, functions, constants, and all global variables used in different parts of the program.

2. Keep the names simple for variables used only briefly and locally.

3. Use common prefixes or suffixes to associate names of the same general category.

4. Avoid deliberate misspellings and meaningless suffixes to obtain different names.

5. Avoid choosing cute names whose meaning has little or nothing to do with the problem.

6. Avoid choosing names that are close to each other in spelling or otherwise easy to confuse.

7. Be careful in the use of the letter "l" (small ell), "O" (capital oh) and "0" (zero).

Documentation Guidelines

1. Place a prologue at the beginning of each function with
   (a) Identification (programmer's name, date, version).
   (b) Statement of the purpose of the function and method used.
   (c) Changes the function makes and what data it uses.
   (d) Reference to further documentation for the program.
2. When each variable, constant, or type is declared, explain what it is and how it is used.
3. Introduce each significant part of the program with a statement of purpose.
4. Indicate the end of each significant section.
5. Avoid comments that parrot what the code does or that are meaningless jargon.
6. Explain any statement that employs a trick or whose meaning is unclear. Better still, avoid such statements.
7. The code itself should explain *how* the program works. The documentation should explain *why* it works and *what* it does.
8. Be sure to modify the documentation along with the program.

Programming Precept

Keep your documentation concise but descriptive.

Programming Precept

The reading time for programs is much more than the writing time. Make reading easy to do.

Refinement and Modularity

Top-down design and refinement:

Programming Precept

Don't lose sight of the forest for its trees.

Division of work:

Programming Precept

Use classes to model the fundamental concepts of the program.

Programming Precept

Each function should do only one task, but do it well.

Programming Precept

Each class or function should hide something.

Categories of Data

- ***Input*** parameters
- ***Output*** parameters
- ***Inout*** parameters
- ***Local*** variables
- ***Global*** variables

Programming Precept

Keep your connections simple. Avoid global variables whenever possible.

Programming Precept

Never cause side effects if you can avoid it.
If you must use global variables as input, document them thoroughly.

Programming Precept

Keep your input and output as separate functions,
so they can be changed easily
and can be custom tailored to your computing system.

Life Methods

int Life :: neighbor_count(**int** row, **int** col)

Pre: The Life object contains a configuration, and the coordinates row and col define a cell inside its hedge.

Post: The number of living neighbors of the specified cell is returned.

void Life :: update( )

Pre: The Life object contains a configuration.

Post: The Life object contains the next generation of configuration.

void instructions( )

Pre: None.

Post: Instructions for using the Life program are printed.

void Life :: initialize( )

Pre: None.

Post: The Life object contains a configuration specified by the user.

void Life :: print( )

Pre: The Life object contains a configuration.

Post: The configuration is written for the user.

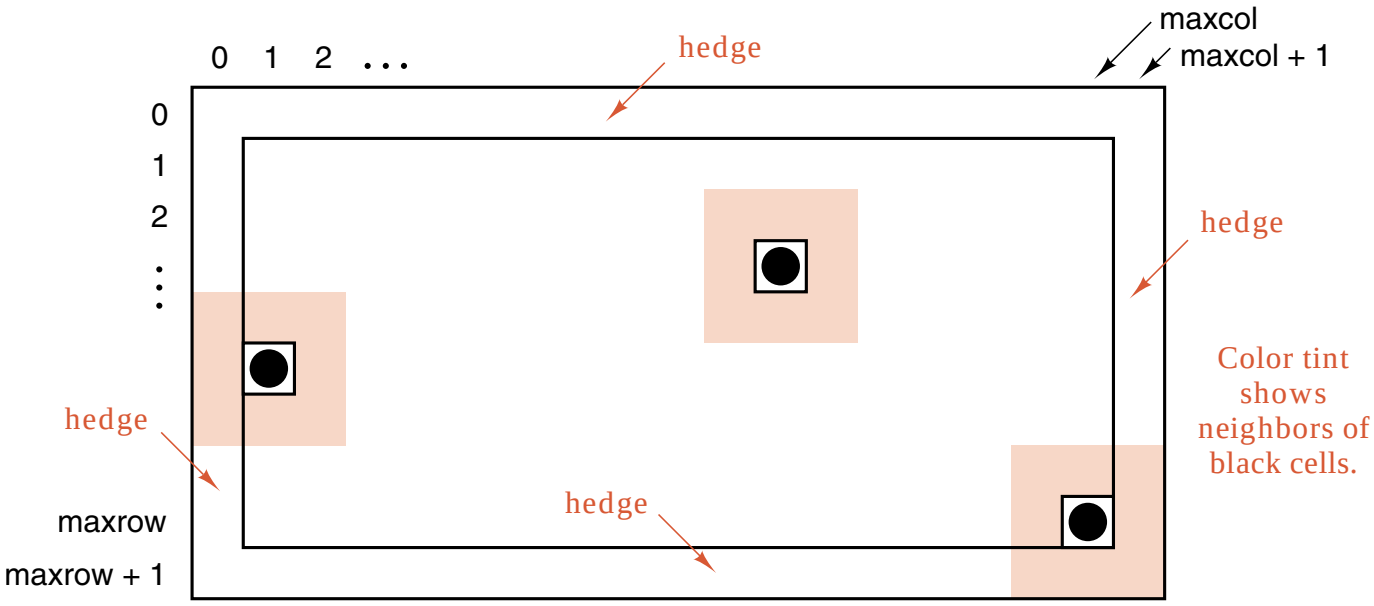
Stubs

```
void instructions( ) { }
bool user_says_yes( ) { return(true); }

class Life {                          //    In file life.h
public:
   void initialize( );
   void print( );
   void update( );
};

void Life :: initialize( ) {}         //    In file life.c
void Life :: print( ) {}
void Life :: update( ) {}
```

Definition of the Class Life

```
const int maxrow = 20, maxcol = 60;   //    grid dimensions

class Life {
public:
   void initialize( );
   void print( );
   void update( );
private:
   int grid[maxrow + 2][maxcol + 2];
                                      //    allows for two extra rows and columns
   int neighbor_count(int row, int col);
};
```

Counting Neighbors




```
int Life :: neighbor_count(int row, int col)
/* Pre:   The Life object contains a configuration, and the coordinates row and col
          define a cell inside its hedge.
   Post:  The number of living neighbors of the specified cell is returned.  */

{
   int i, j;
   int count = 0;
   for (i = row - 1; i <= row + 1; i++)
      for (j = col - 1; j <= col + 1; j++)
         count += grid[i][j];        //  Increase the count if neighbor is alive.
   count -= grid[row][col];
                                     //  Reduce count, since cell is not its own neighbor.
   return count;
}
```

Updating the Grid

```
void Life :: update( )
/* Pre:   The Life object contains a configuration.
   Post:  The Life object contains the next generation of configuration.  */

{
   int row, col;
   int new_grid [maxrow + 2] [maxcol + 2];

   for (row = 1;  row <= maxrow;  row++)
      for (col = 1;  col <= maxcol;  col++)
         switch (neighbor_count(row, col)) {
         case 2:
            new_grid [row] [col] = grid [row] [col];
                                   //    Status stays the same.
            break;
         case 3:
            new_grid [row] [col] = 1;   //    Cell is now alive.
            break;
         default:
            new_grid [row] [col] = 0;   //    Cell is now dead.
         }

   for (row = 1;  row <= maxrow;  row++)
      for (col = 1;  col <= maxcol;  col++)
         grid [row] [col] = new_grid [row] [col];
}
```

Instructions

```
void instructions( )
/* Pre:   None.
   Post:  Instructions for using the Life program have been printed.  */
{
   cout << "Welcome to Conway's game of Life." << endl;
   cout  << "This game uses a grid of size "
         << maxrow << " by " << maxcol << " in which each" << endl;
   cout << "cell can either be occupied by an organism or not." << endl;
   cout << "The occupied cells change from generation to generation"
         << endl;
   cout  << "according to how many neighboring cells are alive."
         << endl;
}
```

Output

```
void Life :: print( )
/* Pre:   The Life object contains a configuration.
   Post:  The configuration is written for the user.  */
{
   int row, col;
   cout << "\nThe current Life configuration is:" << endl;
   for (row = 1;  row <= maxrow;  row++) {
      for (col = 1;  col <= maxcol;  col++)
         if (grid[row] [col] ==  1) cout << '*';
         else cout << ' ';
      cout << endl;
   }
   cout << endl;
}
```

Initialization

```
void Life :: initialize( )
/* Pre:   None.
   Post:  The Life object contains a configuration specified by the user.  */

{
   int row, col;
   for (row = 0;  row <= maxrow + 1;  row++)
      for (col = 0;  col <= maxcol + 1;  col++)
         grid[row] [col] = 0;
   cout << "List the coordinates for living cells." << endl;
   cout << "Terminate the list with the the special pair -1 -1" << endl;
   cin >> row >> col;

   while (row != -1 || col != -1) {
      if (row >= 1 && row <= maxrow)
         if (col >= 1 && col <= maxcol)
            grid[row] [col] = 1;
         else
            cout << "Column " << col << " is out of range." << endl;
      else
         cout << "Row " << row << " is out of range." << endl;
      cin >> row >> col;
   }
}
```

Utility Package

We shall assemble a ***utility package*** that contains various declarations and functions that prove useful in a variety of programs, even though these are not related to each other as we might expect if we put them in a class.

For example, we shall put declarations to help with error processing in the utility package.

Our first function for the utility package obtains a yes-no response from the user:

bool user_says_yes( )

Pre: None.

Post: Returns **true** if the user enters 'y' or 'Y'; returns **false** if the user enters 'n' or 'N'; otherwise requests new response

```
bool user_says_yes( )
{
   int c;
   bool initial_response = true;
   do {                                   //  Loop until an appropriate input is received.
      if (initial_response)
         cout << " (y,n)? " << flush;
      else
         cout << "Respond with either y or n: " << flush;
      do {                                //  Ignore white space.
         c = cin.get( );
      } while (c == '\n' || c == ' ' || c == '\t');
      initial_response = false;
   } while (c != 'y' && c != 'Y' && c != 'n' && c != 'N');
   return (c == 'y' || c == 'Y');
}
```

Drivers

A ***driver*** for a function is an auxiliary program whose purpose is to provide the necessary input for the function, call it, and evaluate the result.

neighbor_count():

```
int main ( )                          //    driver for neighbor_count( )
/* Pre:   None.
   Post:  Verifies that the method neighbor_count( ) returns the correct values.
   Uses:  The class Life and its method initialize( ).   */
{
   Life configuration;
   configuration.initialize( );
   for (row = 1;  row <= maxrow;  row++){
      for (col = 1;  col <= maxrow;  col++)
          cout << configuration.neighbor_count(row, col) << " ";
      cout << endl;
   }
}
```

initialize() and print():

```
configuration.initialize( );
configuration.print( );
```

Both methods can be tested by running this driver and making sure that the configuration printed is the same as that given as input.

Debugging and Testing

Methods for debugging:

1. Structured walkthrough
2. Trace tools and snapshots
3. Scaffolding
4. Static analyzer

Programming Precept

The quality of test data is more important than its quantity.

Programming Precept

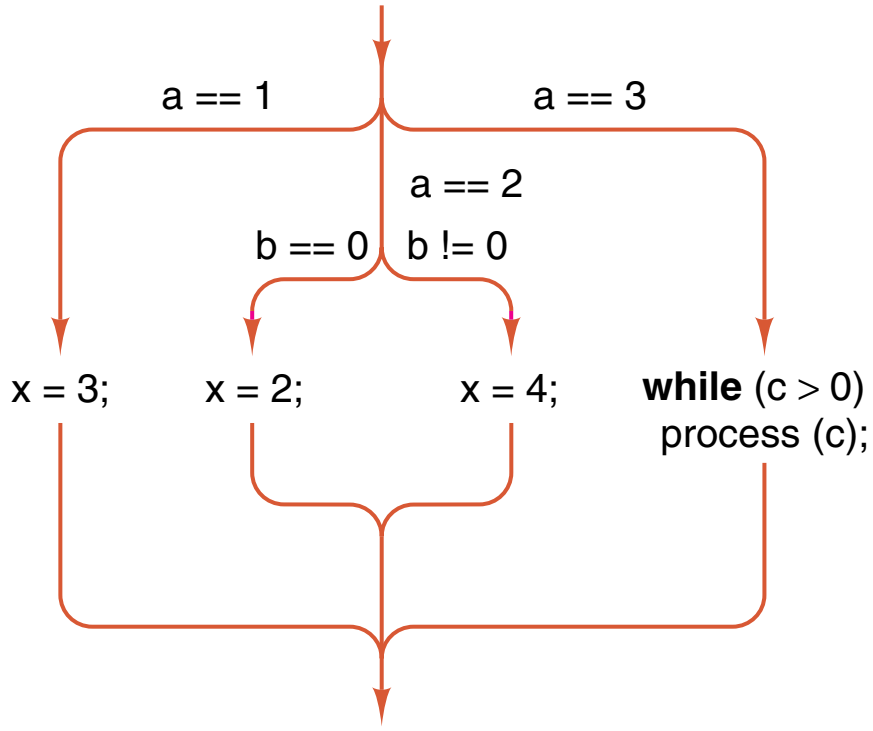
Program testing can be used to show the presence of bugs, but never their absence.

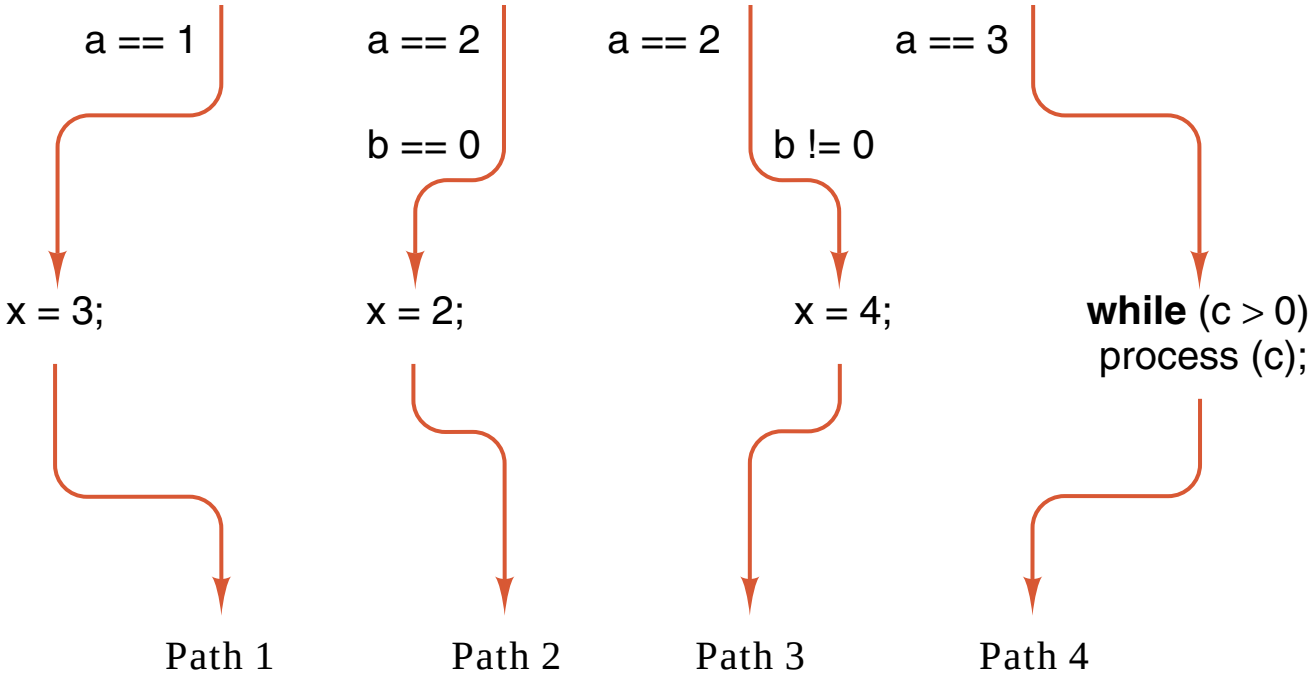
Methods for program testing:

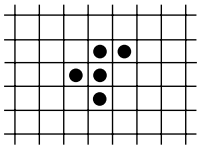
1. The ***black-box*** method:
   (a) Easy values
   (b) Typical, realistic values
   (c) Extreme values
   (d) Illegal values
2. The ***glass-box*** method: Trace all the paths through the program.
3. The ***ticking-box*** method: Don't do anything—Just hope for the best!

Execution Paths

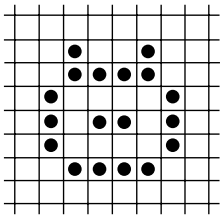
```
switch a {
   case 1: x = 3;
           break;
   case 2: if (b == 0)
              x = 2;
           else
              x = 4;
           break;
   case 3: while (c > 0)
              process (c);
           break;
}
```



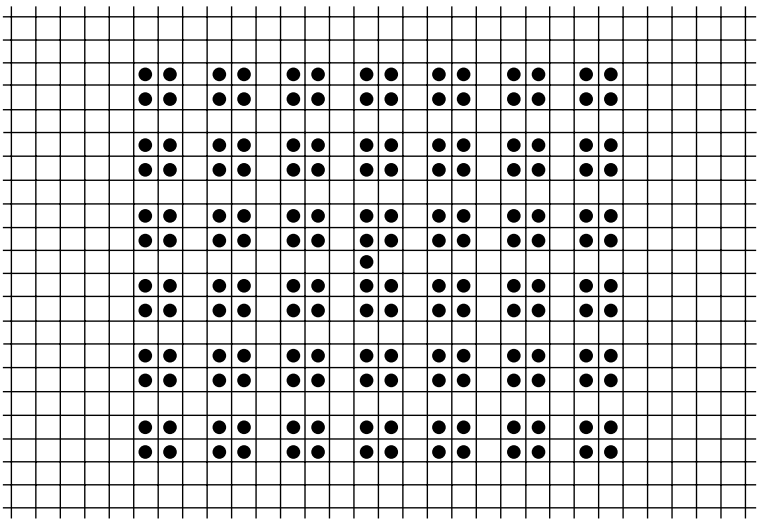




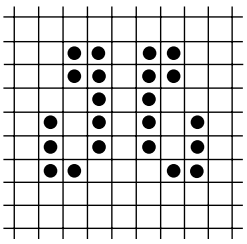
R Pentomino



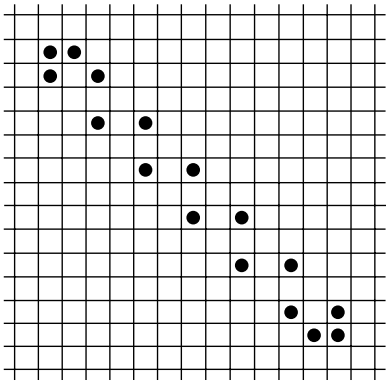
Cheshire Cat



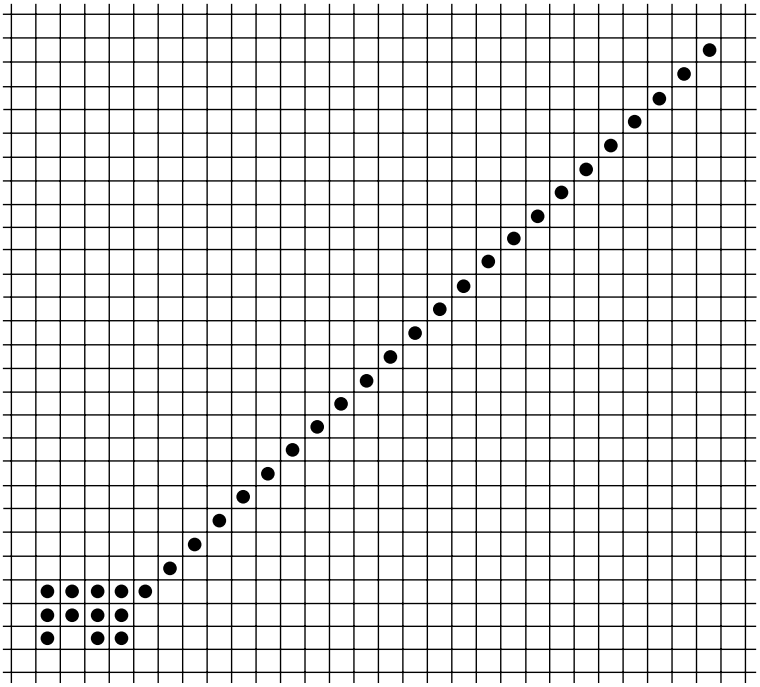
Virus



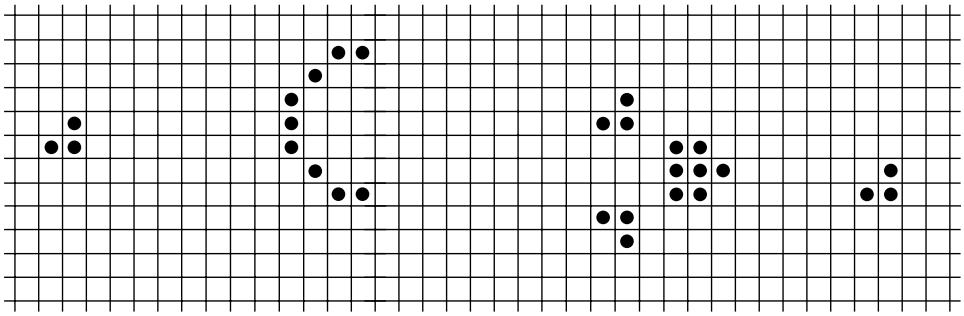
Tumbler



Barber Pole



Harvester



The Glider Gun

Program Maintenance

Programming Precept

For a large and important program, more than half the work comes in the maintenance phase, after it has been completely debugged, tested, and put into use.

1. Does the program solve the problem that is requested, following the problem specifications exactly?
2. Does the program work correctly under all conditions?
3. Does the program have a good user interface? Can it receive input in forms convenient and easy for the user? Is its output clear, useful, and attractively presented? Does the program provide alternatives and optional features to facilitate its use? Does it include clear and sufficient instructions and other information for the user?
4. Is the program logically and clearly written, with convenient classes and short functions as appropriate to do logical tasks? Are the data structured into classes that accurately reflect the needs of the program?
5. Is the program well documented? Do the names accurately reflect the use and meaning of variables, functions, types, and methods? Are precise pre- and postconditions given as appropriate? Are explanations given for major sections of code or for any unusual or difficult code?
6. Does the program make efficient use of time and of space? By changing the underlying algorithm, could the program's performance be improved?

Programming Precept

Be sure you understand your problem completely.
If you must change its terms, explain exactly what you have done.

Programming Precept

Design the user interface with the greatest care possible.
A program's success depends greatly on its attractiveness and ease of use.

Programming Precept

Do not optimize your code unless it is necessary to do so.
Do not start to optimize code until it is complete and correct.

Most programs spend 90 percent of their time
doing 10 percent of their instructions.
Find this 10 percent, and concentrate your efforts for efficiency there.

Programming Precept

Keep your algorithms as simple as you can.
When in doubt, choose the simple way.

Programming Precept

Sometimes postponing problems simplifies their solution.

Software Life Cycle

1. *Analyze* the problem precisely and completely. Be sure to *specify* all necessary user interface with care.

2. *Build* a prototype and *experiment* with it until all specifications can be finalized.

3. *Design* the algorithm, using the tools of data structures and of other algorithms whose function is already known.

4. *Verify* that the algorithm is correct, or make it so simple that its correctness is self-evident.

5. *Analyze* the algorithm to determine its requirements. Make sure that it meets the specifications.

6. *Code* the algorithm into the appropriate programming language.

7. *Test* and *evaluate* the program on carefully chosen test data.

8. *Refine* and *repeat* the foregoing steps as needed for additional functions until the software is complete and fully functional.

9. *Optimize* the code to improve performance, but only if necessary.

10. *Maintain* the program so that it will meet the changing needs of its users.

Problem Analysis

1. What form will the input and output data take? How much data will there be?

2. Are there any special requirements for the processing? What special occurrences will require separate treatment?

3. Will these requirements change? How? How fast will the demands on the system grow?

4. What parts of the system are the most important? Which must run most efficiently?

5. How should erroneous data be treated? What other error processing is needed?

6. What kinds of people will use the software? What kind of training will they have? What kind of user interface will be best?

7. How portable must the software be, to move to new kinds of equipment? With what other software and hardware systems must the project be compatible?

8. What extensions or other maintenance may be anticipated? What is the history of previous changes to software and hardware?

Requirements Specification

1. *Functional requirements* for the system: what it will do and what commands will be available to the user.

2. *Assumptions* and *limitations* on the system: what hardware will be used for the system, what form the input must take, the maximum size of input, the largest number of users, and so on.

3. *Maintenance requirements*: anticipated extensions or growth of the system, changes in hardware, changes in user interface.

4. *Documentation requirements*: what kind of explanatory material is required for what kinds of users.

The requirements specifications state *what* the software will do, not *how* it will be done.

Precepts on Coding

Programming Precept

Never code until the specifications are precise and complete.

Programming Precept

Act in haste and repent at leisure.
Program in haste and debug forever.

Programming Precept

Starting afresh is often easier than patching an old program.

Programming Precept

Always plan to build a prototype and throw it away.
You'll do so whether you plan to or not.

Pointers and Pitfalls

1. To improve your program, review the logic. Don't optimize code based on a poor algorithm.
2. Never optimize a program until it is correct and working.
3. Don't optimize code unless it is absolutely necessary.
4. Keep your functions short; rarely should any function be more than a page long.
5. Be sure your algorithm is correct before starting to code.
6. Verify the intricate parts of your algorithm.
7. Keep your logic simple.
8. Be sure you understand your problem before you decide how to solve it.
9. Be sure you understand the algorithmic method before you start to program.
10. In case of difficulty, divide a problem into pieces and think of each part separately.
11. The nouns that arise in describing a problem suggest useful classes for its solution; the verbs suggest useful functions.
12. Include careful documentation (as presented in Section 1.3.2) with each function as you write it.

13. Be careful to write down precise preconditions and postconditions for every function.

14. Include error checking at the beginning of functions to check that the preconditions actually hold.

15. Every time a function is used, ask yourself why you know that its preconditions will be satisfied.

16. Use stubs and drivers, black-box and glass-box testing to simplify debugging.

17. Use plenty of scaffolding to help localize errors.

18. In programming with arrays, be wary of index values that are off by 1. Always use extreme-value testing to check programs that use arrays.

19. Keep your programs well formatted as you write them—it will make debugging much easier.

20. Keep your documentation consistent with your code, and when reading a program make sure that you debug the code and not just the comments.

21. Explain your program to somebody else: Doing so will help you understand it better yourself.