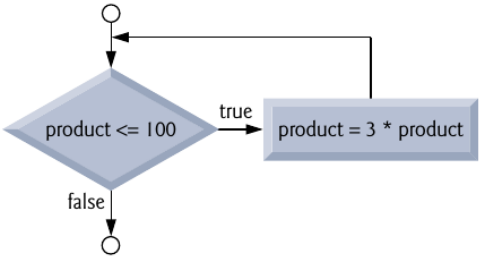
CC-211 Object Oriented Programming

From Procedures to Objects

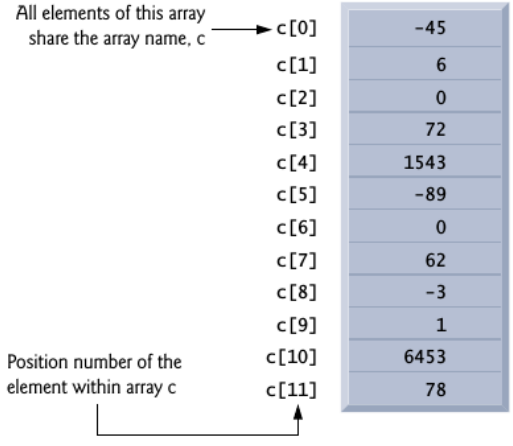
Nazar Khan
Department of Computer Science
University of the Punjab

Welcome to CC-211 Object Oriented Programming

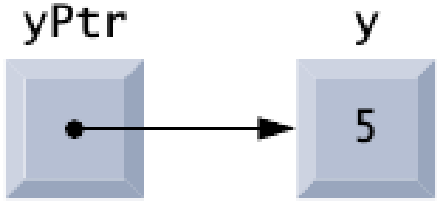
- CC-112 Programming Fundamentals gave us a strong foundation for *procedural programming*.



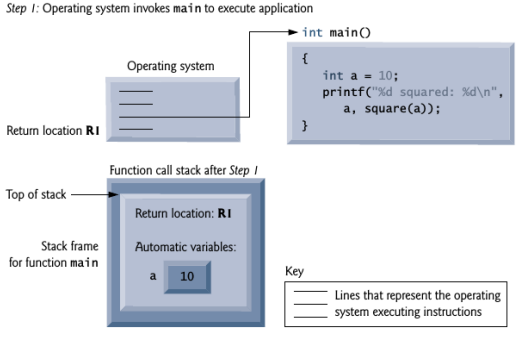
Loops and conditionals



Arrays and data



Addresses and pointers



Functions

```
FILE *fptr = fopen("data
      .txt", "r");
if (fptr == NULL) {
    printf("Error
          opening file\n"
          );
    return 1;
}
```

Files and streams

```
struct employee {
  char firstName[20];
  char lastName[20];
  unsigned int age;
  char gender;
  double hourlySalary;
  struct employee *
       teamLeaderPtr;
};
```

Structures

Problem with procedural programmming

- You can code the solution to any kind of problem using the knowledge gained in the PF course.
- But such programs can become *difficult to change and grow*.

> *Object oriented programming* is a different *way of thinking* about *problems* and *how to design solutions*.

> While we will program in C++, OOP is not just a transition from C to C++. It's a new way of thinking.

What this course is about

- The Programming Fundamentals course (CC-112) gave you a thorough treatment of
  - *procedural programming*, and
  - *top-down program design*

  in the C programming language.
- This Object Oriented Programming course (CC-211) will introduce two additional programming paradigms
  - *object-oriented programming* with classes, encapsulation, objects, operator overloading, inheritance and polymorphism, and
  - *generic programming* with function templates and class templates.

  in the C++ programming language.

> We will make *reusable components* of software.

C++: An enhanced version of C

- When a programming language becomes as entrenched as C, new requirements emerge.
- The language must evolve or simply be displaced by a new language.
- C++ was developed by Bjarne Stroustrup at Bell Laboratories.
- Originally called “C with classes”.
- The name C++ includes C’s increment operator (++) to indicate that C++ is an enhanced version of C.

A Simple C++ Program

```
// Addition program that displays the sum of two numbers.
#include <iostream> // allows program to perform input and
    output

int main()
{
   std::cout << "Enter first integer: "; // prompt user for
       data
   int number1;
   std::cin >> number1; // read first integer from user into
       number1

   std::cout << "Enter second integer: "; // prompt user for
       data
   int number2;
   std::cin >> number2; // read second integer from user into
       number2
```

A Simple C++ Program

```
    int sum = number1 + number2; // add the numbers; store
        result in sum
    std::cout << "Sum is " << sum << std::endl; // display sum;
        end line
}
```

Save the code above in sum_program.cpp.

To compile using gcc: `gcc sum_program.cpp -o sum_program -lstdc++`

To compile using g++: `g++ sum_program.cpp -o sum_program`

To run: `./sum_program`

Differences from C

- `printf` replaced by *stream insertion operator* `std::cout <<`
- `scanf` replaced by *stream extraction operator* `std::cin >>`
- *stream manipulation operator* `std::endl` outputs a newline, then "flushes" the output buffer.
- `return 0` not necessary

Stream insertion operator `std::cout`

- `cout` stands for *character output*.
- It is *not a function*, like `printf()`.
- It is C++'s standard output *stream* used to display characters on the console.
- `std::cout` means "`cout` inside the `std` *namespace*".

namespace

- A namespace is a space/scope containing names of variables, functions, etc.
- Inside the iostream file, you will find something similar to

```
1  #include <ostream>
2  #include <istream>
3
4  namespace std
5  {
6    extern istream cin;
7    extern ostream cout;
8  }
```

A student grading system

- A university needs a program for one student.
- It stores sessional (25%), midterm (35%), final exam (40%), and the student's name.
- It calculates and displays the final marks.

What data and operations do we need?

Think before coding

Data

- name
- sessional mark
- midterm mark
- final exam mark

Operations

- calculate final mark
- display result
- later: validate marks
- later: apply penalties

For this small problem, a structure and functions are a reasonable design.

Procedural solution: StudentRecord

```cpp
#include <iostream>
#include <string>

struct StudentRecord {
    std::string name;
    double sessional, midterm, finalExam;
};
double calculateFinalMark(const StudentRecord& student) {
    return student.sessional * 0.25
         + student.midterm * 0.35
         + student.finalExam * 0.40;
}
int main() {
    StudentRecord student{"Amina", 82.0, 74.0, 91.0};
    std::cout << student.name << ": "
              << calculateFinalMark(student) << '\\n';
    return 0;
}
```

What does this program do well?

- It groups related student data in one record.
- It gives the calculation a clear function.
- It is short, traceable and correct for the current requirement.
- It is a reasonable procedural solution.

Is this good code? Yes, for the problem we have given it.

Change can be problematic

- Is this program wrong? *No.*
- What makes it correct? *It represents the data and applies the formula correctly.*
- What changes if we add validation, attendance, late penalties, several grading schemes and reports?

A program can work today and still need a better design for tomorrow.

New requirements

The system must now enforce:

1. every mark is between 0 and 100
2. a late penalty may reduce the final-exam mark
3. a penalty must not make a mark negative, and
4. the grading formula is used consistently.

Where should these rules live?

A larger procedural version

```
#include <iostream>
#include <string>

struct StudentRecord {
    std::string name;
    double sessional, midterm, finalExam;
};

bool isValidMark(double mark) {
    return mark >= 0.0 && mark <= 100.0;
}

double calculateFinalMark(const StudentRecord& student) {
    return student.sessional * 0.25
         + student.midterm * 0.35
         + student.finalExam * 0.40;
}
```

A larger procedural version

```
void applyLatePenalty(StudentRecord& student, double penalty) {
    student.finalExam -= penalty;
    if (student.finalExam < 0.0) student.finalExam = 0.0;
}
int main() {
    StudentRecord student{"Amina", 82.0, 74.0, 91.0};
    if (!isValidMark(student.sessional)
        || !isValidMark(student.midterm)
        || !isValidMark(student.finalExam)) {
        std::cout << "Invalid mark\\n";
        return 1;
    }
    applyLatePenalty(student, 5.0);
    std::cout << student.name << ": "
              << calculateFinalMark(student) << '\\n';
    return 0;
}
```

What has become difficult?

- `main` can modify `finalExam` directly.
- Another function can bypass `applyLatePenalty`.
- Every update path must remember the validation rules.
- A changed grading policy may require edits in several places.
- The compiler will not necessarily detect a design rule being bypassed.

This is a design and responsibility problem, not a syntax error.

Questions about the change

- Can another function bypass the late-penalty rule? *Yes, fields are directly accessible.*
- What if the valid range changes from 0–100 to 0–50? *Several assumptions may need updating.*
- Is procedural programming the problem? *No, scattered responsibility is the problem. Everyone having access to everything can lead to chaos.*

Four design ideas

State What the entity currently knows: name and marks.

Behavior What it can do: calculate, update and display.

Rules What must remain true: valid marks and consistent grading.

Responsibility What this part of the program should control.

Which part of the program should protect the grading rules?

StudentRecord as a design concept

StudentRecord

State: name, sessional, midterm, final exam

Behavior: record a mark, calculate the result, apply a penalty

Rules: marks remain valid, penalties do not create invalid state

Which code should know and enforce these rules?

Responsibility

- Which code should know how the final mark is calculated?
- Which code should prevent an invalid mark?
- Why is it risky if every function can change every field?
- Is a function the same thing as a responsibility?

A function is an implementation mechanism; responsibility is a design decision about what a part of the program should know or control.

What is an object?

- An object is a meaningful program entity.
- It has state.
- It performs behavior.
- It has responsibilities.
- Its behavior should respect relevant rules.

An object is a design idea before it is a C++ syntax mechanism.

Objects do not have to be physical things

Useful objects may represent:

- a student record
- a bank account
- a calendar appointment
- a transaction
- a report
- a date or measurement

What state, behavior and responsibility belong together?

From procedures to objects

Procedural emphasis

- functions
- control flow
- data passed between operations

Object-oriented emphasis

- entities
- state and behavior
- responsibilities and rules

OOP adds a design vocabulary. It does not erase procedural programming.

A working definition of OOP

Object-oriented programming is a way of designing software around entities that combine related state and behavior, with clearly defined responsibilities and rules.

- It is useful when related state and behavior grow together.
- It can make rules easier to locate and protect.
- It does not automatically make every program better.

When should we use OOP?

- Not every small calculation needs to be considered as an object with behaviors.
- A procedural solution may be clearer for a short, stable task.
- OOP becomes useful when state changes over time, several operations depend on the same state, rules must be applied consistently, or the program must grow and be maintained.

OOP is a design choice, not a replacement for thinking.

The next step: classes

- Today we described a StudentRecord conceptually.
- C++ provides *classes* for expressing this kind of design.
- A class describes a kind of object.
- An object is an instance represented by that description.
- Next lecture we study the syntax and mechanics.

Design first: state, behavior, rules and responsibility.
Syntax follows.

Where this course is going

Procedural programming

↓

Why objects?

↓

Classes and objects

↓

Encapsulation and information hiding

↓

Object lifetime and relationships

↓

Operator overloading

↓

Inheritance and polymorphism

↓

Templates, STL and exceptions

Activity: find the responsibilities

Scenario: A library program stores a book title, author and checkout status. It displays, checks out and returns the book.

Identify

1. the book's state
2. its behavior
3. rules that should always be respected
4. a maintenance risk if every function can change every field, and
5. why a class might be useful later.

Activity worksheet and debrief

Category	Your response
State	
Behavior	
Rules	
Responsibility	
Maintenance risk	

Expected ideas: State is title, author and checkout status. Behavior is display, check out and return. A checked-out book cannot be checked out again. Book-related code should manage checkout state.

Discussion questions

1. Can procedural programming also be modular?
   Yes. Functions and separate modules are valuable forms of modularity.
2. What makes a program easier to change?
   Clear responsibilities, localized rules, low duplication and tests.
3. Does every line of an OOP program need to be inside an object?
   No. Use objects where they clarify the design.
4. What should we decide before writing a class?
   The entity, its state, behavior, rules and responsibilities.

What should you remember?

1. Procedural programming can solve the grading problem correctly.
2. Correct output is only one measure of software quality.
3. Growing systems can scatter state, behavior and rules across functions.
4. OOP helps us reason about meaningful entities and responsibilities.
5. An object combines related state and behavior conceptually.
6. We use OOP when it helps manage change and complexity.

If procedural programming works, why do we need OOP?
To organize changing state, behavior, rules and responsibility more clearly.

Preparation for Lecture 2

Choose a bank account, calendar appointment, shop product or library book. Identify its state, three behaviors, two rules, and why unrelated code should not freely change all of its state.

Do not write a class yet. Bring the design to Lecture 2.

Reading

- Deitel and Deitel, *C++ How to Program*, 10th Edition.
- Chapter 1: selected introductory material on C++, software development and OOP motivation.
- Chapter 3: selected introductory material on classes and objects for Lecture 2.

Next lecture

From the StudentRecord design

to C++ classes and objects

Today: why objects?
Next: how C++ expresses them.

Some C++ Programs

```
// Text-printing program.
#include <iostream> // enables program to output data to the
    screen

// function main begins program execution
int main() {
   std::cout << "Welcome to C++!\n"; // display message

   return 0; // indicate that program ended successfully
} // end function main
```

Some C++ Programs

```
// Printing a line of text with multiple statements.
#include <iostream> // enables program to output data to the
    screen

// function main begins program execution
int main() {
   std::cout << "Welcome ";
   std::cout << "to C++!\n";
} // end function main
```

Some C++ Programs

```
// Printing multiple lines of text with a single statement.
#include <iostream> // enables program to output data to the
    screen

// function main begins program execution
int main() {
   std::cout << "Welcome\nto\n\nC++!\n";
} // end function main
```

Some C++ Programs

```
// Addition program that displays the sum of two integers.
#include <iostream> // enables program to perform input and
    output

// function main begins program execution
int main() {
   // declaring and initializing variables
   int number1{0}; // first integer to add (initialized to 0)
   int number2{0}; // second integer to add (initialized to 0)
   int sum{0}; // sum of number1 and number2 (initialized to 0)

   std::cout << "Enter first integer: "; // prompt user for
       data
   std::cin >> number1; // read first integer from user into
       number1

   std::cout << "Enter second integer: "; // prompt user for
       data
```

Some C++ Programs

```
   std::cin >> number2; // read second integer from user into
       number2

   sum = number1 + number2; // add the numbers; store result in
        sum

   std::cout << "Sum is " << sum << std::endl; // display sum;
       end line
} // end function main
```

```
// Comparing integers using if statements, relational operators
// and equality operators.
#include <iostream> // enables program to perform input and
    output

using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl

// function main begins program execution
int main() {
   int number1{0}; // first integer to compare (initialized to
       0)
   int number2{0}; // second integer to compare (initialized to
        0)

   cout << "Enter two integers to compare: "; // prompt user
       for data
   cin >> number1 >> number2; // read two integers from user
```

```
if (number1 == number2) {
    cout << number1 << " == " << number2 << endl;
}

if (number1 != number2) {
    cout << number1 << " != " << number2 << endl;
}

if (number1 < number2) {
    cout << number1 << " < " << number2 << endl;
}

if (number1 > number2) {
    cout << number1 << " > " << number2 << endl;
}

if (number1 <= number2) {
    cout << number1 << " <= " << number2 << endl;
}
```

```
    if (number1 >= number2) {
        cout << number1 << " >= " << number2 << endl;
    }
} // end function main
```