

КОНСТРУЮВАННЯ БАГАТОФАЙЛОВИХ ПРОГРАМ

Мета роботи: вивчення основних директив препроцесорної обробки програм. Отримання практичних навиків у роботі зі структурами, функціями і модульною структурою програм.

1. Теоретичні відомості

Як тільки програми стають більше, їх слід розбивати на кілька файлів (з метою зручності і поліпшення функціональності). Розглянемо наступну програму, яка складається з двох файлів.

add.cpp:

```
1 int add(int x, int y)
2 {
3     return x + y;
4 }
```

main.cpp:

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "The sum of 3 and 4 is: " << add(3, 4) << std::endl;
6     return 0;
7 }
```

Спробуйте запустити цю програму. Вона не скомпілюється, ви отримаєте наступну помилку:

add: ідентифікатор не знайдено

При компіляції коду, компілятор не знає про існування функцій, які знаходяться в інших файлах. Це зроблено спеціально, щоб функції і змінні з однаковими іменами, але в різних файлах, не викликали конфлікт імен.

Проте, в даному випадку, потрібно, щоб **main.cpp** знав (і використовував) функцію **add()**, яка знаходиться в **add.cpp**. Для надання доступу **main.cpp** до функцій **add.cpp**, потрібно використовувати попереднє оголошення :

```

1 #include <iostream>
2
3 int add(int x, int y); // это нужно для того, чтобы main.cpp знал, что функция add() определена в другом месте
4
5 int main()
6 {
7     std::cout << "The sum of 3 and 4 is: " << add(3, 4) << std::endl;
8     return 0;
9 }

```

Тепер, коли компілятор буде компілювати **main.cpp**, він буде знати, що таке **add ()**.

Не пишіть наступний код в main.cpp:

```
#include "add.cpp"
```

Це призведе до того, що компілятор вставить весь вміст **add.cpp** безпосередньо в main.cpp замість того, щоб розглядати ці файли як окремі.

1.1 Заголовки

Як тільки програми стають більше і вже весь код не поміщається в декількох файлах, записувати кожен раз попередні оголошення для функцій, які хочемо використовувати, але які знаходяться в інших файлах, стає все незручно. Добре було б, якби всі попередні оголошення знаходилися б в одному місці.

Файли **.cpp** не є єдиними файлами в проектах. Є ще один тип файлів – заголовки, які мають розширення **.h**. Метою заголовних файлів є зручне зберігання набору оголошень об'єктів для їх подальшого використання в інших програмах.

Досить просто написати один заголовок і його можна буде повторно використовувати в будь-якій кількості програм. Також вносити зміни в такий код (наприклад, додавання ще одного параметра) набагато легше, ніж перевіряти по всіх файлах в пошуках використовуваних функцій.

Заголовки складаються з двох частин:

- директиви препроцесора, зокрема, header guards, які запобігають виклику заголовку більше одного разу з одного і того ж файлу;
- вміст заголовку : набір оголошень.

Всі заголовки, які написали самостійно, повинні мати розширення **.h**.
add.h:

```

1 // Начинаем с директив препроцессора. ADD_H - это произвольное уникальное имя (обычно используется имя заголовка)
2 #ifndef ADD_H
3 #define ADD_H
4
5 // А это уже содержимое заголовочного файла
6 int add(int x, int y); // прототип функции add() (не забывайте точку с запятой в конце!)
7
8 // Заканчиваем директивой препроцессора
9 #endif

```

Щоб використовувати файл **add.h** в **main.cpp**, вам спочатку потрібно буде підключити його до проекту.

main.cpp, в якому підключений файл **add.h**:

```
1 #include <iostream>
2 #include "add.h"
3
4 int main()
5 {
6     std::cout << "The sum of 3 and 4 is " << add(3, 4) << std::endl;
7     return 0;
8 }
```

add.cpp залишається без змін:

```
1 int add(int x, int y)
2 {
3     return x + y;
4 }
```

Коли компілятор зустрічає **#include "add.h"**, він копіює весь вміст **add.h** в поточний файл. Таким чином, ми отримуємо попереднє оголошення функції **add ()**.

Примітка : При підключення заголовка, весь його вміст вставляється відразу ж після рядка **#include**

1.2 Кутові дужки (<>) vs. Подвійні лапки ("")

Чому використовуються кутові дужки для **iostream** і подвійні лапки для **add.h**. Справа в тому, що, використовуючи кутові дужки, повідомляємо компілятору, що підключається заголовки написані не нами (вони є «системним», тобто надається стандартними бібліотеками C ++), так що шукати цей заголовки слід в системних директоріях. Подвійні лапки повідомляють компілятору, що підключаємо власні заголовки, які написані самостійно, тому шукати їх слід в поточній директорії проекту. Якщо файлу там не виявиться, то компілятор почне перевіряти інші шляхи, у тому числі і системні директорії.

Правило: кутові дужки використовуйте для підключення «системних» заголовних файлів та подвійні лапки для всього іншого (ваших заголовків файлів).

1.3 Директиви препроцесора

Препроцесор найкраще розглядати як окрему програму, яка виконується перед компіляцією. При запуску програми препроцесор переглядає код зверху вниз, файл за файлом, в пошуку директив. Директиви - це спеціальні команди, які починаються з символу **#** і не закінчуються крапкою з комою. Є кілька типів директив.

Директива **#include**

Коли ви **#include** файл, препроцесор копіює вміст підключається файлу в поточний файл відразу після рядка **#include**. Це дуже корисно при використанні певних даних (наприклад, попередні оголошення функцій) відразу в декількох місцях. Директива **#include** має дві форми:

- **#include <filename>**, яка повідомляє препроцесору шукати файл в системних шляхах.
- **#include "filename"**, яка повідомляє препроцесору шукати файл в поточній директорії проекту. Якщо його там не виявиться, то препроцесор почне перевіряти системні шляхи і будь-які інші, які ви вказали в налаштуваннях вашої IDE. Ця форма використовується для підключення користувальницьких заголовків файлів.

Директива **#define**

Директиву **#define** можна використовувати для створення макросів. Макрос - це правило, яке визначає конвертацію ідентифікатора в зазначені дані. Є два основних типи макросів: макроси-функції і макроси-об'єкти.

Макроси-функції поведуться як функції і використовуються в тих же цілях. Макроси-об'єкти можна визначити одним з двох наступних способів:

```
#define identifier  
#define identifier substitution_text
```

Коли препроцесор зустрічає макроси-об'єкти з `substitution_text`, то будь-яке подальше поява `identifier` замінюється на `substitution_text`. Ідентифікатор зазвичай пишеться великими літерами з символами підкреслення замість пробілів

Розглянемо наступний фрагмент коду:

```
1 #define MY_FAVORITE_NUMBER 9  
2  
3 std::cout << "My favorite number is: " << MY_FAVORITE_NUMBER << std::endl;
```

Препроцесор перетворює код вище в:

```
1 std::cout << "My favorite number is: " << 9 << std::endl;
```

Результат виконання:

My favorite number is: 9

1.4 Умовна компіляція

Директиви препроцесора умовної компіляції дозволяють визначити, за яких умов код компілюватиметься, а при яких - ні. У цьому уроці ми розглянемо тільки три директиви умовної компіляції:

```
#ifdef;  
#ifndef;  
#endif.
```

Директива **#ifdef** (англ. « If def ined» = «якщо визначено») дозволяє препроцесору перевірити, чи було значення раніше **#define**. Якщо так, то код між **#ifdef** і **#endif** скомпілюється. Якщо немає, то код буде проігнорований. наприклад:

```
1 #define PRINT_JOE  
2  
3 #ifdef PRINT_JOE  
4 std::cout << "Joe" << std::endl;  
5 #endif  
6  
7 #ifdef PRINT_BOB  
8 std::cout << "Bob" << std::endl;  
9 #endif
```

Оскільки **PRINT_JOE** вже був **#define**, то рядок **std::cout << "Joe" << std::endl;** скомпілюється і виконається. А оскільки **PRINT_BOB** ні **#define**, то рядок **std::cout << "Bob" << std::endl;** НЕ скомпілюється і, отже, не виконається.

Директива **#ifndef** (англ. « If n ot def ined» = «якщо не визначено») - це повна протилежність **#ifdef**, яка дозволяє перевірити, чи не було значення раніше визначено, наприклад:

```
1 #ifndef PRINT_BOB  
2 std::cout << "Bob" << std::endl;  
3 #endif
```

Результатом виконання цього фрагмента коду буде Bob, так як PRINT_BOB раніше ніколи не був `#define`.

2 Хід роботи

1. Напишіть однофайловий програму (з ім'ям `main.cpp`), яка запитує у користувача два цілих числа, додає їх, а потім виводить результат. У програмі має бути 3 функції:

- **`readNumber ()`**, яка запитує у користувача ціле число, а потім повертає його в `main ()`.
- **`writeAnswer ()`**, яка виводить результат на екран. Функція повинна бути без значення, що повертається і мати тільки один параметр.
- **`main ()`**, яка з'єднує все і вся.

Підказка №1 : Для виконання операції додавання не потрібно створювати окрему функцію (просто використовуйте оператор `+`).

Підказка №2 : Функцію **`readNumber ()`** потрібно викликати двічі.

Підказка №3: У Visual Studio першим рядком має бути **`#include "pch.h"`**(якщо ви використовуєте попередньо скомпільовані заголовки).

2. Змініть програму з завдання №1, щоб функції **`readNumber ()`** і **`writeAnswer ()`** знаходилися в окремому файлі **`io.cpp`**. Використовуйте попередні оголошення для доступу до цих функцій з функції **`main ()`**.

Підказка : Якщо у вас виникли проблеми, переконайтеся, що **`io.cpp`** правильно доданий до вашого проекту і підключений до компіляції

3. Змініть програму з завдання №2, щоб вона використовувала заголовки **`io.h`** для доступу до функцій (замість використання попередніх оголошень). Переконайтеся, що ваш заголовки використовує header guards.

3. Контрольні питання

1. Що таке директиви препроцесора?
2. Який синтаксис написання директив препроцесора?
3. Коли здійснюється обробка програми препроцесором?
4. Наведіть основні директиви препроцесора і поясніть їх дію.

5. У чому переваги використання багатофайлових програм?
6. Що таке роздільна компіляція?
7. Що рекомендується зберігати у заголовочних файлах, а що – у програмних файлах?

Список літератури

1. Буч Г., Максимчук Р., Энгл М. Объектно-ориентированный анализ и проектирование с примерами приложений. М.: ООО"ИДВильямс", 2008. 720 с.
2. Дейт Дж. Введение в объектно-ориентированное программирование. М.: Вильямс, 2005. 1328 с.
3. Чубук В. В., Чен Р. М., Павленко Л. А. Об'єктно-орієнтоване програмування у питаннях і відповідях : [навч. посібн.]. Х. : Вид. ХНЕУ, 2004. 288 с.
4. Элиенс А. Принципы объектно-ориентированной разработки программ. М.: Издательский дом «Вильямс», 2002. 496 с.
5. Уроки C++. [Электронный ресурс]. URL: <https://ravesli.com/uroki-cpp>.
6. Язык C++ / Объектно-ориентированное программирование. [Электронный ресурс]. URL: <https://prog-cpp.ru/oor/>.
7. Объектно-ориентированное программирование. [Электронный ресурс]. URL: <https://metanit.com/cpp/tutorial/5.1.php>