PROJEKTY WIELOPLIKOWE

1.Łączenie plików zewnętrznych:

1.1 Extern:

extern oznacza dla kompilatora ,że deklarujemy jakiś obiekt, zmienną globalną czy też funkcje która znajduje się w innym pliku, a zadaniem kompilatora jest by go odszukać np:

main.cpp

```
extern int x;
extern int y;
extern dodaj(x,y)
```

dane.cpp

```
int x = 5;
int y = 6;
int dodaj()
{
    return x+y;
}
```

1.2 Łączenie za pomocą bezpośredniego #include z plikiem cpp

Nie jest to nic trudnego wystarczy na początku programu użyć

```
#include “nazwa pliku.cpp”
```

Ale może też działać i bez tego, pod warunkiem że korzystamy z VS i mamy tez pliki w projekcie. Ale co jeżeli mamy we dwóch plikach po jednej funkcji o tej samej nazwie ? Robimy wtedy tak:

```
namespace p {
#include "plik.cpp"
}
```

Teraz możemy korzystać z funkcji z pliku “plik.cpp” w np ten sposób:

```
p::MyFunction();
```

Jeżeli nadal nie działa ,a korzystasz z VS to spróbuj usunąć pliki zewnętrzne z projektu.

1.3 Łączenie za pomocą pliku nagłówkowego

Po pierwsze osobiście zalecam używanie łączenia plików za pomocą nagłówków. Dlaczego ? Dlatego iż w pliku nagłówkowym możemy wypisać biblioteki z jakich będziemy korzystać np:
header.h

```
#include <iostream>
```

main.cpp

```
#include “header.h”

int main()
{
        std::cout<<”tutaj nie ma biblioteki!”<<endl;
        system(“pause”);
return 0;
}
```

jak wiemy **cout** należy do biblioteki standardowej iostream , mimo że nie ma w pliku main.cpp **#include <iostream>** to i tak można z tego korzystać dzięki połączeniu pliku nagłówkowego.
Łączenie plików cpp do plików nagłówkowych wygląda tak samo jak w poddziale wcześniej (1.2) tylko odbywa się w pliku nagłówkowym. Gdziekolwiek dodamy ten plik nagłówkowy będzie można z niego korzystać do woli.

2.Dyrektywy #define oraz #undef

#define oraz #undef to dyrektywy preprocesora czy taka jakby przednia straż kompilatora . Zanim kompilator rozpocznie swoją pracę,treść programu jest przeglądana przez preprocesor.

2.1 Dyrektywa #define ma postać:

#define <WYRAZ> <ciąg znaków które zastępuje>

Dyrektywa ta powoduje że gdziekolwiek wpiszemy <WYRAZ> kompilator będzie tam widział <ciąg znaków które zastępuje> które podaliśmy np jeżeli użyjemy tego gdzieś w programie :

#define JEDEN 1

to tam gdzie później użyjemy JEDEN kompilator będzie widział 1 czyli np:

int x = JEDEN;

kompilator widzi jako:

int x = 1;

#define może nam też służyć jako makrodefinicja ma wtedy postać:

#define <WYRAZ>(parametry) <wyrażenie>

np:

#define KWADRAT(a) ((a)*(a))

Działa to wtedy tak jak funkcja:

int a = KWADRAT(4);

kompilator widzi to już jako:

int a = ((4)*(4));

Nawiasy. Warto o tym też wspomnieć. Czemu aż tyle ? Wyobraź sobie coś takiego:

#define WYR(a,b) a+a*b+b
int x = WYR(2,4)*5;
Jak to widzi nasz kompilator? A tak :

int x = 2+2*4+4*5;

kolejność działania tego jest raczej oczywista:

2+(2*4)+(4*5)

Do czego można więc używać #define ? do deklaracji stałych i krótkich funkcji (inline). Ale dla deklaracji stałych jest lepszy sposób, wystarczy użyć constexpr.

constexpr int i = 5;

Dlaczego lepszy? ponieważ #define ma mniejsze szanse wykrycia naszych pomyłek.Więc tylko do krótkich funkcji ? Też nie koniecznie ponieważ przy takim wyrażeniu:
#define KWADRAT(a) ((a)*(a))
int i = 4;
int x = KWADRAT(i++);
Po czymś takim nasze zmienne będą wyglądać tak :
x = 20 , i = 6
Dlaczego ? Ponieważ i zostało inkrementowane dwa razy
int x = ((i++)*(i++));
Więc po co to nam ? Przydaje się to gdy chcemy oszukać kompilator.

2.2 Dyrektywa #undef

Ma ona postać:
#undef <WYRAZ>
Nie jest to nic trudnego. jest to po prostu zakończenie #define. Czyli po wpisaniu np:
#undef KWADRAT(a)
kompilator dalej już nie wie co to jest KWADRAT(a).